HP-UX Reference

Release 11.0

Library Functions

Section 3

Volume 4 of 5

Edition 1

B2355-90166

E1097

Printed in: United States

Legal Notices

The information in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Warranty. A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Restricted Rights Legend. Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DOD agencies, and subparagraphs (c) (1) and (c) (2) of the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 for other agencies.

HEWLETT-PACKARD COMPANY
3000 Hanover Street
Palo Alto, California 94304 U.S.A.

Use of this manual and flexible disk(s) or tape cartridge(s) supplied for this pack is restricted to this product only. Additional copies of the programs may be made for security and back-up purposes only. Resale of the programs in their present form or with alterations, is expressly prohibited.

Printing History

The manual printing date and part number indicate its current edition. The printing date will change when a new edition is printed. Minor changes may be made at reprint without changing the printing date. the manual part number will change when extensive changes are made.

Manual updates may be issued between editions to correct errors or document product changes. To ensure that you receive the updated or new editions, you should subscribe to the appropriate product support service. See your HP sales representative for details.

First Edition: October 1997 (HP-UX Release 11.0)

Volume Four
Table of Contents

Section 3

Volume Four
Table of Contents

Section 3

Table of Contents
Volume Four

Section 3: Library Functions

Table of Contents
Volume Four

Entry Name(Section): name **Description**

Table of Contents
Volume Four

Table of Contents
Volume Four

Entry Name(Section): name **Description**

Table of Contents
Volume Four

Entry Name(Section): name **Description**

Table of Contents
Volume Four

Entry Name(Section): name **Description**

Table of Contents
Volume Four

Entry Name(Section): name **Description**

Table of Contents
Volume Four

Entry Name(Section): name **Description**

Table of Contents
Volume Four

Entry Name(Section): name **Description**

Table of Contents
Volume Four

Entry Name(Section): name **Description**

Entry Name(Section): name **Description**

Table of Contents
Volume Four

Table of Contents
Volume Four

Table of Contents
Volume Four

Entry Name(Section): name **Description**

Table of Contents
Volume Four

Entry Name(Section): name **Description**

Table of Contents
Volume Four

Entry Name(Section): name **Description**

Table of Contents
Volume Four

Section 3

Library Functions

Section 3

Library Functions

NAME

intro - introduction to subroutines and libraries

DESCRIPTION

This section describes functions found in various libraries, other than those functions that directly invoke HP-UX system primitives, which are described in Section (2) of this volume. Certain major collections are identified by a letter after the section identifier (3):

(3C) These functions, together with the Operating System Calls and those marked (3S), constitute the Standard C Library, **libc**, which is automatically loaded by the C compiler, *cc*(1). Declarations for some of these functions can be obtained from **#include** files indicated in the appropriate entries.

(3G) These functions constitute the graphics library and are documented in separate manuals.

(3I) These functions constitute the instrument support (Device I/O) library.

(3M) These functions constitute the Math Library, **libm**. The link editor searches this library if the **-lm** option is specified. Declarations for these functions are available in the header files <**math.h**> and <**fenv.h**>. Several generally useful mathematical constants are also defined in <**math.h**> (see *math*(5)).

(3N) These functions are applicable to the Internet network and are part of the standard C library, **libc**.

(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 can be obtained from the **#include** file <**stdio.h**>.

(3T) These functions constitute the Pthreads Library. The link editor **ld** (see *ld*(1)) searches this library if the **-lpthread** option is specified. See *pthread*(3T) for more detailed information on threads.

(3X) Various specialized libraries. The files in which these libraries are found are specified in the appropriate entries.

DIAGNOSTICS

Functions in the C and Math Libraries, (3C) and (3M), may return the conventional values **0** or ±**HUGE_VAL** when the function is undefined for the given arguments or when the value is not representable. **HUGE_VAL** is defined as +INFINITY in the <**math.h**> header file. Functions in the Math Library may also return ±INFINITY or a NaN. In these cases, the external variable **errno** (see *errno*(2)) may also be set to the value [EDOM] or [ERANGE].

FILES

/usr/lib/libc.a	Standard I/O, operating system calls, and general purpose routines archive library.
/usr/lib/libc.sl	Standard I/O, operating system calls, and general purpose routines shared library.
/usr/lib/libcurses.sl	CRT screen handling shared library.
/usr/lib/libm.a	SVID3, XPG4.2, and ANSI C compliant math archive library.
/usr/lib/libm.sl	SVID3, XPG4.2, and ANSI C compliant math shared library.

SEE ALSO

ar(1), cc(1), ld(1), nm(1), intro(2), stdio(3S), hier(5), math(5), Introduction(9).

NAME

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

SYNOPSIS

```
#include <stdlib.h>
long int a64l(const char *s);
char *l64a(long int l);
```

Obsolescent Interface

```
int l64a_r(long int l, char *buffer, int buflen);
```

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.

The leftmost character is the least significant digit. For example,

$$a0 = (38 \times 64^0) + (2 \times 64^1) = 166$$

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()** uses 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.

Obsolescent Interface

l64a_r() converts between long integer and base-64 ASCII string.

APPLICATION USAGE

a64l() and **l64a()** are thread-safe. **a64l()** is async-cancel-safe. **l64a()** is not async-cancel-safe.

WARNINGS

The value returned by **l64a()** is a pointer into a buffer, the contents of which are overwritten subsequent calls by the same thread.

l64a_r() is an obsolescent interface supported only for compatibility with existing DCE applications. New multithreaded applications should use **l64a()**.

STANDARDS CONFORMANCE

a64l(): SVID2, SVID3
l64a(): SVID2, SVID3

NAME

abort( ) - generate a software abort fault

SYNOPSIS

```
#include <stdlib.h>
void abort(void);
```

DESCRIPTION

abort() first closes all open files, streams, directory streams, and message catalogue descriptors, if possible, then causes the signal **SIGABRT** to be sent to the calling process. This may cause a core dump to be generated (see *signal*(2)).

If the signal **SIGABRT** is caught, the handling function is executed. If the handling function returns, the action for **SIGABRT** is then reset to **SIG_DFL**, and the signal **SIGABRT** is sent again to the process to ensure that it terminates.

RETURN VALUE

abort() does not return.

ERRORS

No errors are defined.

APPLICATION USAGE

SIGABRT is not intended to be caught.

abort() is thread-safe. It is not async-cancel-safe.

DIAGNOSTICS

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

SEE ALSO

adb(1), exit(2), kill(2), signal(2). signal(5).

STANDARDS CONFORMANCE

abort(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

abs(), labs() - return integer absolute value

SYNOPSIS

```
#include <stdlib.h>
int abs(int i);
long int labs(long int i);
```

DESCRIPTION

abs() returns the absolute value of its integer operand.

labs() is similar to **abs()**, except that the argument and the returned value each have type **long int**.

The largest negative integer returns itself.

APPLICATION USAGE

The interfaces **abs()** and **labs()** are thread-safe and async-cancel-safe.

WARNINGS

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).

STANDARDS CONFORMANCE

abs(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

labs(): AES, SVID3, XPG4, ANSI C

NAME

acltostr() - convert access control list (ACL) structure to string form (HFS File Systems only)

SYNOPSIS

```
#include <acllib.h>

char *acltostr(int nentries, const struct acl_entry acl[], int form);
```

Obsolescent Interface

```
int acltostr_r(
    int nentries,
    const struct acl_entry acl[],
    int form,
    char *strbuf,
    int length);
```

Remarks:

To ensure continued conformance with emerging industry standards, features described in this manual entry are likely to change in a future release.

DESCRIPTION

acltostr() converts an access control list from structure form to string representation. **acltostr()** takes a pointer to the first element of an array of ACL entries (**acl**), containing the indicated number (*nentries*) of valid entries (zero or more), and the output form desired (**FORM_SHORT** or **FORM_LONG**). It returns a pointer to a static string (overwritten by the next call), which is a symbolic representation of the ACL, ending in a null character. The output forms are described in *acl*(5). In long form, the string returned contains newline characters.

A user ID of **ACL_NSUSER** and a group ID of **ACL_NSGROUP** are both represented by **%**. As with the **ls** command (see *ls*(1)), if an entry contains any other user ID or group ID value not listed in **/etc/passwd** or **/etc/group**, **acltostr()** returns a string equivalent of the ID number instead.

Just as in routines that manage the **/etc/passwd** file, **acltostr()** truncates user and group names to eight characters.

Note: **acltostr()** is complementary in function to **strtoacl()**.

Obsolescent Interface

acltostr_r() converts access control list (ACL) structure to string form.

APPLICATION USAGE

acltostr() is thread-safe. It is not async-cancel-safe.

RETURN VALUE

If **acltostr()** succeeds, it returns a pointer to a null-terminated string. If *nentries* is zero or less, the string is of zero length. If *nentries* is greater than **NACLENTRIES** (defined in <**sys/acl.h**>), or if *form* is an invalid value, the call returns (char *) NULL .

If **acltostr_r()** succeeds, it returns 0. If it fails, it returns -1 and sets **errno**. If *nentries* is zero or less, the string is of zero length.

ERRORS

[EINVAL] *strbuf* equals to **NULL**, or *nentries* is greater than **NACLENTRIES** (defined in <**sys/acl.h**>), or *form* is not one of the valid forms, or *length* of *strbuf* is too short.

[ERANGE] *length* is less than or equal zero.

EXAMPLES

The following code fragment reads the ACL on file **/users/ggd/test** and prints its short-form representation.

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

int nentries;
struct acl_entry acl [NACLENTRIES];
```

```
if ((nentries = getacl ("/users/ggd/test", NACLENTRIES, acl)) < 0)
     error (...);

fputs (acltostr (nentries, acl, FORM_SHORT), stdout);
```

a

WARNINGS

The value returned by **acltostr()** is a pointer into a buffer, the contents of which are overwritten by subsequent calls to **acltostr()** by the same thread.

acltostr_r() is an obsolescent interface supported only for compatibility with existing DCE applications. New multithreaded applications should use **acltostr()**.

DEPENDENCIES

acltostr() is only supported on HFS file system on standard HP-UX operating system.

AUTHOR

acltostr() was developed by HP.

FILES

/etc/passwd
/etc/group

SEE ALSO

getacl(2), setacl(2), cpacl(3C), chownacl(3C), setaclentry(3C), strtoacl(3C), acl(5).

NAME

acos(), acosf() - arccosine functions

SYNOPSIS

```
#include <math.h>
double acos(double x);
float acosf(float x);
```

DESCRIPTION

acos() returns the arccosine of *x* in the range 0 to π.

acosf() is a **float** version of **acos()**; it takes a **float** argument and returns a **float** result. To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options.

acosf() is not specified by any standard, but it is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, make sure your program includes <**math.h**>, and link in the math library by specifying **-lm** on the compiler or linker command line.

Millicode versions of the **acos()** function are available. Millicode versions of math library functions are usually faster than their counterparts in the standard library. To use these versions, compile your program with the **+Olibcalls** or the **+Oaggressive** optimization option.

If an error occurs, the millicode versions return the value described in the *RETURN VALUE* section, but do not set **errno**.

For more information, see the *HP-UX Floating-Point Guide*.

RETURN VALUE

If *x* is NaN, **acos()** returns NaN.

If the magnitude of *x* is greater than one, **acos()** returns NaN and sets **errno** to [EDOM].

ERRORS

If **acos()** fails, **errno** is set to the following value.

[EDOM] The magnitude of *x* is greater than one.

SEE ALSO

acosd(3M), asin(3M), atan(3M), atan2(3M), cos(3M), sin(3M), tan(3M), math(5).

STANDARDS CONFORMANCE

acos(): SVID3, XPG4.2, ANSI C

NAME

acosd(), acosdf() - degree-valued arccosine functions

SYNOPSIS

```
#include <math.h>
double acosd(double x);
float acosdf(float x);
```

DESCRIPTION

acosd() is a degree-valued version of the **acos()** function. It returns the arccosine of *x* in the range 0 to 180.

acosdf() is a **float** version of **acosd()**; it takes a **float** argument and returns a **float** result.

acosd() and **acosdf()** are not specified by any standard, but **acosdf()** is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

If *x* is NaN, **acosd()** returns NaN.

If the magnitude of *x* is greater than one, **acosd()** returns NaN and sets **errno** to [EDOM].

ERRORS

If **acosd()** fails, **errno** is set to the following value.

[EDOM] The magnitude of *x* is greater than one.

SEE ALSO

acos(3M), asind(3M), atand(3M), atan2d(3M), cosd(3M), sind(3M), tand(3M), math(5).

NAME

acosh() - inverse hyperbolic cosine function

SYNOPSIS

```
#include <math.h>
double acosh(double x);
```

DESCRIPTION

The **acosh()** function computes the inverse hyperbolic cosine of its argument.

The ISO/ANSI C committee has approved the **acosh()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

The **acosh()** function returns a value in the range +zero to +INFINITY.

If *x* is +INFINITY, **acosh()** returns +INFINITY.

If *x* is NaN, **acosh()** returns NaN.

If *x* < 1.0, **acosh()** returns NaN and sets **errno** to [EDOM].

ERRORS

If **acosh()** fails, **errno** is set to the following value.

[EDOM] *x* is less than 1.0.

SEE ALSO

asinh(3M), atanh(3M), cosh(3M), math(5).

STANDARDS CONFORMANCE

acosh(): SVID3, XPG4.2

NAME

add_wch, mvadd_wch, mvwadd_wch, wadd_wch — add a complex character and rendition to a window

SYNOPSIS

```
#include <curses.h>
int add_wch(const cchar_t *wch);
int wadd_wch(WINDOW *win, const cchar_t *wch);
int mvadd_wch(int y, int x, const cchar_t *wch);
int mvwadd_wch(WINDOW *win, int y, int x, const cchar_t *wch);
```

DESCRIPTION

These functions add information to the current or specified window at the current or specified position, and then advance the cursor. These functions perform wrapping. These functions perform special-character processing.

- If *wch* refers to a spacing character, then any previous character at that location is removed, a new character specified by *wch* is placed at that location with rendition specified by *wch*; then the cursor advances to the next spacing character on the screen.
- If *wch* refers to a non-spacing character, all previous characters at that location are preserved, the non-spacing characters of *wch* are added to the spacing complex character, and the rendition specified by *wch* is ignored.

RETURN VALUE

Upon successful completion, these functions return **OK**. Otherwise, they return **ERR**.

ERRORS

No errors are defined.

SEE ALSO

Rendition of Characters Placed into a Window in curses_intro(3X), addch(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

a

NAME

add_wchnstr, add_wchstr, mvadd_wchnstr, mvadd_wchstr, mvwadd_wchnstr, mvwadd_wchstr, wadd_wchnstr, wadd_wchstr — add an array of complex characters and renditions to a window

SYNOPSIS

```
#include <curses.h>
int add_wchnstr(const cchar_t *wchstr, int n);
int add_wchstr(const cchar_t *wchstr);
int wadd_wchnstr(WINDOW *win, const cchar_t *wchstr, int n);
int wadd_wchstr(WINDOW *win, const cchar_t *wchstr);
int mvadd_wchnstr(int y, int x, const cchar_t *wchstr, int n);
int mvadd_wchstr(int y, int x, const cchar_t *wchstr);
int mvwadd_wchnstr(WINDOW *win, int y, int x, const cchar_t *wchstr,
                   int n);
int mvwadd_wchstr(WINDOW *win, int y, int x, const cchar_t *wchstr);
```

DESCRIPTION

These functions write the array of **cchar_t** specified by *wchstr* into the current or specified window starting at the current or specified cursor position.

These functions do not advance the cursor. The results are unspecified if *wchstr* contains any special characters.

The functions end successfully on encountering a null **cchar_t**. The functions also end successfully when they fill the current line. If a character cannot completely fit at the end of the current line, those columns are filled with the background character and rendition.

The **add_wchnstr()**, **mvadd_wchnstr()**, **mvwadd_wchnstr()** and **wadd_wchnstr()** functions end successfully after writing *n* **cchar_t**s (or the entire array of **cchar_t**s, if *n* is –1).

RETURN VALUE

Upon successful completion, these functions return **OK**. Otherwise, they return **ERR**.

ERRORS

No errors are defined.

SEE ALSO

<curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

addch, mvaddch, mvwaddch, waddch — add a single-byte character and rendition to a window and advance the cursor

SYNOPSIS

```
#include <curses.h>
int addch(const chtype ch);
int mvaddch(int y, int x, const chtype ch);
int mvwaddch(WINDOW *win, int y, int x, const chtype ch);
int waddch(WINDOW *win, const chtype ch);
```

DESCRIPTION

The **addch()**, **mvaddch()**, **mvwaddch()** and **waddch()** functions place *ch* into the current or specified window at the current or specified position, and then advance the window's cursor position. These functions perform wrapping. These functions perform special-character processing.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise they return ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the A_ prefix.

SEE ALSO

Rendition of Characters Placed into a Window in curses_intro(3X), add_wch(3X), attroff(3X), doupdate(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4

The entry is rewritten for clarity. Also the type of argument *ch* is changed from **chtype** to **const chtype**.

(ENHANCED CURSES)

a

NAME

addchnstr, mvaddchnstr, mvwaddchnstr, waddchnstr — add length limited string of single-byte characters and renditions to a window

SYNOPSIS

```
#include <curses.h>
int addchnstr(const chtype *chstr, int n);
int mvaddchnstr(int y, int x, const chtype *chstr, int n);
int mvwaddchnstr(WINDOW *win, int y, int x, const chtype *chstr,
                 int n);
int waddchnstr(WINDOW *win, const chtype *chstr, int n);
```

DESCRIPTION

These functions overlay the contents of the current or specified window, starting at the current or specified position, with the contents of the array pointed to by *chstr* until a null **chtype** is encountered in the array pointed to by *chstr*.

These functions do not change the cursor position. These functions do not perform special-character processing. These functions do not perform wrapping.

These functions copy at most *n* items, but no more than will fit on the line. If *n* is –1 then the whole string is copied, to the maximum number that fit on the line.

RETURN VALUE

Upon successful completion, these functions return **OK**. Otherwise, they return **ERR**.

ERRORS

No errors are defined.

APPLICATION USAGE

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the A_ prefix.

SEE ALSO

addch(3X), add_wch(3X), add_wchstr(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

addchstr, mvaddchstr, mvwaddchstr, waddchstr — add string of single-byte characters and renditions to a window

SYNOPSIS

```
#include <curses.h>
int addchstr(const chtype *chstr);
int mvaddchstr(int y, int x, const chtype *chstr);
int mvwaddchstr(WINDOW *win, int y, int x, const chtype *chstr);
int waddchstr(WINDOW *win, const chtype *chstr);
```

DESCRIPTION

These functions overlay the contents of the current or specified window, starting at the current or specified position, with the contents of the array pointed to by *chstr* until a null **chtype** is encountered in the array pointed to by *chstr*.

These functions do not change the cursor position. These functions do not perform special-character processing. These functions do not perform wrapping.

RETURN VALUE

Upon successful completion, these functions return **OK**. Otherwise, they return **ERR**.

ERRORS

No errors are defined.

APPLICATION USAGE

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the A_ prefix.

SEE ALSO

addch(3X), addchnstr(3X), add_wch(3X), add_wchnstr(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

(ENHANCED CURSES)

NAME

addnstr, addstr, mvaddnstr, mvaddstr, mvwaddnstr, mvwaddstr waddnstr, waddstr — add a string of multi-byte characters without rendition to a window and advance cursor

SYNOPSIS

```
#include <curses.h>
int addnstr(const char *str, int n);
int addstr(const char *str);
int mvaddnstr(int y, int x, const char *str, int n);
int mvaddstr(int y, int x, const char *str);
int mvwaddnstr(WINDOW *win, int y, int x, const char *str, int n);
int mvwaddstr(WINDOW *win, int y, int x, const char *str);
int waddnstr(WINDOW *win, const char *str, int n);
int waddstr(WINDOW *win, const char *str);
```

DESCRIPTION

These functions write the characters of the string *str* on the current or specified window starting at the current or specified position using the background rendition.

These functions advance the cursor position. These functions perform special character processing. These functions perform wrapping.

The **addstr()**, **mvaddstr()**, **mvwaddstr()** and **waddstr()** functions are similar to calling **mbstowcs()** on *str*, and then calling **addwstr()**, **mvaddwstr()**, **mvwaddwstr()** and **waddwstr()**, respectively.

The **addnstr()**, **mvaddnstr()**, **mvwaddnstr()** and **waddnstr()** functions use at most *n* bytes from *str*. These functions add the entire string when *n* is –1. These functions are similar to calling **mbstowcs()** on the first *n* bytes of *str*, and then calling **addwstr()**, **mvaddwstr()**, **mvwaddwstr()** and **waddwstr()**, respectively.

RETURN VALUE

Upon successful completion, these functions return **OK**. Otherwise, they return **ERR**.

ERRORS

No errors are defined.

SEE ALSO

addnwstr(3X), mbstowcs() (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

In X/Open Curses, Issue 3, the **addstr()**, **mvaddstr()**, **mvwaddstr()** and **waddstr()** functions were described in the **addstr()** entry. In X/Open Curses, Issue 4, the type of the *str* argument defined for these functions is changed from **char *** to **char *const**, and the *DESCRIPTION* is changed to indicate that the functions will handle multi-byte sequences correctly.

NAME

addnwstr, addwstr, mvaddnwstr, mvaddwstr, mvwaddnwstr, mvwaddwstr, waddnwstr, waddwstr — add a wide-character string to a window and advance the cursor

SYNOPSIS

```
#include <curses.h>
int addnwstr(const wchar_t *wstr, int n);
int addwstr(const wchar_t *wstr);
int mvaddnwstr(int y, int x, const wchar_t *wstr, int n);
int mvaddwstr(int y, int x, const wchar_t *wstr);
int mvwaddnwstr(WINDOW *win, int y, int x, const wchar_t *wstr, int n);
int mvwaddwstr(WINDOW *win, int y, int x, const wchar_t *wstr);
int waddnwstr(WINDOW *win, const wchar_t *wstr, int n);
int waddwstr(WINDOW *win, const wchar_t *wstr);
```

DESCRIPTION

These functions write the characters of the wide character string *wstr* on the current or specified window at that window's current or specified cursor position.

These functions advance the cursor position. These functions perform special character processing. These functions perform wrapping.

The effect is similar to building a **cchar_t** from the **wchar_t** and the background rendition and calling **wadd_wch()** once for each **wchar_t** character in the string. The cursor movement specified by the *mv* functions occurs only once at the start of the operation.

The **addnwstr()**, **mvaddnwstr()**, **mvwaddnwstr()** and **waddnwstr()** functions write at most *n* wide characters. If *n* is –1, then the entire string will be added.

RETURN VALUE

Upon successful completion, these functions return **OK**. Otherwise, they return **ERR**.

ERRORS

No errors are defined.

SEE ALSO

add_wch(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

addsev() - define additional severities for formatting routines

SYNOPSIS

```
#include <pfmt.h>
int addsev(int sev, const char *sev_string);
```

DESCRIPTION

The **addsev()** routine allows the user to define additional severities to be used by formatting routines (see *pfmt*(3C)) in the standard message format. *sev* is the severity level. It must be between 5 and 255 inclusively. *sev_string* is a character string to be associated for this severity level. If the severity component of the *flags* parameter of the formatting routines matches *sev*, *sev_string* is printed as the severity string.

The **addsev()** routine may be called multiple times to set up a list of associations. If *sev* is already set, *sev_string* overwrites the previous string. If *sev_string* is NULL, the association is removed from the list.

addsev() assumes that *sev_string* has already been translated into a locale-specific string using the current locale.

APPLICATION USAGE

addsev() is thread-safe. It is not async-cancel-safe.

RETURN VALUE

If successful, **addsev()** returns zero. Otherwise, it returns **−1**.

EXAMPLE

```
#define MM_USER 10
addsev(MM_USER, "MY_NOTE");
pfmt(stdout, MM_USER|MM_GET,"my_appl_cat:1:The file is writable");
```

This example writes the following message to standard output:

```
MY_NOTE: The file is writable
```

SEE ALSO

pfmt(3C).

STANDARDS COMPLIANCE

addsev(): SVID3

NAME

asin(), asinf() - arcsine functions

SYNOPSIS

```
#include <math.h>
double asin(double x);
float asinf(float x);
```

DESCRIPTION

asin() returns the arcsine of *x* in the range $-\pi/2$ to $\pi/2$.

asinf() is a **float** version of **asin()**; it takes a **float** argument and returns a **float** result. To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options.

asinf() is not specified by any standard, but it is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, make sure your program includes <**math.h**>, and link in the math library by specifying **-lm** on the compiler or linker command line.

Millicode versions of the **asin()** function are available. Millicode versions of math library functions are usually faster than their counterparts in the standard library. To use these versions, compile your program with the **+Olibcalls** or the **+Oaggressive** optimization option.

If an error occurs, the millicode versions return the value described in the *RETURN VALUE* section, but do not set **errno**.

For more information, see the *HP-UX Floating-Point Guide*.

RETURN VALUE

If *x* is NaN, **asin()** returns NaN.

If the magnitude of *x* is greater than one, **asin()** returns NaN and sets **errno** to [EDOM].

ERRORS

If **asin()** fails, **errno** is set to the following value.

[EDOM] The magnitude of *x* is greater than one.

SEE ALSO

acos(3M), asind(3M), atan(3M), atan2(3M), cos(3M), sin(3M), tan(3M), math(5).

STANDARDS CONFORMANCE

asin(): SVID3, XPG4.2, ANSI C

NAME

asind(), asindf() - degree-valued arcsine functions

SYNOPSIS

```
#include <math.h>
double asind(double x);
float asindf(float x);
```

DESCRIPTION

asind() is a degree-valued version of the **asin()** function. It returns the arcsine of *x* in the range –90 to 90.

asindf() is a **float** version of **asind()**; it takes a **float** argument and returns a **float** result.

asind() and **asindf()** are not specified by any standard, but **asindf()** is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

If *x* is NaN, **asind()** returns NaN.

If the magnitude of *x* is greater than one, **asind()** returns NaN and sets **errno** to [EDOM].

ERRORS

If **asind()** fails, **errno** is set to the following value.

[EDOM] The magnitude of *x* is greater than one.

SEE ALSO

acosd(3M), asin(3M), atand(3M), atan2d(3M), cosd(3M), sind(3M), tand(3M), math(5).

NAME
asinh( ) - inverse hyperbolic sine function

SYNOPSIS
```
#include <math.h>
double asinh(double x);
```

DESCRIPTION
The **asinh()** function computes the inverse hyperbolic sine of its argument.

The ISO/ANSI C committee has approved the **asinh()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE
If *x* is ±INFINITY, **asinh()** returns ±INFINITY respectively.

If *x* is NaN, **asinh()** returns NaN.

ERRORS
No errors are defined.

SEE ALSO
acosh(3M), atanh(3M), sinh(3M), math(5).

STANDARDS CONFORMANCE
asinh(): SVID3, XPG4.2

NAME

assert() - verify program assertion

SYNOPSIS

```
#include <assert.h>

void assert(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, stops assertions from being compiled into the program.

APPLICATION USAGE

assert() is thread-safe. It is not async-cancel-safe.

WARNINGS

The expression argument used by **assert()** in compatibility mode cannot contain string literals or double quotes without escapes.

SEE ALSO

cpp(1), abort(3C).

STANDARDS CONFORMANCE

assert(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

atan(), atanf() - arctangent functions

SYNOPSIS

```
#include <math.h>
double atan(double x);
float atanf(float x);
```

DESCRIPTION

atan() returns the arctangent of *x* in the range $-\pi/2$ to $\pi/2$.

atanf() is a **float** version of **atan()**; it takes a **float** argument and returns a **float** result. To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options.

atanf() is not specified by any standard, but it is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, make sure your program includes <**math.h**>, and link in the math library by specifying **-lm** on the compiler or linker command line.

Millicode versions of the **atan()** function are available. Millicode versions of math library functions are usually faster than their counterparts in the standard library. To use these versions, compile your program with the **+Olibcalls** or the **+Oaggressive** optimization option.

For special cases, the millicode versions return the same values as their standard library counterparts (see the *RETURN VALUE* section).

For more information, see the *HP-UX Floating-Point Guide*.

RETURN VALUE

If *x* is ±INFINITY, **atan()** returns $\pm\pi/2$ respectively.

If *x* is NaN, **atan()** returns NaN.

If the correct value after rounding would be smaller in magnitude than **MINDOUBLE**, **atan()** returns zero.

ERRORS

No errors are defined.

SEE ALSO

acos(3M), asin(3M), atan2(3M), atand(3M), cos(3M), sin(3M), tan(3M), math(5), values(5).

STANDARDS CONFORMANCE

atan(): SVID3, XPG4.2, ANSI C

NAME

atan2(), atan2f() - arctangent-and-quadrant functions

SYNOPSIS

```
#include <math.h>
double atan2(double y, double x);
float atan2f(float y, float x);
```

DESCRIPTION

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.

atan2f() is a **float** version of **atan2()**; it takes **float** arguments and returns a **float** result. To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options.

atan2f() is not specified by any standard, but it is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, make sure your program includes <**math.h**>, and link in the math library by specifying **-lm** on the compiler or linker command line.

Millicode versions of the **atan2()** function are available. Millicode versions of math library functions are usually faster than their counterparts in the standard library. To use these versions, compile your program with the **+Olibcalls** or the **+Oaggressive** optimization option.

For special cases, the millicode versions return the same values as their standard library counterparts (see the *RETURN VALUE* section).

For more information, see the *HP-UX Floating-Point Guide*.

RETURN VALUE

If y and x are +INFINITY, **atan2()** returns $\pi/4$.

If y is +INFINITY and x is –INFINITY, **atan2()** returns $3*\pi/4$.

If y is –INFINITY and x is +INFINITY, **atan2()** returns $-\pi/4$.

If y and x are –INFINITY, **atan2()** returns $-3*\pi/4$.

If y is zero and x is greater than zero, **atan2()** returns zero.

If y is zero and x is less than zero, **atan2()** returns π.

If y is –zero and x is less than zero, **atan2()** returns $-\pi$.

If y is greater than zero and x is zero, **atan2()** returns $\pi/2$.

If y is less than zero and x is zero, **atan2()** returns $-\pi/2$.

If y/x would overflow, **atan2()** returns $\pm\pi/2$. The result will be $\pi/2$ if y is greater than zero and $-\pi/2$ if y is less than zero.

If y/x after rounding would be smaller in magnitude than **MINDOUBLE**, **atan2()** returns $\pm\pi$ or zero. The result is zero if x is greater than zero, π if x is less than zero and y is greater than zero, and $-\pi$ if x and y are both less than zero.

If both x and y are zero, **atan2()** returns NaN.

If x or y is NaN, **atan2()** returns NaN.

If the correct value after rounding would be smaller in magnitude than **MINDOUBLE**, **atan2()** returns zero.

ERRORS

No errors are defined.

SEE ALSO

acos(3M), asin(3M), atan(3M), atan2d(3M), cos(3M), sin(3M), tan(3M), math(5), values(5).

STANDARDS CONFORMANCE

atan2(): SVID3, XPG4.2, ANSI C

a

NAME

atan2d(), atan2df() - degree-valued arctangent-and-quadrant functions

SYNOPSIS

```
#include <math.h>
double atan2d(double y, double x);
float atan2df(float y, float x);
```

DESCRIPTION

atan2d() is a degree-valued version of the **atan2()** function. It returns the arctangent of *y*/*x*, in the range –180 to 180, using the signs of both arguments to determine the quadrant of the return value.

atan2df() is a **float** version of **atan2d()**; it takes **float** arguments and returns a **float** result.

atan2d() and **atan2df()** are not specified by any standard, but **atan2df()** is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

If *y* and *x* are +INFINITY, **atan2d()** returns 45.

If *y* is +INFINITY and *x* is –INFINITY, **atan2d()** returns 135.

If *y* is –INFINITY and *x* is +INFINITY, **atan2d()** returns –45.

If *y* and *x* are –INFINITY, **atan2d()** returns –135.

If *y* is zero and *x* is greater than zero, **atan2d()** returns zero.

If *y* is zero and *x* is less than zero, **atan2d()** returns 180.

If *y* is –zero and *x* is less than zero, **atan2d()** returns –180.

If *y* is greater than zero and *x* is zero, **atan2d()** returns 90.

If *y* is less than zero and *x* is zero, **atan2d()** returns –90.

If *y*/*x* would overflow, **atan2d()** returns ±90. The result will be 90 if *y* is greater than zero and –90 if *y* is less than zero.

If *y*/*x* after rounding would be smaller in magnitude than **MINDOUBLE**, **atan2d()** returns ±180 or zero. The result will be zero if *x* is greater than zero, 180 if *x* is less than zero and *y* is greater than zero, and –180 if *x* and *y* are both less than zero.

If both *x* and *y* are zero, **atan2d()** returns NaN.

If *x* or *y* is NaN, **atan2d()** returns NaN.

If the correct value after rounding would be smaller in magnitude than **MINDOUBLE**, **atan2d()** returns zero.

ERRORS

No errors are defined.

SEE ALSO

acosd(3M), asind(3M), atan2(3M), atand(3M), cosd(3M), sind(3M), tand(3M), math(5), values(5).

a

NAME

atand(), atandf() - degree-valued arctangent functions

SYNOPSIS

```
#include <math.h>
double atand(double x);
float atandf(float x);
```

DESCRIPTION

atand() is a degree-valued version of the **atan()** function. It returns the arctangent of *x* in the range –90 to 90.

atandf() is a **float** version of **atand()**; it takes a **float** argument and returns a **float** result.

atand() and **atandf()** are not specified by any standard, but **atandf()** is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

If *x* is ±INFINITY, **atand()** returns ±90 respectively.

If *x* is NaN, **atand()** returns NaN.

If the correct value after rounding would be smaller in magnitude than **MINDOUBLE**, **atand()** returns zero.

ERRORS

No errors are defined.

SEE ALSO

acosd(3M), asind(3M), atan(3M), atan2d(3M), cosd(3M), sind(3M), tand(3M), math(5), values(5).

NAME

atanh() - inverse hyperbolic tangent function

SYNOPSIS

```
#include <math.h>
double atanh(double x);
```

DESCRIPTION

The **atanh()** function computes the inverse hyperbolic tangent of its argument.

The ISO/ANSI C committee has approved the **atanh()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

If *x* is NaN, **atanh()** returns NaN.

If $|x| > 1.0$, **atanh()** returns NaN and sets **errno** to [EDOM].

If $|x| = 1.0$, **atanh()** returns $\pm$**HUGE_VAL** (according to the sign of *x*) and sets **errno** to [ERANGE].

ERRORS

If **atanh()** fails, **errno** is set to one of the following values.

[EDOM]	*x* has an absolute value greater than 1.0.
[ERANGE]	*x* has an absolute value equal to 1.0.

SEE ALSO

acosh(3M), asinh(3M), tanh(3M), math(5).

STANDARDS CONFORMANCE

atanh(): SVID3, XPG4.2

NAME

atexit - register a function to be called at program termination

SYNOPSIS

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

DESCRIPTION

atexit() registers the function *func* to be called, without arguments, at normal program termination. Functions registered by **atexit()** are called in reverse order of registration.

An **atexit()** call during exit processing is always unsuccessful.

The number of registered functions should not exceed **ATEXIT_MAX** as specified in <**limits.h**>.

When necessary, **crt0()** or **dld.sl()** (see *crt0*(3) and *dld.sl*(5)) registers one or more functions with **atexit()** to allow some processing at normal program termination. This registration occurs before any other.

RETURN VALUE

atexit() returns zero if the registration is successful; non-zero if unsuccessful.

SEE ALSO

exit(2), crt0(3), dld.sl(5).

STANDARDS CONFORMANCE

atexit(): AES, SVID3, XPG4, ANSI C

(ENHANCED CURSES)

a

NAME

attr_get, attr_off, attr_on, attr_set, color_set, wattr_get, wattr_off, wattr_on, wattr_set, wcolor_set — window attribute control functions

SYNOPSIS

```
#include <curses.h>
int attr_get(attr_t *attrs, short *color, void *opts);
int attr_off(attr_t attrs, void *opts);
int attr_on(attr_t attrs, void *opts);
int attr_set(attr_t attrs, short color, void *opts);
int color_set(short color, void *opts);
int wattr_get(WINDOW *win, attr_t *attrs, short *color, void *opts);
int wattr_off(WINDOW *win, attr_t attrs, void *opts);
int wattr_on(WINDOW *win, attr_t attrs, void *opts);
int wattr_set(WINDOW *win, attr_t attrs, short color, void *opts);
int wcolor_set(WINDOW *win, short color, void *opts);
```

DESCRIPTION

These functions manipulate the attributes and color of the window rendition of the current or specified window.

The **attr_get()** and **wattr_get()** functions obtain the current rendition of a window.

The **attr_off()** and **wattr_off()** functions turn off *attrs* in the current or specified window without affecting any others.

The **attr_on()** and **wattr_on()** functions turn on *attrs* in the current or specified window without affecting any others.

The **attr_set()** and **wattr_set()** functions set the window rendition of the current or specified window to *attrs* and *color*.

The **color_set()** and **wcolor_set()** functions set the window color of the current or specified window to color.

The *opts* argument is reserved for definition in a future edition of this document. Currently, the application must provide a null pointer as *opts*.

RETURN VALUE

These functions always return OK.

ERRORS

No errors are defined.

SEE ALSO

attroff(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

attroff, attron, attrset, wattroff, wattron, wattrset — restricted window attribute control functions

SYNOPSIS

```
#include <curses.h>
int attroff(int attrs);
int attron(int attrs);
int attrset(int attrs);
int wattroff(WINDOW *win, int attrs);
int wattron(WINDOW *win, int attrs);
int wattrset(WINDOW *win, int attrs);
```

DESCRIPTION

These functions manipulate the window attributes of the current or specified window.

The **attroff()** and **wattroff()** functions turn off *attrs* in the current or specified window without affecting any others.

The **attron()** and **wattron()** functions turn on *attrs* in the current or specified window without affecting any others.

The **attrset()** and **wattrset()** functions set the background attributes of the current or specified window to *attrs*.

It is unspecified whether these functions can be used to manipulate attributes other than **A_BLINK**, **A_BOLD**, **A_DIM**, **A_REVERSE**, **A_STANDOUT** and **A_UNDERLINE**.

RETURN VALUE

These functions always return either **OK** or **1**.

ERRORS

No errors are defined.

SEE ALSO

attr_get(3X), standend(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4

This entry is rewritten for clarity. The *DESCRIPTION* is updated to specify that it is undefined whether these functions can be used to manipulate attributes beyond those defined in X/Open Curses, Issue 3. The **standend()**, **standout()**, **wstandend()** and **wstandout()** functions are moved to the **standend()** entry.

NAME

basename(), dirname() - extract components of a path name

SYNOPSIS

```
#include <libgen.h>

char *basename(char *path);

char *dirname(char *path);
```

DESCRIPTION

basename() takes the path name pointed to by *path* and returns a pointer to the final component of the path name, deleting any trailing '/' characters. If the string consists entirely of '/' characters, **basename()** returns a pointer to the string "/". If *path* is a null pointer or points to the empty string, **basename()** returns a pointer to the string ".".

dirname() takes the path name pointed to by *path* and returns a pointer to a string that is a path name of the parent directory of that file. If *path* is a null pointer, points to the empty string, or does not contain a '/' character, then **dirname()** returns a pointer to the string ".".

RETURN VALUE

basename() returns a pointer to the final component of *path*.

dirname() returns a pointer to a string that is the parent directory of *path*.

EXAMPLES

The following code fragment calls **basename()** and **dirname()**.

```
#include <libgen.h>

const char *path="/usr/local/bin/foo";
char *base, *dir;
char *dup0, *dup1;
char *dup2, *dup3;

    /*Use strdup in case literals
    * are in text, or basename()
    * and dirname() modify the
    * input string.
    */

    dup0=strdup(path);
    dup1=strdup(path);

    base=basename(dup0);
    dir=dirname(dup1);

    /* Use strdup before modifying
    * return string in case a
    * pointer to a literal is
    * returned.
    */

    dup2=strdup(base);
    dup3=strdup(dir);
```

APPLICATION USAGE

basename() and **dirname()** are thread-safe. They are not async-cancel-safe.

WARNINGS

basename() and **dirname()** may overwrite *path*.

AUTHOR

basename() and **dirname()** were developed by HP.

SEE ALSO
basename(1).

STANDARDS CONFORMANCE
basename(): XPG4.2
dirname(): XPG4.2

b

NAME
baudrate — get terminal baud rate

SYNOPSIS
```
#include <curses.h>
int baudrate(void);
```

DESCRIPTION
The **baudrate()** function extracts the output speed of the terminal in bits per second.

RETURN VALUE
The **baudrate()** function returns the output speed of the terminal.

ERRORS
No errors are defined.

SEE ALSO
tcgetattr() (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The argument list is explicitly declared as **void**.

NAME
beep — audible signal

SYNOPSIS
```
#include <curses.h>
int beep(void);
```

DESCRIPTION
The **beep()** function alerts the user. It sounds the audible alarm on the terminal, or if that is not possible, it flashes the screen (visible bell). If neither signal is possible, nothing happens.

RETURN VALUE
The **beep()** function always returns OK.

ERRORS
No errors are defined.

APPLICATION USAGE
Nearly all terminals have an audible alarm, but only some can flash the screen.

SEE ALSO
flash(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The argument list is explicitly declared as **void**. The *RETURN VALUE* section is changed to indicate that the function always returns OK. The **flash()** function is moved to its own entry.

NAME

bgets() - read stream up to next delimiter

SYNOPSIS

```
#include <libgen.h>
char *bgets (char *buffer, size_t *count, FILE *stream,
const char *breakstring);
```

DESCRIPTION

bgets reads characters from *stream* into *buffer* until either *count* is exhausted or one of the characters in *breakstring* is encountered in the stream. The read data is terminated with a null byte ('**\0**') and a pointer to the trailing null is returned. If a *breakstring* character is encountered, the last non-null is the delimiter character that terminated the scan.

Note that, except for the fact that the returned value points to the **end** of the read string rather than to the beginning, the call

```
bgets(buffer, sizeof buffer, stream, \n);
```

is identical to

```
fgets(buffer, sizeof buffer, stream);
```

There is always enough room reserved in the buffer for the trailing null.

If *breakstring* is a null pointer, the value of *breakstring* from the previous call is used. If *breakstring* is null at the first call, no characters will be used to delimit the string.

RETURN VALUE

NULL is returned on error or end-of-file. Reporting the condition is delayed to the next call if any characters were read but not yet returned.

APPLICATION USAGE

bgets is thread-safe. It is not async-cancel-safe. A cancellation point may occur when a thread is executing **bgets**.

EXAMPLES

```
#include  <libgen.h>
char buffer[8];
/* read in first user name from /etc/passwd */
fp = fopen("/etc/passwd","r");
bgets(buffer, 8, fp, ":");
```

SEE ALSO

gets(3S).

NAME

bigcrypt - generate hashing encryption on large strings

SYNOPSIS

```
#include <hpsecurity.h>
#include <prot.h>

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

DESCRIPTION

bigcrypt() acts like *crypt*(3C), but handles much larger strings. **bigcrypt** takes the segments of cleartext and encrypts them individually, at first using the salt passed in, and then using the first two characters of the previous encrypted segment as the salt for the next segment. This avoids duplicated ciphertext chunks when the password characters are repeated, so that the encryption of a segment involves the encryption of all the previous segments.

Each chipertext segment is concatenated, with the salt at the beginning, to form the entire encrypted string.

AUTHOR

bigcrypt was developed by SecureWare Inc.

SEE ALSO

crypt(3C).

NAME

bindresvport() - bind socket to privileged IP port

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>

int bindresvport(int sd, struct sockaddr_in *sin);
```

DESCRIPTION

The **bindresvport()** function binds a socket descriptor to a privileged IP port; that is, a port number in the range 0 to 1023.

sd is a socket descriptor that was previously defined by a successful call to **socket()** (see *socket*(2)).

Upon successful completion of **bindresvport()**, the *sin_port* field in the **struct** pointed to by *sin* contains the number of the privileged port bound to the *sd* socket. Due to the need to protect the port numbers used by various networking commands, **bindresvport()** only returns a port number within a smaller subrange in the range of 0 to 1023.

Only superuser can bind to a privileged port.

RETURN VALUE

bindresvport() returns the following values:

0 Successful completion.
-1 Failure. **errno** is set to indicate the error.

ERRORS

If **bindresvport()** fails, **errno** is set to one of the following values:

[EACCES]	The requested address is protected, and the current user has inadequate permission to access it.
[EADDRINUSE]	The specified address is already in use.
[EADDRNOTAVAIL]	The specified address is bad or not available from the local machine.
[EAFNOSUPPORT]	Requested address does not match the address family of this socket.
[EBADF]	*sd* is not a valid descriptor.
[EINVAL]	The socket is already bound to an address, or the socket has been shut down.
[ENOBUFS]	Insufficient buffer memory is available.
[ENOTSOCK]	*sd* is not a socket.
[EOPNOTSUPP]	The socket whose descriptor is *sd* is of a type that does not support address binding.
[EPFNOSUPPORT]	The value specified in the *sin_family* field of the *sockaddr_in* struct was not **AF_INET**.

AUTHOR

bindresvport() was developed by Sun Microsystems, Inc.

SEE ALSO

bind(2), socket(2).

(ENHANCED CURSES)

NAME

bkgd, bkgdset, getbkgd, wbkgd, wbkgdset — set or get background character and rendition using a single-byte character

SYNOPSIS

```
#include <curses.h>
int bkgd(chtype ch);
void bkgdset(chtype ch);
chtype getbkgd(WINDOW *win);
int wbkgd(WINDOW *win, chtype ch);
void wbkgdset(WINDOW *win, chtype ch);
```

DESCRIPTION

The **bkgdset()** and **wbkgdset()** functions set the background property of the current or specified window based on the information in *ch*. If *ch* refers to a multi-column character, the results are undefined.

The **bkgd()** and **wbkgd()** functions set the background property of the current or specified window and then apply this setting to every character position in that window:

- The rendition of every character on the screen is changed to the new background rendition.
- Wherever the former background character appears, it is changed to the new background character.

The **getbkgd()** function extracts the specified window's background character and rendition.

RETURN VALUE

Upon successful completion, **bkgd()** and **wbkgd()** return OK. Otherwise, they return ERR.

The **bkgdset()** and **wbkgdset()** functions do not return a value.

Upon successful completion, **getbkgd()** returns the specified window's background character and rendition. Otherwise, it returns (**chtype**)ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the A_ prefix.

SEE ALSO

Rendition in curses_intro(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

(ENHANCED CURSES)

NAME

bkgrnd, bkgrndset, getbkgrnd, wbkgrnd, wbkgrndset, wgetbkgrnd — set or get background character and rendition using a complex character

SYNOPSIS

```
#include <curses.h>
int bkgrnd(const cchar_t *wch);
void bkgrndset(const cchar_t *wch);
int getbkgrnd(cchar_t *wch);
int wbkgrnd(WINDOW *win, const cchar_t *wch);
void wbkgrndset(WINDOW *win, const cchar_t *wch);
int wgetbkgrnd(WINDOW *win, cchar_t *wch);
```

DESCRIPTION

The **bkgrndset()** and **wbkgrndset()** functions set the background property of the current or specified window based on the information in *wch*.

The **bkgrnd()** and **wbkgrnd()** functions set the background property of the current or specified window and then apply this setting to every character position in that window:

- The rendition of every character on the screen is changed to the new background rendition.
- Wherever the former background character appears, it is changed to the new background character.

If *wch* refers to a non-spacing complex character for **bkgrnd()**, **bkgrndset()**, **wbkgrnd()** and **wbkgrndset()**, then *wch* is added to the existing spacing complex character that is the background character. If *wch* refers to a multi-column character, the results are unspecified.

The **getbkgrnd()** and **wgetbkgrnd()** functions store, into the area pointed to by *wch*, the value of the window's background character and rendition.

RETURN VALUE

The **bkgrndset()** and **wbkgrndset()** functions do not return a value.

Upon successful completion, the other functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

SEE ALSO

Rendition in curses_intro(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

blopen(), blclose(), blread(), blget(), blset() - terminal block-mode library interface

SYNOPSIS

```
#include <sys/blmodeio.h>
int blopen(int fildes);
int blclose(int bfdes);
int blread(int bfdes, char *buf, size_t nbyte);
int blget(int bfdes, struct blmodeio *arg);
int blset(int bfdes, const struct blmodeio *arg);
```

DESCRIPTION

This terminal library interface allows support of block-mode transfers with HP terminals. Block mode only affects input processing. Therefore, data is written with the standard **write()** interface (see *write*(2)).

In character mode, the terminal sends each character to the system as it is typed. However, in block mode, data is buffered and possibly edited locally in the terminal memory as it is typed, then sent as a block of data when the **[Enter]** key is pressed on the terminal. During block-mode data transmissions, the incoming data is not echoed by the interface and no special character processing is performed, other than recognizing a data block terminator character. For subsequent character mode transmissions, the existing termio state (see *termio*(7)) continues to determine echo and character processing.

Block-mode protocol has two component parts: block-mode handshake and block-mode transmission.

Block-Mode Handshake

At the beginning of a read, a *trigger* character is sent to the terminal to notify it that the system wants a block of data (the *trigger* character, if defined, is sent at the beginning of all reads, whether in character- or block-mode. It is necessary for block-mode reads to work correctly).

After receiving the *trigger* character, and when the user has typed all the data into the terminal's memory and pressed the **[Enter]** key, the terminal sends an *alert* character to the system to notify it that the terminal has a block of data to send.

The system might then send user-definable cursor-positioning or other data sequences to the terminal, such as for cursor-home or lock-keyboard.

The system then sends a second *trigger* character to the terminal. In response, the terminal transmits the data block as described in the Block-Mode Transmission section.

Block-Mode Transmission

The second part of the block-mode protocol is the block-mode transmission. After the block-mode handshake has successfully completed, the terminal transmits the data block to the system. During this transmission of data, the incoming data is not echoed by the system and no special character processing is performed, other than recognizing the data block termination character. It is possible to bypass the block-mode handshake and have the block-mode transmission occur after only the first *trigger* character is sent, see **CB_BMTRANS** below.

It is possible to intermix both character-mode and block-mode data transmissions. If **CB_BMTRANS** (see below) is set, all transfers are block-mode transfers. When **CB_BMTRANS** is not set, character mode transmissions are processed as described in *termio*(7). In this case, if an *alert* character is received anywhere in the input data, the transmission mode is automatically switched to block mode for a single transmission. Any data received before the *alert* is discarded. The *alert* character can be escaped with a backslash (\) character.

XON/XOFF Flow Control

To prevent data loss, XON/XOFF flow control should be used between the system and the terminal. The IXOFF bit (see *termio*(7)) should be set and the terminal strapped appropriately. If flow control is not used, it is possible for incoming data to overflow and be lost. (Note: some older terminals do not support XON/XOFF flow control.)

Read Requests

Read requests that receive data from block-mode transmissions do not return until the transmission is complete (the terminal has transmitted all characters). If the read is satisfied by byte count or if a data

transmission error occurs, all subsequent data is discarded until the transmission is complete. The read waits until a terminator character is seen, or until a time interval specified by the system has passed that is longer than necessary for the number of characters specified.

The data-block-terminator character is included in the data returned to the user, and is included in the byte count. If the number of bytes transferred by the terminal in a block-mode transfer exceeds the number of bytes requested by the user, the read returns the requested number of bytes and the remaining bytes are discarded. The user can determine if data was discarded by checking the last character of the returned data. If the last character is not the terminator character, then more data was received than was requested and data was discarded.

The EIO error can be caused by several events, including errors in transmission, framing, parity, break, and overrun, or if the internal timer expires. The internal timer starts when the second trigger character is sent by the computer, and ends when the terminating character is received by the computer. The length of this timer is determined by the number of bytes requested in the read and the current baud rate, plus an additional ten seconds.

User Control of Handshaking

If desired, the application program can provide its own handshake mechanism in response to the *alert* character by selecting the **OWNTERM** mode (see **CB_OWNTERM** below). With this mode selected, the driver completes a read request when the *alert* character is received. No data is discarded before the *alert*, and the *alert* is returned in the data read. The *alert* character may be escaped with a backslash (\) character. The second *trigger* is sent when the application issues the next read.

blmode Control Calls

First, the standard **open()** call to a tty device must be made to obtain a file descriptor for the subsequent block-mode control calls (an **open()** is done automatically by the system for **stdin** on the terminal).

int bfdes;

bfdes = blopen (int fildes)

A call to **blopen()** must be made before any block-mode access is allowed on the specified file descriptor. **blopen()** initializes the block-mode parameters as described below. The return value from **blopen()** is a block-mode file descriptor that must be passed to all subsequent block-mode control calls.

int blclose (int bfdes)

A call to **blclose()** must be issued before the standard **close()** to ensure proper closure of the device (see *close*(2)). Otherwise unpredictable results can occur. The argument *bfdes* is the file descriptor returned from a previous **blopen()** system call.

int blread (int bfdes, char *buf, size_t nbyte)

The **blread()** routine has the same parameters as the **read()** sytem call (see *read*(2)). At the beginning of a read, the **cb_trig1c** character (if defined) is sent to the device. If **CB_BMTRANS** is not set, and no **cb_alertc** character is received, the read data is processed according to *termio*(7). If **CB_BMTRANS** is set, or if a non-escaped **cb_alertc** character is received, echo is turned off for the duration of the transfer, and no further special character processing is done other than that required for the termination character. The argument *bfdes* is the file descriptor returned from a previous **blopen()** system call.

int blget (int bfdes, struct blmodeio *arg)

A call to **blget()** returns the current values of the **blmodeio** structure (see below). The argument *bfdes* is the file descriptor returned from a previous **blopen()** system call.

int blset (int bfdes, const struct blmodeio *arg)

A call to **blset()** sets the block-mode values from the structure whose address is *arg*. The argument *bfdes* is the file descriptor returned from a previous **blopen()** system call.

blmode Structure

The two block-mode control calls, **blget()** and **blset()**, use the following structure, defined in <**sys/blmodeio.h**>:

```
#define     NBREPLY     64
struct      blmodeio      {
    unsigned long     cb_flags;          /* Modes */
    unsigned char     cb_trig1c;         /* First trigger */
    unsigned char     cb_trig2c;         /* Second trigger */
```

```
        unsigned char     cb_alertc;          /* Alert character */
        unsigned char     cb_termc;           /* Terminating char */
        unsigned char     cb_replen;          /* cb_reply length */
        char              cb_reply[NBREPLY];  /* optional reply */
    };
```

The **cb_flags** field controls the basic block-mode protocol:

CB_BMTRANS	**0000001**	Enable mandatory block-mode transmission.
CB_OWNTERM	**0000002**	Enable user control of handshake.

If **CB_BMTRANS** is set, all transmissions are processed as block-mode transmissions. The block-mode handshake is not required and data read is processed as block-mode transfer data. The block-mode handshake can still be invoked by receipt of an *alert* character as the first character seen. A **blread()** issued with the **CB_BMTRANS** bit set causes any existing input buffer data to be flushed.

If **CB_BMTRANS** is not set, and if the *alert* character is defined and is detected anywhere in the input stream, the input buffer is flushed and the block-mode handshake is invoked. The system then sends the **cb_trig2c** character to the terminal, and a block-mode transfer follows. The *alert* character can be escaped by preceding it with a backslash (\).

If **CB_OWNTERM** is set, reads are terminated upon receipt of a non-escaped *alert* character. No input buffer flushing is performed, and the *alert* character is returned in the data read. This allows application code to perform its own block-mode handshaking. If the bit is clear, a non-escaped *alert* character causes normal block-mode handshaking to be used.

The initial **cb_flags** value is all-bits-cleared.

There are several special characters (both input and output) that are used with block mode. These characters and the initial values for these characters are described below. Any of these characters can be undefined by setting its value to 0377.

cb_trig1c (default DC1) is the initial *trigger* character sent to the terminal at the beginning of a read request.

cb_trig2c (default DC1) is the secondary *trigger* character sent to the terminal after the *alert* character has been seen.

cb_alertc (default DC2) is the *alert* character sent by the terminal in response to the first *trigger* character. It signifies that the terminal is ready to send the data block. The *alert* character can be escaped by preceding it with a backslash ("\").

cb_termc (default RS) is sent by the terminal after the block-mode transfer has completed. It signifies the end of the data block to the computer.

The **cb_replen** field specifies the length in bytes of the **cb_reply** field. If set to zero, the **cb_reply** string is not used. The **cb_replen** field is initially set to zero.

The **cb_reply** array contains a string to be sent out after receipt of the *alert* character, but before the second *trigger* character is sent by the computer. Any character can be included in the reply string. The number of characters sent is specified by **cb_replen**. The initial value of all characters in the **cb_reply** array is NULL.

RETURNS

If an error occurs, all calls return a value of –1 and **errno** is set to indicate the error. If no error is detected, **blread()** returns the number of characters read. All other calls return 0 upon successful completion.

During a read, it is possible for the user's buffer to be altered, even if an error value is returned. The data in the user's buffer should be ignored as it is not complete. The following errors can be returned by the library calls indicated:

blopen()

[ENOTTY] The file descriptor specified is not related to a terminal device.

blclose()

[ENOTTY] No previous **blopen** has been issued for the specified file descriptor.

blread()

[EDEADLK] A resource deadlock would occur as a result of this operation (see *lockf*(2)).

[EFAULT] **buf** points outside the allocated address space. The reliable detection of this error is implementation dependent.

[EINTR] A signal was caught during the **read** system call.

[EIO] An I/O error occurred during block-mode data transmissions.

[ENOTTY] No previous **blopen** has been issued for the specified file descriptor.

blget()

[ENOTTY] No previous **blopen** has been issued for the specified file descriptor.

blset()

[EINVAL] An illegal value was specified in the structure passed to the system.

[ENOTTY] No previous **blopen** has been issued for the specified file descriptor.

WARNINGS

blopen(), **blclose()**, **blread()**, **blget()** and **blset()** are not thread-safe.

Once **blopen** has been called with a file descriptor and returned successfully, that file descriptor should not subsequently be used as a parameter to the following system calls: **close()**, **dup()**, **dup2()**, **fcntl()**, **ioctl()**, **read()**, or **select()** until a **blclose** is called with the same file descriptor as its parameter. Additionally, **scanf()**, **fscanf()**, **getc()**, **getchar()**, **fgetc()**, and **fgetw()** should not be called for a stream associated with a file descriptor that has been used in a **blopen()** call but has not been used in a **blclose()** call. These functions call **read()**, and calling these routines results in unpredictable behavior.

AUTHOR

blopen(), **blclose()**, **blread()**, **blget()**, and **blset()** were developed by HP.

SEE ALSO

termio(7).

NAME

border, wborder — draw borders from single-byte characters and renditions

SYNOPSIS

```
#include <curses.h>

int border(chtype ls, chtype rs, chtype ts, chtype bs, chtype tl,
           chtype tr, chtype bl, chtype br);

int wborder(WINDOW *win, chtype ls, chtype rs, chtype ts, chtype bs,
            chtype tl, chtype tr, chtype bl, chtype br);
```

DESCRIPTION

The **border()** and **wborder()** functions draw a border around the edges of the current or specified window. These functions do not advance the cursor position. These functions do not perform special character processing. These functions do not perform wrapping.

The arguments in the left-hand column of the following table contain single-byte characters with renditions, which have the following uses in drawing the border:

Argument Name	Usage	Default Value
ls	Starting-column side	ACS_VLINE
rs	Ending-column side	ACS_VLINE
ts	First-line side	ACS_HLINE
bs	Last-line side	ACS_HLINE
tl	Corner of the first line and the starting column	ACS_ULCORNER
tr	Corner of the first line and the ending column	ACS_URCORNER
bl	Corner of the last line and the starting column	ACS_BLCORNER
br	Corner of the last line and the ending column	ACS_BRCORNER

If the value of any argument in the left-hand column is 0, then the default value in the right-hand column is used. If the value of any argument in the left-hand column is a multi-column character, the results are undefined.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the A_ prefix.

SEE ALSO

border_set(3X), box(3X), hline(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

(ENHANCED CURSES)

NAME

border_set, wborder_set — draw borders from complex characters and renditions

SYNOPSIS

```
#include <curses.h>

int border_set(const cchar_t *ls, const cchar_t *rs, const cchar_t *ts,
               const cchar_t *bs, const cchar_t *tl, const cchar_t *tr,
               const cchar_t *bl, const cchar_t *br);

int wborder_set(WINDOW *win, const cchar_t *ls, const cchar_t *rs,
               const cchar_t *ts, const cchar_t *bs,
               const cchar_t *tl, const cchar_t *tr,
               const cchar_t *bl, const cchar_t *br);
```

DESCRIPTION

The **border_set()** and **wborder_set()** functions draw a border around the edges of the current or specified window. These functions do not advance the cursor position. These functions do not perform special character processing. These functions do not perform wrapping.

The arguments in the left-hand column of the following table contain spacing complex characters with renditions, which have the following uses in drawing the border:

Argument Name	Usage	Default Value
ls	Starting-column side	WACS_VLINE
rs	Ending-column side	WACS_VLINE
ts	First-line side	WACS_HLINE
bs	Last-line side	WACS_HLINE
tl	Corner of the first line and the starting column	WACS_ULCORNER
tr	Corner of the first line and the ending column	WACS_URCORNER
bl	Corner of the last line and the starting column	WACS_BLCORNER
br	Corner of the last line and the ending column	WACS_BRCORNER

If the value of any argument in the left-hand column is a null pointer, then the default value in the right-hand column is used. If the value of any argument in the left-hand column is a multi-column character, the results are undefined.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

SEE ALSO

box_set(3X), hline_set(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME
box — draw borders from single-byte characters and renditions

SYNOPSIS
```
#include <curses.h>
int box(WINDOW *win, chtype verch, chtype horch);
```

DESCRIPTION
The **box()** function draws a border around the edges of the specified window. This function does not advance the cursor position. This function does not perform special character processing. This function does not perform wrapping.

The function **box(***win*, *verch*, *horch***)** has an effect equivalent to:

```
wborder(win, verch, verch, horch, horch, 0, 0, 0, 0);
```

RETURN VALUE
Upon successful completion, **box()** returns OK. Otherwise, it returns ERR.

ERRORS
No errors are defined.

APPLICATION USAGE
These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the A_ prefix.

SEE ALSO
border(3X), box_set(3X), hline(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The *DESCRIPTION* is changed to describe this function in terms of a call to the **wborder()** function.

(ENHANCED CURSES)

NAME

box_set — draw borders from complex characters and renditions

SYNOPSIS

```
#include <curses.h>
int box_set(WINDOW *win, const cchar_t *verch, const cchar_t *horch);
```

DESCRIPTION

The **box_set()** function draws a border around the edges of the specified window. This function does not advance the cursor position. This function does not perform special character processing. This function does not perform wrapping.

The function **box_set(***win*, *verch*, *horch***)** has an effect equivalent to:

```
wborder_set(win, verch, verch, horch, horch, NULL, NULL, NULL, NULL);
```

RETURN VALUE

Upon successful completion, this function returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

SEE ALSO

border_set(3X), hline_set(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

bsd_signal - simplified signal facilities

SYNOPSIS

```
#include <signal.h>

void (*bsd_signal(int sig, void (*func)(int)))(int);
```

DESCRIPTION

The **bsd_signal()** function provides a partially compatible interface for programs written to historical system interfaces (see *APPLICATION USAGE* below).

The function call **bsd_signal(sig,func)** has an effect as if implemented as:

```
void (*bsd_signal(int sig, void (*func)(int)))(int)
{
    struct sigaction act, oact;

    act.sa_handler = func;
    act.sa_flags = SA_RESTART;
    sigemptyset(&act.sa_mask);
    sigaddset(&act.sa_mask, sig);
    if (sigaction(sig, &act, &oact) == -1)
        return(SIG_ERR);
    return(oact.sa_handler);
}
```

The handler function should be declared:

```
void handler(int sig);
```

where *sig* is the signal number. The behaviour is undefined if *func* is a function that takes more than one argument, or an argument of a different type.

RETURN VALUE

Upon successful completion, **bsd_signal()** returns the previous action for *sig*. Otherwise, SIG_ERR is returned and **errno** is set to indicate the error.

ERRORS

Refer to **sigaction()**.

APPLICATION USAGE

This function is a direct replacement for the BSD **signal()** function for simple applications that are installing a single-argument signal handler function. If a BSD signal handler function is being installed that expects more than one argument, the application has to be modified to use **sigaction()**. The **bsd_signal()** function differs from **signal()** in that the SA_RESTART flag is set and the SA_RESETHAND will be clear when **bsd_signal()** is used. The state of these flags is not specified for **signal()**.

SEE ALSO

sigaction(2), sigaddset(3C), sigemptyset(3C), signal(2), <signal.h>.

NAME

killpg, getpgrp, setpgrp, sigvec, signal - 4.2 BSD-compatible process control facilities

SYNOPSIS

```
#include <signal.h>

int killpg(int pgrp, int sig);

int getpgrp(int pid);

int setpgrp(int pid, int pgrp);

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

void (*signal(int sig, void (*func)(int)))(int);
```

DESCRIPTION

These calls simulate (and are provided for backward compatibility with) functions of the same name in the 4.2 Berkeley Software Distribution.

This version of **setpgrp()** is equivalent to the system call **setpgid(** *pid*, *pgrp* **)** (see *setpgid*(2)).

This version of **getpgrp()** is equivalent to the system call **getpgrp2(** *pid* **)** (see *getpid*(2)).

killpg() is equivalent to the system call **kill(-** *pgrp*, *sig* **)** (see *kill*(2)).

sigvec() is equivalent to the system call **sigvector(** *sig*, *vec*, *ovec* **)** (see *sigvector*(2)), except for the following:

When SIGCHLD or SIGCLD is used and *vec* specifies a catching function, the routine acts as if the **SV_BSDSIG** flag were included in the **sv_flags** field of *vec*.

The name **sv_onstack** can be used as a synonym for the name of the **sv_flags** field of *vec* and *ovec*.

If *vec* is not a null pointer and the value of (*vec–>sv_flags* & 1) is "true", the routine acts as if the **SV_ONSTACK** flag were set.

If *ovec* is not a null pointer, the flag word returned in *ovec–>sv_flags* (and therefore the value of *ovec–>sv_onstack*) will be equal to 1 if the system was reserving space for processing of that signal because of a call to *sigspace*(2), and 0 if not. The **SV_BSDSIG** bit in the value placed in *ovec–>sv_flags* is always clear.

If the reception of a caught signal occurs during certain system calls, the call will always be restarted, regardless of the return value from a catching function installed with **sigvec()**. The affected calls are *wait*(2), *semop*(2), *msgsnd*(2), *msgrcv*(2), and *read*(2) or *write*(2) on a slow device (such as a terminal or pipe, but not a file). Other interrupted system calls are not restarted.

This version of **signal()** has the same effect as **sigvec(**sig, *vec*, *ovec***)**, where *vec–>sv_handler* is equal to *func*, *vec–>sv_mask* is equal to 0, and *vec–>sv_flags* is equal to 0. **signal()** returns the value that would be stored in *ovec–>sv_handler* if the equivalent **sigvec()** call would have succeeded. Otherwise, **signal()** returns –1 and **errno** is set to indicate the reason as it would have been set by the equivalent call to **sigvec()**.

WARNINGS

While the 4.3 BSD release defined extensions to some of the interfaces described here, only the 4.2 BSD interfaces are emulated by this package.

bsdproc() should not be used in conjunction with the facilities described under *sigset*(3C).

APPLICATION USAGE

Threads Considerations

The signal disposition (such as **catch/ignore/default**) established by **sigvec()** and **signal()** is shared by all threads in the process.

For more information regarding signals and threads, refer to *signal*(5).

AUTHOR

bsdproc() was developed by HP and the University of California, Berkeley.

SEE ALSO

ld(1), kill(2), getpid(2), msgsnd(2), msgrcv(2), read(2), semop(2), setpgid(2), setsid(2), sigvector(2), wait(2), write(2), sigset(3C), sigstack(2), signal(5).

b

NAME

bsearch() - binary search a sorted table

SYNOPSIS

```
#include <stdlib.h>

void *bsearch(
   const void *key,
   const void *base,
   size_t nel,
   size_t size,
   int (*compar)(const void *, const void *)
);
```

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. *size* is the size of each element 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 indicating that the first argument (the key) is to be considered less than, equal to, or greater than the second argument (the array element).

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-void.

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

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

APPLICATION USAGE

bsearch() is thread-safe and async-cancel-safe.

RETURN VALUE

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

EXAMPLES

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 <stdlib.h>
#include <string.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;
    /* routine to compare 2 nodes */
    int node_compare(const void *, const void *);
    char str_space[20];        /* space to read string into */
```

```
        .
        .
        .
        node.string = str_space;
        while (scanf("%s", node.string) != EOF) {
            node_ptr = (struct node *)bsearch((void *)(&node),
                   (void *)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(const void *node1, const void *node2)
    struct node *node1, *node2;
    {
        return strcoll(((const struct node *)node1)->string,
                    ((const struct node *)node2)->string);
    }
```

WARNINGS

If the table being searched contains two or more entries that match the selection criteria, a random entry is returned by **bsearch()** as determined by the search algorithm.

SEE ALSO

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

STANDARDS CONFORMANCE

bsearch(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

bufsplit() - split buffer into fields

SYNOPSIS

```
#include <libgen.h>
size_t bufsplit(char *buf, size_t n, char **a);
```

DESCRIPTION

bufsplit examines the buffer, *buf*, and assigns values to the pointer array, *a*, so that the pointers point to the first *n* fields in *buf* that are delimited by tabs or new-lines.

To change the characters used to separate fields, call **bufsplit** with *buf* pointing to the string of characters, and *n* and *a* set to zero. For example, to use '**:**', '**.**', and '**,**' as separators along with tab and new-line:

```
bufsplit(":.,\t\n", 0, (char**)0);
```

RETURN VALUE

The number of fields assigned in the array *a*. If *buf* is zero, the return value is zero and the array is unchanged. Otherwise the value is at least one. The remainder of the elements in the array are assigned the address of the null byte at the end of the buffer.

APPLICATION USAGE

bufsplit is not thread safe. It is async-cancel-safe.

EXAMPLES

```
/*
 * set a[0] = "This", a[1] = "is", a[2] = "a",
 * a[3] = "test"
 */
bufsplit("This\tis\ta\ttest\n", 4, a);
```

WARNINGS

bufsplit changes the delimiters to null bytes in *buf*.

NAME

htonl(), htons(), ntohl(), ntohs() - convert values between host and network byte order

SYNOPSIS

```
#include <netinet/in.h>

_XOPEN_SOURCE_EXTENDED only
#include <arpa/inet.h>

unsigned long htonl(unsigned long hostlong);

unsigned short htons(unsigned short hostshort);

unsigned long ntohl(unsigned long netlong);

unsigned short ntohs(unsigned short netshort);
```

DESCRIPTION

These routines convert 16- and 32-bit quantities between network byte order and host byte order. On HP-UX systems, network and host byte orders are identical, so these routines are defined as null macros in the include file <**netinet/in.h**>. If _XOPEN_SOURCE_EXTENDED is defined then these routines are defined in the include file <**arpa/inet.h>.**

These routines are most often used in conjunction with Internet addresses and ports as returned by **gethostent()** and **getservent()** (see *gethostent*(3N) and *getservent*(3N)). Use these routines to write portable programs.

AUTHOR

byteorder() was developed by the University of California, Berkeley.

SEE ALSO

gethostent(3N), getservent(3N).

STANDARDS CONFORMANCE

byteorder(): XPG4

NAME

can_change_color, color_content, has_colors, init_color, init_pair, start_color, pair_content — color manipulation functions

SYNOPSIS

```
#include <curses.h>
bool can_change_color(void);
int color_content(short color, short *red, short *green, short *blue);
int COLOR_PAIR(int n);
bool has_colors(void);
int init_color(short color, short red, short green, short blue);
int init_pair(short pair, short f, short b);
int pair_content(short pair, short *f, short *b);
int PAIR_NUMBER(int value);
int start_color(void);
extern int COLOR_PAIRS;
extern int COLORS;
```

DESCRIPTION

These functions manipulate colour on terminals that support colour.

Querying Capabilities

The **has_colors()** function indicates whether the terminal is a colour terminal. The **can_change_color()** function indicates whether the terminal is a colour terminal on which colours can be redefined.

Initialisation

The **start_color()** function must be called in order to enable use of colours and before any colour manipulation function is called. The function initialises eight basic colours (black, blue, green, cyan, red, magenta, yellow, and white) that can be specified by the colour macros (such as COLOR_BLACK) defined in **<curses.h>**. (See *Colour-related Macros* in **<curses.h>**.) The initial appearance of these eight colours is not specified.

The function also initialises two global external variables:

- COLORS defines the number of colours that the terminal supports. (See *Colour Identification* below.) If COLORS is 0, the terminal does not support redefinition of colours (and **can_change_colour()** will return FALSE).
- COLOR_PAIRS defines the maximum number of colour-pairs that the terminal supports. (See *User-Defined Colour Pairs* below.)

The **start_color()** function also restores the colours on the terminal to terminal-specific initial values. The initial background colour is assumed to be black for all terminals.

Colour Identification

The **init_color()** function redefines colour number *color*, on terminals that support the redefinition of colours, to have the red, green, and blue intensity components specified by *red*, *green*, and *blue*, respectively. Calling **init_color()** also changes all occurrences of the specified colour on the screen to the new definition.

The **color_content()** function identifies the intensity components of colour number *color*. It stores the red, green, and blue intensity components of this colour in the addresses pointed to by *red*, *green*, and *blue*, respectively.

For both functions, the *color* argument must be in the range from 0 to and including COLORS – 1. Valid intensity values range from 0 (no intensity component) up to and including 1000 (maximum intensity in that component).

User-Defined Colour Pairs

Calling **init_pair()** defines or redefines colour-pair number *pair* to have foreground colour *f* and background colour *b*. Calling **init_pair()** changes any characters that were displayed in the colour pair's old definition to the new definition and refreshes the screen.

After defining the colour pair, the macro COLOR_PAIR(*n*) returns the value of colour pair *n*. This value is the colour attribute as it would be extracted from a **chtype**. Conversely, the macro PAIR_NUMBER(*value*) returns the colour pair number associated with the colour attribute *value*.

C

The **pair_content()** function retrieves the component colours of a colour-pair number *pair*. It stores the foreground and background colour numbers in the variables pointed to by *f* and *b*, respectively.

With **init_pair()** and **pair_content()**, the value of *pair* must be in a range from 0 to and including COLOR_PAIRS – 1. (There may be an implementation-specific lower limit on the valid value of *pair*, but any such limit is at least 63.) Valid values for *f* and *b* are the range from 0 to and including COLORS – 1.

RETURN VALUE

The **has_colors()** function returns TRUE if the terminal can manipulate colors; otherwise, it returns FALSE.

The **can_change_color()** function returns TRUE if the terminal supports colors and can change their definitions; otherwise, it returns FALSE.

Upon successful completion, the other functions return OK; otherwise, they return ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

To use these functions, **start_color()** must be called, usually right after **initscr()**.

The **can_change_color()** and **has_colors()** functions facilitate writing terminal-independent programs. For example, a programmer can use them to decide whether to use colour or some other video attribute.

On color terminals, a typical value of COLORS is 8 and the macros such as COLOR_BLACK return a value within the range from 0 to and including 7. However, applications cannot rely on this to be true.

SEE ALSO

attroff(3X), delscreen(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

catgets( ) - get a program message

SYNOPSIS

```
#include <nl_types.h>

char *catgets(
   nl_catd catd,
   int set_num,
   int msg_num,
   const char *def_str
);
```

C

DESCRIPTION

The **catgets()** function reads message **msg_num** in set **set_num** from the message catalog identified by **catd**, a catalog descriptor returned from a previous call to **catopen()** (see *catopen*(3C)). If the call fails, **def_str** points to a default message string returned by **catgets()**.

A message longer than **NL_TEXTMAX** bytes is truncated. The returned message string is always terminated with a null byte. **NL_TEXTMAX** is defined in <**limits.h**>.

APPLICATION USAGE.

catgets() is thread-safe. It is not async-cancel-safe. A cancellation point may occur when a thread is executing **catgets()**.

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

If the call is successful, **catgets()** returns a pointer to an internal buffer area containing the null-terminated message string. If the call is unsuccessful, **catgets()** returns a pointer to **def_str**.

ERRORS

catgets() fails and sets **errno** under any of the following conditions:

[EBADF] **catd** is not a valid catalog descriptor.

[EINTR] A signal was caught during the *read*(2) system call.

[EINVAL] The message catalog identified by **catd** is corrupted.

[ENOMSG] The message identified by **set_num** or **msg_num** is not in the message catalog.

[ERANGE] A message longer than **NL_TEXTMAX** bytes was truncated.

WARNINGS

The **catgets()** function returns a pointer to a static area that is overwritten on each call.

AUTHOR

catgets() was developed by HP.

FILES

/usr/include/nl_types.h

SEE ALSO

dumpmsg(1), gencat(1), catclose(3C), catopen(3C).

STANDARDS CONFORMANCE

catgets(): AES, SVID3, XPG2, XPG3, XPG4

NAME

catopen(), catclose() - open and close a message catalog for reading

SYNOPSIS

```
#include <nl_types.h>
nl_catd catopen(const char *name, int oflag);
int catclose(nl_catd catd);
```

C

DESCRIPTION

The **catopen()** function opens a message catalog and returns a catalog descriptor. The **name** specifies the name of the message catalog being opened. A **name** containing a slash (**/**) specifies a path name for the message catalog. Otherwise, the environment variable **NLSPATH** is used (see *environ*(5)). If **NLSPATH** specifies more than one path, **catopen()** returns the catalog descriptor for the first path on which it is able to successfully open the specified message catalog. If **NLSPATH** does not exist in the environment, or if a message catalog cannot be opened for any **NLSPATH**-specified path, **catopen()** uses a system-wide default path. The default is affected by **LC_MESSAGES** if the value of **oflag** is **NL_CAT_LOCALE**. If the value of **oflag** is zero, the default is affected by the environment variable **LANG**. See *environ*(5) for details.

A message catalog descriptor remains valid in a process until the process closes it, or until a successful call to one of the **exec()** functions. A change in the setting of the **LC_MESSAGES** category may invalidate existing open catalogs.

If a file descriptor is used to implement message catalog descriptors, the **FD_CLOEXEC** flag will be set.

If **oflag** is zero, the **LANG** environment variable is used to locate the catalog. If **oflag** is **NL_CAT_LOCALE**, the **LC_MESSAGES** category is used to locate the message catalog only if a successful call to **setlocale()** has been made prior to the call to **catopen()**. The result of setting **oflag** to any other value is undefined.

The **catclose()** function closes the message catalog **catd**, a message catalog descriptor returned from an earlier successful call to **catopen()**.

APPLICATION USAGE

catopen() and **catclose()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing **catopen()** or **catclose()**.

RETURN VALUE

Upon success, **catopen()** returns a message catalog descriptor. Otherwise, **catopen()** returns a value of (*nl_catd*)−1 and sets **errno** to indicate the error.

Upon success, **catclose()** returns zero. Otherwise, **catclose()** returns –1 and sets **errno** to indicate the error.

ERRORS

catopen() fails without opening a message catalog, and sets **errno** for the last path attempted under any of the following conditions:

[EACCES]	A component of the path prefix denies search permission, or read permission is denied for the named file.
[EMFILE]	The maximum number of file descriptors allowed are currently open.
[ENAMETOOLONG]	The length of the specified path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.
[ENFILE]	The system file table is full.
[ENOENT]	The named catalog does not exist or the path is null.
[ENOTDIR]	A component of the path prefix is not a directory.

catgets() can be used to provide default messages when called following a failed **catopen()** (see *catgets*(3C)). **catgets()** returns its **def_str** parameter if it is passed an invalid catalog descriptor.

catclose() fails if the following is true:

[EBADF] **catd** is not a valid open message catalog descriptor.

WARNINGS

When using **NLSPATH**, **catopen()** does not provide a default value for **LANG**.

AUTHOR

catopen() and **catclose()** were developed by HP.

FILES

/usr/include/nl_types.h

/usr/lib/nls/msg Message catalog default path for core HP-UX products only.

SEE ALSO

catgets(3C), setlocale(3C), environ(5).

STANDARDS CONFORMANCE

catopen(): AES, SVID3, XPG2, XPG3, XPG4

catclose(): AES, SVID3, XPG2, XPG3, XPG4

NAME
cbreak, nocbreak, noraw, raw — input mode control functions

SYNOPSIS
```
#include <curses.h>
int cbreak(void);
int nocbreak(void);
int noraw(void);
int raw(void);
```

DESCRIPTION
The **cbreak()** function sets the input mode for the current terminal to *cbreak* mode and overrides a call to **raw()**.

The **nocbreak()** function sets the input mode for the current terminal to Cooked Mode without changing the state of ISIG and IXON.

The **noraw()** function sets the input mode for the current terminal to Cooked Mode and sets the ISIG and IXON flags.

The **raw()** function sets the input mode for the current terminal to Raw Mode.

RETURN VALUE
Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS
No errors are defined.

APPLICATION USAGE
If the application is not certain what the input mode of the process was at the time it called **initscr()**, it should use these functions to specify the desired input mode.

SEE ALSO
Input Mode in curses_intro(3X), <curses.h>.

X/Open System Interface Definitions, Issue 4, Version 2 specification, Chapter 9, *General Terminal Interface*.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The **raw()** and **noraw()** functions are merged with this entry. In previous issues, they appeared in entries of their own.

The entry is rewritten for clarity. The argument list for all these functions is explicitly declared as **void**.

NAME

cbrt(), cbrtf() - cube root functions

SYNOPSIS

```
#include <math.h>
double cbrt(double x);
float cbrtf(float x);
```

DESCRIPTION

cbrt() returns the cube root of *x*.

The ISO/ANSI C committee has approved the **cbrt()** function for inclusion in the C9X draft standard.

cbrtf() is a **float** version of **cbrt()**; it takes a **float** argument and returns a **float** result.

cbrtf() is not specified by any standard, but it is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

If *x* is ±INFINITY, **cbrt()** returns ±INFINITY.

If *x* is NaN, **cbrt()** returns NaN.

ERRORS

No errors are defined.

SEE ALSO

exp(3M), log(3M), pow(3M), sqrt(3M), math(5).

STANDARDS CONFORMANCE

cbrt(): SVID3, XPG4.2

NAME
ceil() - ceiling function

SYNOPSIS
```
#include <math.h>
double ceil(double x);
```

DESCRIPTION
ceil() returns the smallest integer (represented as a double-precision number) not less than *x*.

To use this function, make sure your program includes <**math.h**>, and link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE
If *x* is ±INFINITY or ±zero, **ceil()** returns *x*.

If *x* is NaN, **ceil()** returns NaN.

If the correct value would overflow, **ceil()** returns **HUGE_VAL** and sets **errno** to [ERANGE]. (This description complies with XPG4.2; however, the **ceil()** function never overflows on HP-UX systems.)

ERRORS
If **ceil()** fails, **errno** is set to the following value.

[ERANGE] The correct value would overflow.

SEE ALSO
fabs(3M), floor(3M), fmod(3M), rint(3M), math(5).

STANDARDS CONFORMANCE
ceil(): SVID3, XPG4.2, ANSI C

NAME

cfgetospeed(), cfsetospeed(), cfgetispeed(), cfsetispeed() - tty baud rate functions

SYNOPSIS

```
#include <termios.h>
speed_t cfgetospeed(const struct termios *termios_p);
int cfsetospeed(struct termios *termios_p, speed_t speed);
speed_t cfgetispeed(const struct termios *termios_p);
int cfsetispeed(struct termios *termios_p, speed_t speed);
```

C

DESCRIPTION

These functions set and get the input and output speed codes in the *termios* structure referenced by *termios_p*. The *termios* structure contains these speed codes representing input and output baud rates as well as other terminal related parameters. Setting the parameters on a terminal file does not become effective until **tcsetattr()** is successfully called.

cfgetospeed() returns the output speed code from the *termios* structure referenced by *termios_p*.

cfsetospeed() sets the output speed code in the *termios* structure referenced by *termios_p* to *speed*. The speed code for a baud rate of zero, **B0**, is used to terminate the connection. If **B0** is specified, the modem control lines are no longer asserted, which normally disconnects the line.

cfgetispeed() returns the input speed code from the *termios* structure referenced by *termios_p*.

cfsetispeed() sets the input speed code in the *termios* structure referenced by *termios_p* to *speed*.

APPLICATION USAGE

cfgetospeed(), **cfsetospeed()**, **cfgetispeed()** and **cfsetispeed()** are thread-safe and async-cancel-safe.

RETURN VALUE

cfgetospeed() returns the output speed code from the *termios* structure referenced by *termios_p*.

cfgetispeed() returns the input speed code from the *termios* structure referenced by *termios_p*.

cfsetispeed() and **cfsetospeed()** return zero upon successful completion. Otherwise, they return -1 and set **errno** to indicate the error.

ERRORS

cfsetispeed() and **cfsetospeed()** fail when the following condition is encountered:

[EINVAL] The value of *speed* is outside the range of possible speed codes as specified in <**termios.h**>.

WARNINGS

cfsetispeed() and **cfsetospeed()** can be used to set speed codes in the *termios* structure that are not supported by the terminal hardware.

SEE ALSO

tcattribute(3C), termio(7).

STANDARDS CONFORMANCE

cfgetispeed(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

cfgetospeed(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

cfsetispeed(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

cfsetospeed(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

(ENHANCED CURSES)

NAME

chgat, mvchgat, mvwchgat, wchgat — change renditions of characters in a window

SYNOPSIS

```
#include <curses.h>

int chgat(int n, attr_t attr, short color, const void *opts);

int mvchgat(int y, int x, int n, attr_t attr, short color,
            const void *opts);

int mvwchgat(WINDOW *win, int y, int x, int n, attr_t attr,
             short color, const void *opts);

int wchgat(WINDOW *win, int n, attr_t attr, short color,
           const void *opts);
```

C

DESCRIPTION

These functions change the renditions of the next *n* characters in the current or specified window (or of the remaining characters on the line, if *n* is –1), starting at the current or specified cursor position. The attributes and colors are specified by *attr* and *color* as for **setcchar()**.

These functions do not update the cursor. These functions do not perform wrapping.

A value of *n* that is greater than the remaining characters on a line is not an error.

The *opts* argument is reserved for definition in a future edition of this document. Currently, the application must provide a null pointer as *opts*.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

SEE ALSO

setcchar(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

chownacl() - change owner and/or group represented in a file's access control list (ACL) (HFS File Systems only)

SYNOPSIS

```
#include <acllib.h>

void chownacl(
   int nentries,
   const struct acl_entry *acl,
   uid_t olduid,
   gid_t oldgid,
   uid_t newuid,
   gid_t newgid
);
```

Remarks:

To ensure continued conformance with emerging industry standards, features described in this manual entry are likely to change in a future release.

DESCRIPTION

This routine alters an access control list (ACL) to reflect the change in a file's owner or group ID when an old file is copied to a new file and the ACL is also copied. **chownacl()** transfers ownership (that is, it modifies base ACL entries) in a manner similar to **chown()** (see *chown*(2)). The algorithm is described below and also in *acl*(5).

The *nentries* parameter is the current number of ACL entries in the **acl[]** array (zero or more; a negative value is treated as zero). The *olduid* and *oldgid* values are the user and group IDs of the original file's owner, typically the **st_uid** and **st_gid** values from **stat()** (see *stat*(2)). The *newuid* and *newgid* values are the user and group IDs of the new file's owner, typically the return values from **geteuid()** and **getegid()** (see *geteuid*(2) and *getegid*(2)).

If an ACL entry in **acl[]** has a *uid* of *olduid* and a *gid* of **ACL_NSGROUP** (that is, an owner base ACL entry), **chownacl()** changes *uid* to *newuid* (with exceptions – see below). If an entry has a *uid* of **ACL_NSUSER** and a *gid* of *oldgid* (that is, a group base ACL entry), **chownacl()** changes *gid* to *newgid*. In either case, only the last matching ACL entry is altered; a valid ACL can have only one of each type.

As with *chown*(2), if the new user or group already has an ACL entry (that is, a *uid* of *newuid* and a *gid* of **ACL_NSGROUP**, or a *uid* of **ACL_NSUSER** and a *gid* of *newgid*), **chownacl()** does not change the old user or group base ACL entry; both the old and new ACL entries are preserved.

As a special case, if *olduid* (*oldgid*) is equal to *newuid* (*newgid*), **chownacl()** does not search **acl[]** for an old user (group) base ACL entry to change. Calling it with both *olduid* equal to *newuid* and *oldgid* equal to *newgid* causes **chownacl()** to do nothing.

Suggested Use

This routine is useful in a program that creates a new or replacement copy of a file whose original was (or possibly was) owned by a different user or group, and that copies the old file's ACL to the new file. Copying another user's and/or group's file is equivalent to having the original file's owner and/or group copy and then transfer a file to a new owner and/or group using **chown()**. This routine is not needed for merely changing a file's ownership; **chown()** modifies the ACL appropriately in that case.

If a program also copies file miscellaneous mode bits from an old file to a new one, it must use **chmod()** (see *chmod*(2)). However, since **chmod()** deletes optional ACL entries, it must be called before **setacl()** (see *setacl*(2)). Furthermore, to avoid leaving a new file temporarily unprotected, the **chmod()** call should set only the file miscellaneous mode bits, with all access permission mode bits set to zero (that is, mask the mode with 07000). The **cpacl()** library call encapsulates this operation, and handles remote files appropriately too.

APPLICATION USAGE

chownacl() is thread-safe. It is not async-cancel-safe.

EXAMPLES

The following code fragment gets **stat()** information and the ACL from **oldfile**, transfers ownership of **newfile** to the caller, and sets the revised ACL to **newfile**.

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

int nentries;
struct acl_entry acl [NACLENTRIES];
struct stat statbuf;

if (stat ("oldfile", & statbuf) < 0)
    error (...);

if ((nentries = getacl ("oldfile", NACLENTRIES, acl)) < 0)
    error (...);

chownacl (nentries, acl, statbuf.st_uid, statbuf.st_gid,
    geteuid(), getegid());

if (setacl ("newfile", nentries, acl))
    error (...);
```

C

DEPENDENCIES

chownacl() is only supported on HFS file system on standard HP-UX operating system.

AUTHOR

chownacl() was developed by HP.

SEE ALSO

chown(2), getacl(2), getegid(2), geteuid(2), setacl(2), stat(2), acltostr(3C), cpacl(3C), setaclentry(3C), strtoacl(3C), acl(5).

(CURSES)

NAME

clear, erase, wclear, werase — clear a window

SYNOPSIS

```
#include <curses.h>
int clear(void);
int erase(void);
int wclear(WINDOW *win);
int werase(WINDOW *win);
```

DESCRIPTION

The **clear()**, **erase()**, **wclear()** and **werase()** functions clear every position in the current or specified window.

The **clear()** and **wclear()** functions also achieve the same effect as calling **clearok()**, so that the window is cleared completely on the next call to **wrefresh()** for the window and is redrawn in its entirety.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

SEE ALSO

clearok(3X), doupdate(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4

The **erase()** and **werase()** functions are merged with this entry. In previous issues, they appeared in entries of their own.

The entry is rewritten for clarity. The argument list for the **clear()** and **erase()** functions is explicitly declared as **void**.

NAME
clearenv - clear the process environment

SYNOPSIS
```
#include <stdlib.h>
int clearenv(void);
```

DESCRIPTION
clearenv() clears the process environment. No environment variables are defined immediately after a call to **clearenv()**.

clearenv() modifies the value of the pointer *environ*. This means that copies of that pointer are invalid after a call to **clearenv()**.

APPLICATION USAGE
clearenv() is thread-safe. It is not async-cancel-safe.

RETURN VALUE
Upon successful completion, **clearenv()** returns zero; otherwise, it returns –1 and sets **errno** to indicate the error.

ERRORS
clearenv() fails if the following condition is encountered:

[ENOMEM] Failed to free or reallocate memory for the process environment.

SEE ALSO
environ(5), getenv(3C), putenv(3C), <stdlib.h>.

STANDARDS CONFORMANCE
clearenv(): AES

(CURSES)

NAME

clearok, idlok, leaveok, scrollok, setscrreg, wsetscrreg — terminal output control functions

SYNOPSIS

```
#include <curses.h>
int clearok(WINDOW *win, bool bf);
int idlok(WINDOW *win, bool bf);
int leaveok(WINDOW *win, bool bf);
int scrollok(WINDOW *win, bool bf);
int setscrreg(int top, int bot);
int wsetscrreg(WINDOW *win, int top, int bot);
```

DESCRIPTION

These functions set options that deal with output within Curses.

The **clearok()** function assigns the value of *bf* to an internal flag in the specified window that governs clearing of the screen during a refresh. If, during a refresh operation on the specified window, the flag in *curscr* is TRUE or the flag in the specified window is TRUE, then the implementation clears the screen, redraws it in its entirety, and sets the flag to FALSE in *curscr* and in the specified window. The initial state is unspecified.

The **idlok()** function specifies whether the implementation may use the hardware insert-line, delete-line, and scroll features of terminals so equipped. If *bf* is TRUE, use of these features is enabled. If *bf* is FALSE, use of these features is disabled and lines are instead redrawn as required. The initial state is FALSE.

The **leaveok()** function controls the cursor position after a refresh operation. If *bf* is TRUE, refresh operations on the specified window may leave the terminal's cursor at an arbitrary position. If *bf* is FALSE, then at the end of any refresh operation, the terminal's cursor is positioned at the cursor position contained in the specified window. The initial state is FALSE.

The **scrollok()** function controls the use of scrolling. If *bf* is TRUE, then scrolling is enabled for the specified window, with the consequences discussed in *Truncation, Wrapping and Scrolling* in *curses_intro*(3X). If *bf* is FALSE, scrolling is disabled for the specified window. The initial state is FALSE.

The **setscrreg()** and **wsetscrreg()** functions define a software scrolling region in the current or specified window. The *top* and *bot* arguments are the line numbers of the first and last line defining the scrolling region. (Line 0 is the top line of the window.) If this option and **scrollok()** are enabled, an attempt to move off the last line of the margin causes all lines in the scrolling region to scroll one line in the direction of the first line. Only characters in the window are scrolled. If a software scrolling region is set and **scrollok()** is not enabled, an attempt to move off the last line of the margin does not reposition any lines in the scrolling region.

RETURN VALUE

Upon successful completion, **setscrreg()** and **wsetscrreg()** return OK. Otherwise, they return ERR.

The other functions always return OK.

ERRORS

No errors are defined.

APPLICATION USAGE

The only reason to enable the **idlok()** feature is to use scrolling to achieve the visual effect of motion of a partial window, such as for a screen editor. In other cases, the feature can be visually annoying.

The **leaveok()** option provides greater efficiency for applications that do not use the cursor.

SEE ALSO

clear(3X), delscreen(3X), doupdate(3X), scrl(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4

The **idlok()**, **leaveok()**, **scrollok()**, **setscrreq()** and **wsetscrreq()** functions are merged with this entry. In previous issues, they appeared in entries of their own.

The entry is rewritten for clarity. The *DESCRIPTION* of **clearok()** is updated to indicate that clearing of a screen applies if the flag is TRUE in either *curscr* or the specified window.

C

The *RETURN VALUE* is changed to indicate that the **clearok()**, **leaveok()** and **scrollok()** functions always return OK.

NAME
clock() - report CPU time used

SYNOPSIS
```
#include <time.h>

clock_t clock(void);
```

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()**, **system()** or **pclose()** (see *wait*(2), *system*(3S), and *popen*(3S)). To determine the time in seconds, the value returned by **clock()** should be divided by the value of the macro **CLOCKS_PER_SEC**.

The resolution of the clock varies, depending on the hardware and on software configuration.

If the processor time used is not available or its value cannot be represented, the function returns the value (**clock_t**)−1.

APPLICATION USAGE
clock() is thread-safe. It is not async-cancel-safe.

WARNINGS
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 wraps around after accumulating only 4295 seconds of CPU time (about 72 minutes).

DEPENDENCIES
The default clock resolution is 10 milliseconds.

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

STANDARDS CONFORMANCE
clock(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, ANSI C

NAME
clrtobot, wclrtobot — clear from cursor to end of window

SYNOPSIS
```
#include <curses.h>
int clrtobot(void);
int wclrtobot(WINDOW *win);
```

C

DESCRIPTION
The **clrtobot()** and **wclrtobot()** functions erase all lines following the cursor in the current or specified window, and erase the current line from the cursor to the end of the line, inclusive.

RETURN VALUE
Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS
No errors are defined.

SEE ALSO
doupdate(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The entry is rewritten for clarity. The argument list for the **clrtobot()** function is explicitly declared as **void**.

(CURSES)

NAME

clrtoeol, wclrtoeol — clear from cursor to end of line

SYNOPSIS

```
#include <curses.h>
int clrtoeol(void);
int wclrtoeol(WINDOW *win);
```

DESCRIPTION

The **clrtoeol()** and **wclrtoeol()** functions erase the current line from the cursor to the end of the line, inclusive, in the current or specified window.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

SEE ALSO

doupdate(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4

The entry is rewritten for clarity. The argument list for the **clrtoeol()** function is explicitly declared as **void**.

(ENHANCED CURSES)

NAME

COLS — number of columns on terminal screen

SYNOPSIS

```
#include <curses.h>
extern int COLS;
```

DESCRIPTION

The external variable *COLS* indicates the number of columns on the terminal screen.

SEE ALSO

initscr(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

confstr() - get string-valued configuration values

SYNOPSIS

```
#include <unistd.h>

size_t confstr(int name, char *buf, size_t len);
```

DESCRIPTION

confstr() provides a method for applications to get configuration-defined string values. Its use and purpose are similar to **sysconf()**, (see *sysconf*(2)) except that it is used where string values rather than numeric values are returned.

The *name* parameter can take on the following *name* values, which are defined in <**unistd.h**>.

_CS_PATH A default value for the **PATH** environment variable which can be used to locate commands in Section 1 of the *HP-UX Reference* and utilities defined in the POSIX.2 standard that are currently implemented in the HP-UX operating system.

_CS_HW_CPU_SUPP_BITS
Which kernel is supported on the hardware. Current values returned include "32", "32/64" or "64".

_CS_KERNEL_BITS
Whether the kernel is 32-bit or 64-bit. Current values returned include "32" or "64".

_CS_MACHINE_MODEL
The hardware model string.

_CS_XBS5_ILP32_OFF32_CFLAGS
The set of initial options to be given to the cc(1) and c89(1) utilities to build an application using a programming model with 32-bit int, long, pointer, and off_t types.

_CS_XBS5_ILP32_OFF32_LDFLAGS
The set of final options to be given to the cc(1) and c89(1) utilities to build an application using a programming model with 32-bit int, long, pointer, and off_t types.

_CS_XBS5_ILP32_OFF32_LIBS
The set of libraries to be given to the cc(1) and c89(1) utilities to build an application using a programming model with 32-bit int, long, pointer, and off_t types.

_CS_XBS5_ILP32_OFF32_LINTFLAGS
The set of options to be given to the lint(1) utility to check application source using a programming model with 32-bit int, long, pointer, and off_t types.

_CS_XBS5_ILP32_OFFBIG_CFLAGS
The set of initial options to be given to the cc(1) and c89(1) utilities to build an application using a programming model with 32-bit int, long, and pointer types, and an off_t type using at least 64-bits.

_CS_XBS5_ILP32_OFFBIG_LDFLAGS
The set of final options to be given to the cc(1) and c89(1) utilities to build an application using a programming model with 32-bit int, long, and pointer types, and an off_t type using at least 64-bits.

_CS_XBS5_ILP32_OFFBIG_LIBS
The set of libraries to be given to the cc(1) and c89(1) utilities to build an application using a programming model with 32-bit int, long, and pointer types, and an off_t type using at least 64-bits.

_CS_XBS5_ILP32_OFFBIG_LINTFLAGS
The set of options to be given to the lint(1) utility to check application source using a programming model with 32-bit int, long, and pointer types, and an off_t type using at least 64-bits.

_CS_XBS5_LP64_OFF64_CFLAGS
The set of initial options to be given to the cc(1) and c89(1) utilities to build an application using a programming model with 32-bit int, and 64-bit long, pointer, and off_t types.

_CS_XBS5_LP64_OFF64_LDFLAGS
The set of final options to be given to the cc(1) and c89(1) utilities to build an application using a programming model with 32-bit int, and 64-bit long, pointer, and off_t types.

_CS_XBS5_LP64_OFF64_LIBS
The set of libraries to be given to the cc(1) and c89(1) utilities to build an application using a programming model with 32-bit int, and 64-bit long, pointer, and off_t types.

_CS_XBS5_LP64_OFF64_LINTFLAGS
The set of options to be given to the lint(1) utility to check application source using a programming model with 32-bit int, and 64-bit long, pointer, and off_t types.

_CS_XBS5_LPBIG_OFFBIG_CFLAGS
The set of initial options to be given to the cc(1) and c89(1) utilities to build an application using a programming model with an int type using 32 bits and long, pointer, and off_t types using at least 64-bits.

_CS_XBS5_LPBIG_OFFBIG_LDFLAGS
The set of libraries to be given to the cc(1) and c89(1) utilities to build an application using a programming model with an int type using 32 bits and long, pointer, and off_t types using at least 64-bits.

_CS_XBS5_LPBIG_OFFBIG_LIBS
The set of libraries to be given to the cc(1) and c89(1) utilities to build an application using a programming model with an int type using 32 bits and long, pointer, and off_t types using at least 64-bits.

_CS_XBS5_LPBIG_OFFBIG_LINTFLAGS
The set of options to be given to the lint(1) utility to check application source using a programming model with an int type using 32 bits and long, pointer, and off_t types using at least 64-bits.

If *len* is not zero, and if *name* is known and has a configuration-defined value, **confstr()** copies that value into the *len*-byte buffer pointed to by *buf*. If the string to be returned is longer than *len* bytes, including the terminating null, **confstr()** truncates the string to *len* - 1 bytes and null-terminates the result. The application can detect that the string was truncated by comparing the value returned by **confstr()** with *len*.

If *len* is zero and *buf* is NULL, **confstr()** returns the integer value as defined below, but does not return a string. If *len* is zero but *buf* is not NULL, the result is unspecified.

Application Usage
confstr() is thread-safe. A cancellation point may occur when a thread is executing **confstr()**.

RETURN VALUE

If *name* is invalid, **confstr()** returns zero and sets **errno** to EINVAL.

If *name* does not have a configuration-defined value, **confstr()** returns 0 (zero) and leaves **errno** unchanged.

If *name* has a configuration-defined value, **confstr()** returns the size of buffer that would be needed to hold the entire configuration-defined value. If this return value is less than *len*, the string returned in *buf* has been truncated.

EXAMPLES

The following code fragment calls **confstr()** to determine the correct buffer size for **_CS_PATH**, allocates space for this buffer, then gets the configuration value for **_CS_PATH**.

```
#include <unistd.h>
#include <stddef.h>

size_t bufsize;
char *buffer;

bufsize=confstr(_CS_PATH,NULL,(size_t)0);
buffer=(char *)malloc(bufsize+1);
confstr(_CS_PATH,buffer,bufsize+1);
```

AUTHOR
confstr() was developed by HP.

FILES
/usr/include/unistd.h

SEE ALSO
getconf(1), errno(2), fpathconf(2), pathconf(2), sysconf(2), malloc(3C).

STANDARDS CONFORMANCE
confstr(): XPG4, POSIX.2

NAME

toupper(), tolower(), _toupper(), _tolower(), toascii() - translate characters

SYNOPSIS

```
#include <ctype.h>
int toupper(int c);
int tolower(int c);
int _toupper(int c);
int _tolower(int c);
int toascii(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. Arguments outside the domain cause undefined results.

The macros **_toupper()** and **_tolower()** are identical to **toupper()** and **tolower()**, respectively.

toascii() yields its argument with all bits that are not part of a standard 7-bit ASCII character cleared; it is intended for compatibility with other systems.

APPLICATION USAGE

toupper(), **tolower()**, **_toupper()**, **_tolower()** and **toascii()** are thread-safe and async-cancel-safe.

WARNING

toascii() is supplied both as a library function and as a macro defined in the <**ctype.h**> header. Normally, the macro version is used. To obtain the library function, either use a **#undef** to remove the macro definition or, if compiling in ANSI C mode, enclose the function name in parenthesis or take its address. The following examples use the library function for **toascii()**:

```
#include <ctype.h>
#undef toascii
   …
main()
{
   …
    c1 = toascii(c);
   …
}
```

or

```
#include <ctype.h>
   …
main()
{
   int (*conv_func)();
   …
   c1 = (toascii)(c);
   …
   conv_func = toascii;
   …
}
```

The following example use the library function for **toupper()**:

```
#include <ctype.h>
#undef toupper
   …
main()
```

```
{
   ...
char *c;
*c=toupper ((unsigned char*) c);
   ...
}
```

EXTERNAL INFLUENCES

Locale

The **LC_CTYPE** category determines the translations to be done.

International Code Set Support

Single-byte character code sets are supported.

AUTHOR

conv() was developed by IBM, OSF, and HP.

SEE ALSO

ctype(3C), getc(3S), setlocale(3C), lang(5).

STANDARDS CONFORMANCE

_tolower(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

_toupper(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

toascii(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

tolower(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

toupper(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

copylist() - copy a file into memory

SYNOPSIS

```
#include <libgen.h>
char *copylist(const char *filenm, off_t *szptr);
```

DESCRIPTION

copylist copies a list of items from a file into freshly allocated memory, replacing new-lines with null characters. It expects two arguments: a pointer *filenm* to the name of the file to be copied, and a pointer *szptr* to a variable where the size of the file will be stored.

Upon success, **copylist** returns a pointer to the memory allocated. Otherwise it returns **NULL** if it has trouble finding the file, calling **malloc**, or opening the file.

APPLICATION USAGE

copylist is thread-safe. It is not async-cancel-safe. A cancellation point may occur when a thread is executing copylist.

EXAMPLES

```
/* read "file" into buf */
off_t size;
char *buf;
buf = copylist("file", &size) ;"
for (i = 0; i < size; i++)
    if(buf[i])
          putchar(buf[i]);
    else
          putchar('\n');
```

SEE ALSO

malloc(3C).

NAME

copysign(), copysignf() - copysign functions

SYNOPSIS

```
#include <math.h>
double copysign(double x, double y);
float copysignf(float x, float y);
```

DESCRIPTION

The **copysign()** function returns *x* with its sign changed to the sign of *y*.

The **copysign()** function is recommended by the IEEE-754 standard for floating-point arithmetic. The ISO/ANSI C committee has approved the **copysign()** function for inclusion in the C9X draft standard.

copysignf() is a **float** version of **copysign()**; it takes **float** arguments and returns a **float** result.

copysignf() is not specified by any standard, but it is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

The **copysign()** function returns a value with the magnitude of *x* and the sign of *y*.

ERRORS

No errors are defined.

SEE ALSO

fpclassify(3M), remainder(3M), scalb(3M), scalbn(3M), signbit(3M), math(5).

NAME
copywin — copy a region of a window

SYNOPSIS
```
#include <curses.h>

int copywin(const WINDOW *srcwin, WINDOW *dstwin, int sminrow,
            int smincol, int dminrow, int dmincol, int dmaxrow,
            int dmaxcol, int overlay);
```

C

DESCRIPTION
The **copywin()** function provides a finer granularity of control over the **overlay()** and **overwrite()** functions. As in the **prefresh()** function, a rectangle is specified in the destination window, (*dminrow*, *dmincol*) and (*dmaxrow*, *dmaxcol*), and the upper-left-corner coordinates of the source window, (*sminrow*, *smincol*). If *overlay* is TRUE, then copying is non-destructive, as in **overlay()**. If *overlay* is FALSE, then copying is destructive, as in **overwrite()**.

RETURN VALUE
Upon successful completion, **copywin()** returns OK. Otherwise, it returns ERR.

ERRORS
No errors are defined.

SEE ALSO
newpad(3X), overlay(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 4.

NAME

cos(), cosf() - cosine functions

SYNOPSIS

```
#include <math.h>
double cos(double x);
float cosf(float x);
```

DESCRIPTION

cos() returns the cosine of *x* (*x* specified in radians).

cos() may lose accuracy when *x* is far from zero.

cosf() is a **float** version of **cos()**; it takes a **float** argument and returns a **float** result. To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options.

cosf() is not specified by any standard, but it is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, make sure your program includes <**math.h**>, and link in the math library by specifying **-lm** on the compiler or linker command line.

Millicode versions of the **cos()** and **cosf()** functions are available. Millicode versions of math library functions are usually faster than their counterparts in the standard library. To use these versions, compile your program with the **+Olibcalls** or the **+Oaggressive** optimization option.

For special cases, the millicode versions return the same values as their standard library counterparts (see the *RETURN VALUE* section).

For more information, see the *HP-UX Floating-Point Guide*.

RETURN VALUE

If *x* is NaN or ±INFINITY, **cos()** returns NaN.

If the correct value after rounding would be smaller in magnitude than **MINDOUBLE**, **cos()** returns zero.

ERRORS

No errors are defined.

SEE ALSO

acos(3M), asin(3M), atan(3M), atan2(3M), cosd(3M), sin(3M), tan(3M), math(5), values(5).

STANDARDS CONFORMANCE

cos(): SVID3, XPG4.2, ANSI C

NAME

cosd(), cosdf() - degree-valued cosine functions

SYNOPSIS

```
#include <math.h>
double cosd(double x);
float cosdf(float x);
```

C

DESCRIPTION

The **cosd()** function is a degree-valued version of **cos()**. It returns the cosine of *x* (*x* specified in degrees).

The **cosd()** function may lose accuracy when *x* is far from zero.

The **cosdf()** function is a **float** version of **cosd()**; it takes a **float** argument and returns a **float** result.

The **cosd()** and **cosdf()** functions are not specified by any standard, but **cosdf()** is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

If *x* is NaN or ±INFINITY, **cosd()** returns NaN.

If the correct value after rounding would be smaller in magnitude than **MINDOUBLE**, **cosd()** returns zero.

ERRORS

No errors are defined.

SEE ALSO

acosd(3M), asind(3M), atan2d(3M), atand(3M), cos(3M), sind(3M), tand(3M), math(5), values(5).

NAME

cosh(), coshf() - hyperbolic cosine functions

SYNOPSIS

```
#include <math.h>
double cosh(double x);
float coshf(float x);
```

DESCRIPTION

The **cosh()** function returns the hyperbolic cosine of its argument.

The **coshf()** function is a **float** version of **cosh()**; it takes a **float** argument and returns a **float** result. To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options.

The **coshf()** function is not specified by any standard, but it is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, make sure your program includes <**math.h**>, and link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

If *x* is ±INFINITY, **cosh()** returns +INFINITY.

If *x* is NaN, **cosh()** returns NaN.

If the correct value would overflow, **cosh()** returns **HUGE_VAL** and sets **errno** to [ERANGE].

ERRORS

If **cosh()** fails, **errno** is set to the following value.

[ERANGE] The correct value would overflow.

SEE ALSO

acosh(3M), sinh(3M), tanh(3M), math(5).

STANDARDS CONFORMANCE

cosh(): SVID3, XPG4.2, ANSI C

NAME

cpacl(), fcpacl() - copy the access control list (ACL) and mode bits from one file to another (HFS File Systems only)

SYNOPSIS

```
#include <acllib.h>

int cpacl(
   const char *fromfile,
   const char *tofile,
   mode_t frommode,
   uid_t fromuid,
   gid_t fromgid,
   uid_t touid,
   gid_t togid
);

int fcpacl(
   int fromfd,
   int tofd,
   mode_t frommode,
   uid_t fromuid,
   gid_t fromgid,
   uid_t touid,
   gid_t togid
);
```

Remarks:

To ensure continued conformance with emerging industry standards, features described in this manual entry are likely to change in a future release.

DESCRIPTION

Both **cpacl()** and **fcpacl()** copy the access control list and mode bits (that is, file access permission bits and miscellaneous mode bits; see *chmod*(2)) from one file to another, and transfer ownership much like *chown*(2). **cpacl()** and **fcpacl()** take the following parameters:

- Path names (*fromfile* and *tofile*) or open file descriptors (*fromfd* and *tofd*).
- A mode value (*frommode*, typically the **st_mode** value returned by **stat()** – see *stat*(2)) containing file miscellaneous mode bits which are always copied, and file access permission bits which are copied instead of the access control list if either file is remote.
- User ID and group ID of the file (*fromuid*, *touid* and *fromgid*, *togid*) for transferring ownership. (Typically *fromuid* and *fromgid* are the **st_uid** and **st_gid** values returned by **stat()**, and *touid* and *togid* are the return values from **geteuid()** and **getegid()** – see *geteuid*(2) and *getegid*(2).)

When both files are local, the **cpacl()** routines copy the access control list and call **chownacl()** (see *chownacl*(3C)) to transfer ownership from the *fromfile* to the *tofile*, if necessary.

cpacl() (**fcpacl()**) handles remote copying (via NFS) after recognizing failures of **getacl()** (**fgetacl()**) or **setacl()** (**fsetacl()**) (see *setacl*(2)). When copying the mode from *fromfile* (*fromfd*) to *tofile* (*tofd*), **cpacl()** copies the entire *frommode* (that is, the file miscellaneous mode bits and the file access permission bits) to *tofile* (*tofd*) using **chmod()** (**fchmod()**). Some of the miscellaneous mode bits can be turned off; see *chmod*(2).

cpacl() (**fcpacl()**) can copy an access control list from *fromfile* (*fromfd*) to *tofile* (*tofd*) without transferring ownership, but ensuring error checking and handling of remote files. This is done by passing *fromuid* equal to *touid* and *fromgid* equal to *togid* (that is, four zeros). For remote files, *fromuid*, *touid*, *fromgid*, and *togid* are ignored.

APPLICATION USAGE

cpacl() and **fcpacl()** are thread-safe. These interfaces are not async-cancel-safe.

RETURN VALUE

If successful, **cpacl()** and **fcpacl()** return zero. If an error occurs, they set **errno** to indicate the cause of failure and return a negative value, as follows:

–1 Unable to perform **getacl()** (**fgetacl()**) on a local *fromfile* (*fromfd*).

–2 Unable to perform **chmod()** (**fchmod()**) on *tofile* (*tofd*) to set its file miscellaneous mode bits. **cpacl()** (**fcpacl()**) attempts this regardless of whether a file is local or remote, as long as *fromfile* (*fromfd*) is local.

–3 Unable to perform **setacl()** (**fsetacl()**) on a local *tofile* (*tofd*). As a consequence, the file's optional ACL entries are deleted, its file access permission bits are zeroed, and its miscellaneous mode bits might be altered.

–4 Unable to perform **chmod()** (**fchmod()**) on *tofile* (*tofd*) to set its mode. As a consequence, if *fromfile* (*fromfd*) is local, *tofile*'s (*tofd*'s) optional ACL entries are deleted, its access permission bits are zeroed, and its file miscellaneous mode bits might be altered, regardless of whether the file is local or remote.

EXAMPLES

The following code fragment gets *stat* information on **oldfile** and copies its file miscellaneous bits and access control list to **newfile** owned by the caller. If either file is remote, only the **st_mode** on **oldfile** is copied.

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

struct stat statbuf;

if (stat ("oldfile", & statbuf) < 0)
error (...);

if (cpacl ("oldfile", newfile , statbuf.st_mode,
statbuf.st_uid, statbuf.st_gid, geteuid(), getegid()) < 0)
{
error (...);
}
```

DEPENDENCIES

cpacl() and **fcpacl()** are only supported on HFS file system on standard HP-UX operating system.

AUTHOR

cpacl() and **fcpacl()** were developed by HP.

SEE ALSO

chown(2), getacl(2), getegid(2), geteuid(2), setacl(2), stat(2). acltostr(3C), chownacl(3C), strtoacl(3C), acl(5).

NAME

cr_close - close a crash dump descriptor

SYNOPSIS

```
#include <libcrash.h>
int cr_close(CRASH *crash_cb);
```

DESCRIPTION

The **cr_close()** function closes the crash dump descriptor structure pointed to by *crash_cb*. The **crash_cb** structure is a crash dump descriptor filled in by a *cr_open*(3) call. All associated resources (memory and files) are released.

RETURN VALUE

Returns zero for success. Other possible return values are described in *libcrash*(5).

AUTHOR

cr_close() was developed by HP.

SEE ALSO

cr_open(3), libcrash(5).

NAME

cr_info - retrieve crash dump information

SYNOPSIS

```
#include <libcrash.h>

cr_info_t *cr_info(CRASH *crash_cb);
```

DESCRIPTION

The **cr_info()** function returns a pointer to a **cr_info_t** structure with information about an open crash dump.

cr_info_t

The **cr_info_t** structure contains the following fields. Note that there is no necessary correlation between the placement in this list and the order in the structure, and the structure may contain other, reserved fields.

```
char             *cri_hostname    Name of system dumped
char             *cri_model       Model string from system
char             *cri_panic       Panic message
char             *cri_release     Release string from system
char             *cri_dumptime    Time of the crash dump
char             *cri_savetime    Time crash dump saved
uint64_t          cri_chunksize   Size of uncompressed image files.
uint64_t          cri_memsize     Physical size of memory (bytes)
uint64_t          cri_num_err     Number of error messages
char            **cri_errmsg      Error messages from INDEX file
uint64_t          cri_num_warn    Number of warning messages
char            **cri_warnmsg     Warning messages from INDEX file
uint64_t          cri_num_mod     Number of module structures
cri_modinfo_t    *cri_modinfo     Array of module structures
```

The fields *cri_hostname*, *cri_model*, *cri_panic*, *cri_release*, *cri_dumptime*, and *cri_savetime* are all null-terminated strings; respectively, representing the name and model of the system that dumped, the panic message, the release version string of the kernel, and the times that the system dumped and the dump was saved. The amount, in bytes, of physical memory on the system is in *cri_memsize*. The target size, in bytes, of individual uncompressed image files is in *cri_chunksize*. *cri_errmsg* and *cri_warnmsg* are arrays of null-terminated strings with the list of saved messages. *cri_num_err* and *cri_num_warn* are the number of strings in each list. *cri_modinfo* is an array of **cri_modinfo_t** structures defining the loaded kernel modules; there are *cri_num_mod* elements in the array.

cri_modinfo_t

The **cri_modinfo_t** structure contains information specific to kernel modules loaded into memory at the time of the system crash. The specific fields are:

```
char       *mdi_loadpath   File path when loaded into memory
char       *mdi_savepath   File path when saved in crash directory
uint64_t    mdi_size       Size in bytes
uint64_t    mdi_checksum   Checksum of the file from cksum(1)
```

Again, the order of these fields within the structure is not specified, and other reserved fields may be present in the structure.

The caller must not modify any information in any of the structures returned by **cr_info()**.

RETURN VALUE

cr_info() returns a pointer to the **cr_info_t** structure described above. If **crash_cb** is not a valid descriptor of a crash dump opened with **cr_open()**, the return value is undefined.

EXAMPLES

Assuming a process opened a crash dump, the following call to *cr_info*(3) retrieves the information about the open crash dump.

```
#include <libcrash.h>

CRASH *crshdes;
```

```
cr_info_t *cri;

cri = cr_info(crshdes);
```

AUTHOR

cr_info() was developed by HP.

SEE ALSO

cr_open(3), libcrash(5).

C

NAME

cr_isaddr - validate whether physical page number was dumped

SYNOPSIS

```
#include <libcrash.h>

int cr_isaddr(CRASH *crash_cb, uint64_t pagenum, int *avail);
```

DESCRIPTION

The **cr_isaddr()** checks to see if the specified page number, *pagenum*, is present in the open crash dump represented by *crash_cb*. It sets the Boolean to which *avail* points to indicate the presence (1) or absence (0) of the page.

RETURN VALUE

Returns zero for success. Other possible return values are described in *libcrash*(5).

EXAMPLES

Assuming a process opened a crash dump, the following call to *cr_isaddr*(3) tests for the availability of page 256:

```
#include <libcrash.h>

CRASH cr_cb;
int avail;

int retval = cr_isaddr(&cr_cb, (uint64_t)256, &avail);

if (retval >= 0 && avail > 0) {
      /* Read and process page 256 */
} /* Else ignore pages that are not available */
```

AUTHOR

cr_isaddr() was developed by HP.

SEE ALSO

cr_open(3), cr_read(3), libcrash(5).

NAME

cr_open() - open crash dump for reading

SYNOPSIS

```
#include <libcrash.h>

int cr_open(const char *path, CRASH **crash_cb, int flags);
```

DESCRIPTION

The **cr_open()** library call opens a crash dump and passes back a crash dump descriptor.

The *path* argument points to a path name naming a crash dump directory or file, and must not exceed **PATH_MAX** bytes in length.

The **CRASH *** to which *crash_cb* points is set to a crash dump descriptor, which can then be passed to the other **cr_*()** functions to access the crash dump.

flags is a bitmask of zero or more of the following flag values, which affect the operation of future calls to **libcrash** routines for this crash dump, except for *cr_verify*(3), which has its own *flags* parameter.

CR_NOCHECKSUM The library will not attempt to verify checksums of files in the crash dump if this flag is set.

CR_DELAYMSGS The library will write messages to stderr during time-consuming operations (decompressions and checksums) if this flag is set.

RETURN VALUE

Returns zero for success. Other possible return values are described in *libcrash*(5).

EXAMPLES

The following call to **cr_open()** opens crash dump contained in the directory **/var/adm/crash/core.0** and returns the crash dump descriptor **crash_cb**. For an example of reading the crash dump **/var/adm/crash/core.0**, see the *cr_read*(3) manual entry.

```
#include <libcrash.h>

CRASH *crash_cb;
int    ret;

ret = cr_open ("/var/adm/crash/core.0", &crash_cb, CR_DELAYMSGS);
```

AUTHOR

cr_open() was developed by HP.

SEE ALSO

cr_close(3), cr_perror(3), libcrash(5).

NAME

cr_perror - print a libcrash error or warning message

SYNOPSIS

```
#include <libcrash.h>

void cr_perror(CRASH *crash_cb, int error);
```

DESCRIPTION

cr_perror() prints to standard error an error or warning message corresponding to *error*, which should be the return value from an immediately previous call to one of the **libcrash** calls. The message describes the problem that occurred, explains its implications when appropriate, and gives corrective action where appropriate.

If called with a zero *error* value, indicating success, **cr_perror()** prints nothing. Therefore, it can be called after each **libcrash** call without harm. (If called after **cr_verify()** with the **CR_ERRORMSGS** flag set, repeated error messages may result.)

crash_cb should be the same crash dump descriptor as was passed to the call that returned *error*. If the call was **cr_open()**, pass the crash dump descriptor returned by that call, even if it is NULL.

If *error* is **CRERR_ERRNO**, indicating that the system variable **errno** contains the error information, **cr_perror()** will refer to **errno** to print the appropriate message. In these cases it is critical that **cr_perror()** be called immediately after the offending **libcrash** call returns.

AUTHOR

cr_perror () was developed by HP.

SEE ALSO

cr_open(3), cr_verify(3), libcrash(5).

NAME

cr_read - read from crash dump

SYNOPSIS

```
#include <libcrash.h>

int cr_read(CRASH *crash_cb, void *buf,
            uint64_t mem_page, int *num_pages);
```

DESCRIPTION

The **cr_read()** function attempts to read the memory area defined by *mem_page* and *num_pages* into the buffer pointed to by *buf* from the crash dump opened using *crash_cb*.

The **cr_read()** starts at the position in the crash dump associated with the physical memory offset given by *mem_page*. If the physical memory page *mem_page* does not exist in the crash dump, **cr_read()** sets **num_pages* to 0 and returns 0.

No data transfer will occur past a page of memory that does not exist in the crash dump. If the starting position, *mem_page*, plus the read length, **num_pages*, goes past an area of memory that does not exist in the crash dump, **cr_read()** sets **num_pages* to the number of consecutive pages (starting at *mem_page*) actually read.

RETURN VALUE

Returns zero for success. Other possible return values are described in *libcrash*(5).

EXAMPLES

Assuming a process opened a crash dump, the following call to *cr_read*(3) reads the first **CRSIZE** pages from the crash dump into the buffer pointed to by *mybuf*:

```
#include <libcrash.h>

CRASH in_cb;
char mybuf[CRSIZE*NBPG];
int rstat;
int size = CRSIZE;

rstat = cr_read (in_cb, mybuf, 0, &size);
```

WARNING

cr_read() may return fewer pages than requested due to implementation details. Always check the number of pages returned. If they are fewer than requested, issue a new request starting at the first page not returned. Only if that new request reads zero pages (or returns an error) can you be sure that the page was not dumped.

AUTHOR

cr_read() was developed by HP.

SEE ALSO

cr_open(3), libcrash(5).

NAME

cr_uncompress - uncompress a file in a crash dump

SYNOPSIS

```
#include <libcrash.h>

int cr_uncompress(CRASH *crash_cb, const char *pathname,
                  uint64_t size, uint64_t checksum);
```

DESCRIPTION

The **cr_uncompress()** ensures that a file, part of a crash dump described by *crash_cb*, is uncompressed and matches its expected size and checksum (as computed by *cksum*(1)). This call is most often used to ensure the integrity of module files that are a part of the crash dump; see *cr_info*(3).

pathname is the name of the file to uncompress. Supported compression methods include *gzip*(1), which appends a **.gz** to the filename, and *compress*(1), which appends **.Z**. Respectively, *size* and *checksum* are the expected size and checksum of the uncompressed file. Either validity check can be disabled by specifying zero for the corresponding parameter.

RETURN VALUE

Returns zero for success. Other possible return values are described in *libcrash*(5).

EXAMPLE

The following call to *cr_uncompress*(3) ensures that the kernel file **vmunix** is uncompressed and validated.

```
#include <libcrash.h>

result = cr_uncompress (crash, "vmunix", vmunix_size, vmunix_cksum);
```

AUTHOR

cr_uncompress() was developed by HP.

SEE ALSO

gunzip(1), uncompress(1), cr_info(3), cr_open(3), libcrash(5).

NAME

cr_verify - verify integrity of crash dump

SYNOPSIS

```
#include <libcrash.h>

int cr_verify(CRASH *crash_cb, int flags);
```

DESCRIPTION

cr_verify() uncompresses and verifies the sizes and checksums of every file in the crash dump identified by *crash_cb*.

flags is a bitmask of zero or more of the following flag values:

CR_NOCHECKSUM **cr_verify()** will not attempt to verify checksums of files in the crash dump if this flag is set. Only sizes will be verified.

CR_DELAYMSGS **cr_verify()** will write messages to stderr during time-consuming operations (decompressions and checksums) if this flag is set.

CR_ERRORMSGS **cr_verify()** will write messages to stderr describing any validation problems that are encountered. If this flag is set, **cr_perror()** should not be called when **cr_verify()** returns; repeated error messages would result.

RETURN VALUE

Returns zero for success. Other possible return values are described in *libcrash*(5).

EXAMPLES

The following call to *cr_verify*(3) verifies the integrity of the dump.

```
#include <libcrash.h>

CRASH *crash_cb;
int ret;

ret = cr_verify(crash_cb, CR_DELAYMSGS | CR_ERRORMSGS);
```

WARNING

Because it uncompresses and checksums all files in a dump, **cr_verify()** can be very time-consuming. If **CR_DELAYMSGS** was not specified, the calling application should notify its user before calling **cr_verify()** that there may be a significant delay.

AUTHOR

cr_verify() was developed by HP.

SEE ALSO

cr_open(3), cr_perror(3), libcrash(5).

NAME

crt0.o, gcrt0.o, mcrt0.o - execution startup routines (PA64 ELF uses crt0.o only)

DESCRIPTION

PA32 SOM

The C, Pascal, and FORTRAN compilers link in the object files **crt0.o**, **gcrt0.o**, or **mcrt0.o** to provide startup capabilities and environments for program execution. All are identical except that **gcrt0.o** and **mcrt0.o** provide additional functionality for *gprof*(1) and *prof*(1) profiling support respectively.

The following symbols are defined in these object files:

_environ An array of character pointers to the environment in which the program will run. This array is terminated by a null pointer.

_FPU_MODEL A variable of type *short* containing the FPU model number returned by the FP status instruction. This variable is initialized with data from the kernel.

_FPU_REVISION A variable of type *short* containing the FPU revision number returned by the FP status instruction. This variable is initialized with data from the kernel.

_CPU_KEYBITS_1
A variable of type *int* containing CPU specific information. This variable is initialized with data from the kernel.

_CPU_REVISION A variable of type *int* containing the CPU revision of the machine. This variable is initialized with data from the kernel.

_SYSTEM_ID A variable of type *int* containing the system id value for an executable program.

$START$ Execution start address.

_start A secondary startup routine for C programs, called from **$START$**, which in turn calls **main**. This routine is contained in the C library rather than the **crt0.o** file. For Pascal and FORTRAN programs, this symbol labels the beginning of the outer block (main program) and is generated by the compilers.

$global$ The initial address of the program's data pointer. The startup code loads this address into general register 27.

$UNWIND_START The beginning of the stack unwind table.

$UNWIND_END The end of the stack unwind table.

$RECOVER_START
The beginning of the try/recover table.

$RECOVER_END The end of the try/recover table.

The **crt0.o** file defines a null procedure for **_mcount**, so programs compiled with profiling can be linked without profiling.

The linker defines the following two symbols:

__text_start The beginning address of the program's text area.
__data_start The beginning address of the program's data area.

PA64 ELF

The C, Pascal, and FORTRAN compilers link in the object file **crt0.o** to provide startup capabilities and environments for program execution. It contains startup code that must be linked using *ld*(1) to every **-noshared** PA64 program. In a **-dynamic** program, the **crt0.o** object file is not used, and all actions normally associated with it are instead done by the dynamic loader *dld*(1). Additional functionality for *prof*(1) and *gprof*(1) profiling support is no longer handled by **crt0.o**.

In PA64, **crt0.o** processes *initializers* and *terminators*. *Initializers* are routines that are called before the program entry point and *terminators* are routines that are called when the program terminates via the **exit** routine. *Initializers* are invoked in reverse order of the link line so that dependent libraries are initialized before the libraries that depend on them. *Terminators*, on the other hand, are invoked in the forward order.

Unlike the SOM version of **crt0.o**, **crt0.o** for PA64 ELF does not define any variables. It, however, sets the following global variables:

__argc A variable of type *long* containing the number of arguments.

__argv An array of character pointers to the arguments themselves.

_environ, __envp
An array of character pointers to the environment in which the program will run. This array is terminated by a null pointer.

_CPU_KEYBITS_1
A variable of type *int* containing CPU specific information. This variable is initialized with data from the kernel.

_FPU_MODEL A variable of type *long* containing the FPU model number returned by the FP status instruction. This variable is initialized with data from the kernel.

_FPU_REVISION A variable of type *long* containing the FPU revision number returned by the FP status instruction. This variable is initialized with data from the kernel.

_CPU_REVISION A variable of type *long* containing the CPU revision of the machine. This variable is initialized with data from the kernel.

_SYSTEM_ID A variable of type *long* containing the system id value for an executable program.

__tls_size A variable of type *long* containing the requested thread local storage size. This variable is initialized with data from the kernel.

__load_info A variable of type *void* * containing load information passed from the kernel.

AUTHOR

The features described in this entry originated from AT&T UNIX System III.

FILES

crt0.h

SEE ALSO

Profiling and Debugging Tools:

gprof(1)	display call graph profile data
monitor(3C)	prepare execution profile
prof(1)	display profile data
profil(2)	execution time profile

System Tools:

cc(1)	invoke the HP-UX C compiler
exec(2)	execute a file
f77(1)	invoke the HP-UX FORTRAN compiler
ld(1)	invoke the link editor
dld(1)	the dynamic loader
pc(1)	invoke the HP-UX Pascal compiler

Miscellaneous:

end(3C)	symbol of the last locations in program

NAME

crypt, setkey, encrypt - generate hashing encryption

SYNOPSIS

```
#include <crypt.h>
#include <unistd.h>

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

void setkey(const char *key);

void encrypt(char block[64], int edflag);
```

Obsolescent Interfaces

```
char *crypt_r(const char *key, const char *salt, CRYPTD *cd);

void setkey_r(const char *key, CRYPTD *cd);

void encrypt_r(char block[64], int edflag, CRYPTD *cd);
```

DESCRIPTION

crypt():

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

key is a user's typed password. *salt* is a two-character string chosen from the set **[a-zA-Z0-9./]**; this string is used to perturb the hashing algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password. The first two characters are the salt itself.

setkey() and encrypt():

setkey() and **encrypt()** provide (rather primitive) access to the actual hashing algorithm. The argument to **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 is used with the hashing algorithm to encrypt or decrypt the string *block* with the function **encrypt()**.

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

Obsolescent Interfaces

crypt_r(), **setkey_r()**, **encrypt_r()** generate hashing encryption.

APPLICATION USAGE

crypt(), **encrypt()** and **setkey()** are thread-safe. These interfaces are not async-cancel-safe.

WARNINGS

The return value for **crypt()** points to data whose content is overwritten by each call.

crypt_r(), **setkey_r()**, and **encrypt_r()** are obsolescent interfaces supported only for compatibility with existing DCE applications. New multithreaded applications should use **crypt()**, **encrypt()** and **setkey().**

SEE ALSO

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

STANDARDS CONFORMANCE

crypt(): SVID2, SVID3, XPG2, XPG3, XPG4

encrypt(): SVID2, SVID3, XPG2, XPG3, XPG4

setkey(): SVID2, SVID3, XPG2, XPG3, XPG4

NAME

ctermid() - generate file name for terminal

SYNOPSIS

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

DESCRIPTION

ctermid() generates a string that, when used as a pathname, refers to the the controlling terminal for the current process.

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.

If the process has no controlling terminal, the pathname for the controlling terminal cannot be determined, or some other error occurs, **ctermid()** returns an empty string.

For multi-thread applications, if *s* is a NULL pointer, the operation is not performed and a NULL pointer is returned.

APPLICATION USAGE

ctermid() is thread-safe. It is not async-cancel-safe.

NOTES

The difference between **ctermid()** and **ttyname()** 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 refers to the terminal if used as a file name. (see *ttyname*(3C)). Thus **ttyname()** is useful only if the process already has at least one file open to a terminal.

SEE ALSO

ttyname(3C).

STANDARDS CONFORMANCE

ctermid(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

ctime(), ctime_r(), localtime(), localtime_r(), gmtime(), gmtime_r(), mktime(), difftime(), asctime(), asctime_r(), timezone(), daylight(), tzname(), tzset() - convert date and time to string

SYNOPSIS

```
#include <time.h>
char *asctime(const struct tm *timeptr);
char *asctime_r(const struct tm *timeptr, char *buffer);
char *ctime(const time_t *timer);
char *ctime_r(const time_t *timer, char *buffer);
double difftime(time_t time1, time_t time0);
struct tm *gmtime(const time_t *timer);
struct tm *gmtime_r(const time_t *timer, struct tm *result);
struct tm *localtime(const time_t *timer);
struct tm *localtime_r(const time_t *timer, struct tm *result);
time_t mktime(struct tm *timeptr);
extern long timezone;
extern int daylight;
extern char *tzname[2];
void tzset(void);
```

DESCRIPTION

asctime() Convert the broken-down time contained in the structure pointed to by *timeptr* and return a pointer to a 26-character string in the form:

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

All the fields have constant width.

asctime() returns NULL and sets errno to **ERANGE** if *tm_year* in **timeptr** is less than 0 or is greater than 8099. In both 32-bit and 64-bit HP-UX, the minimum date supported by **asctime()** is January 1 00:00:00 1900 and the maximum date supported by **asctime()** is December 31 23:59:59 9999.

asctime_r() is identical to **asctime()**, except that it places the result in the user supplied **buffer** and returns a pointer to **buffer** upon success. A buffer length of at least 26 is required.

ctime() Convert the calendar time pointed to by *timer*, representing the time in seconds since the Epoch, and return a pointer to the local time in the form of a string. Equivalent to:

```
asctime(localtime(timer))
```

The minimum date supported by **ctime()** in both 32-bit and 64-bit HP-UX is Friday December 13 20:45:52 UTC 1901. The maximum dates supported by **ctime()** are Tuesday January 19 03:14:07 UTC 2038 and Friday December 31 23:59:59 UTC 9999 in 32-bit HP-UX and 64-bit HP-UX, respectively.

In 64-bit HP-UX, **ctime()** returns NULL and sets errno to **ERANGE** if timer is less than the number of seconds that corresponds to the minimum date supported (i.e., **INT_MIN,** as defined in **limits.h**), or exceeds the number of seconds that corresponds to the maximum date supported.

ctime_r() is identical to **ctime()**, except that it places the result in the user supplied **buffer** and returns a pointer to **buffer** upon success. A buffer length of at least 26 is required.

gmtime() Convert directly to Coordinated Universal Time (UTC), the time standard used by the HP-UX operating system. **gmtime()** returns a pointer to the **tm** structure described below.

The minimum date supported by **gmtime()** in both 32-bit and 64-bit HP-UX is Friday December 13 20:45:52 UTC 1901. The maximum dates supported by **gmtime()** are Tuesday January 19 03:14:07 UTC 2038 and Friday December 31 23:59:59 UTC 9999 in 32-bit HP-UX and 64-bit HP-UX, respectively.

In 64-bit HP-UX, **gmtime()** returns NULL and sets errno to **ERANGE** if timer is less than the number of seconds that corresponds to the minimum date supported (i.e., **INT_MIN,** as defined in **limits.h**), or exceeds the number of seconds that corresponds to the maximum date supported.

C

gmtime_r() is identical to **gmtime()**, except that **gmtime_r()** stores the result in the **tm struct** pointed to by **result** and returns **result** upon success.

localtime() Correct for the time zone and any summer time zone adjustments (such as Daylight Savings Time in the USA), according to the contents of the TZ environment variable (see Environment Variables below). **localtime()** returns a pointer to the **tm** structure described below.

The minimum date supported by **localtime()** in both 32-bit and 64-bit HP-UX is Friday December 13 20:45:52 UTC 1901. The maximum dates supported by **localtime()** are Tuesday January 19 03:14:07 UTC 2038 and Friday December 31 23:59:59 UTC 9999 in 32-bit HP-UX and 64-bit HP-UX, respectively.

In 64-bit HP-UX, **localtime()** returns NULL and sets errno to **ERANGE** if timer is less than the number of seconds that corresponds to the minimum date supported (i.e., **INT_MIN,** as defined in **limits.h**), or exceeds the number of seconds that corresponds to the maximum date supported.

localtime_r()
is identical to **localtime()**, except that **localtime_r()** stores the result in the **tm struct** pointed to by **result** and returns **result** upon success.

difftime() Return the difference in seconds between two calendar times: *time1 – time0*.

mktime() Convert the broken-down time (expressed as local time) in the structure pointed to by *timeptr* into a calendar time value with the same encoding as that of the values returned by *time*(2). The original values of the **tm_wday** and **tm_yday** components of the structure are ignored, and the original values of the other components are not restricted to the ranges indicated below.

A positive or zero value for **tm_isdst** causes **mktime()** to initially presume that Daylight Saving Time respectively is or is not in effect for the specified time. A negative value for **tm_isdst** causes **mktime()** to attempt to determine whether Daylight Saving Time is in effect for the specified time.

Upon successful completion, all the components are set to represent the specified calendar time, but with their values forced to the ranges indicated below. The final value of **tm_mday** is not set until **tm_mon** and **tm_year** are determined. **mktime()** returns the specified calendar time encoded as a value of type **time_t**.

If the calendar time cannot be represented, the function returns the value (**time_t**)—1 and sets **errno** to **ERANGE**. Note the value (**time_t**)—1 also corresponds to the time 23:59:59 on Dec 31, 1969 (plus or minus time zone and Daylight Saving Time adjustments). Thus it is necessary to check both the return value and **errno** to reliably detect an error condition.

tzset() Sets the values of the external variables *timezone*, *daylight*, and *tzname* according to the contents of the **TZ** environment variable (independent of any time value). The functions **localtime()**, **mktime()**, **ctime()**, **asctime()**, and **strftime()** (see *strftime*(3C)) call **tzset()** and use the values returned in the external variables described below for their operations. **tzset()** can also be called directly by the user.

The <**time.h**> header file contains declarations of all relevant functions and externals. It also contains the **tm** structure, which includes the following members:

```
int tm_sec;      /* seconds after the minute - [0,61] */
int tm_min;      /* minutes after the hour - [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;     /* years since 1900 */
int tm_wday;     /* days since Sunday - [0,6] */
int tm_yday;     /* days since January 1 - [0,365] */
int tm_isdst;    /* daylight savings time flag */
```

The value of **tm_isdst** is positive if a summer time zone adjustment such as Daylight Savings Time is in effect, zero if not in effect, and negative if the information is not available.

The external variable **timezone** contains the difference, in seconds, between UTC and local standard time (for example, in the U.S. Eastern time zone (EST), **timezone** is 5*60*60). The external variable **daylight** is non-zero only if a summer time zone adjustment is specified in the TZ environment variable. The external variable **tzname[2]** contains the local standard and local summer time zone abbreviations as specified by the **TZ** environment variable.

C

EXTERNAL INFLUENCES

Locale

The **LC_CTYPE** category determines the interpretation of the bytes within *format* as single and/or multi-byte characters.

Environment Variables

The **tzset()** function uses the contents of **TZ** to set the values of the external variables **timezone**, **daylight** , and **tzname**. **TZ** also determines the time zone name substituted for the **%Z** and **%z** directives and the time zone adjustments performed by **localtime()**, **mktime()**, and **ctime()**. Two methods for specifying a time zone within **TZ** are described in *environ*(5).

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

For **asctime_r()** and **ctime_r(),** if the buffer is of insufficient length, a NULL is returned and errno set to EINVAL.

asctime_r(), ctime_r(), gmtime_r(), and **localtime_r(),** return a NULL and set errno to EINVAL if NULL pointers are passed in as arguments.

A NULL is returned and errno is set to **ERANGE** if the input to the following routines is not within the supported range: **asctime()**, **asctime_r()**, **ctime()**, **ctime_r()**, **gmtime()**, **gmtime_r()**, **localtime()**, **localtime_r()**.

APPLICATION USAGE

The return values for **asctime()**, **ctime()**, **gmtime()**, and **localtime()** point to static data whose contents is overwritten by each call. Thus, these routines are unsafe in multi-thread applications. **asctime_r()**, **ctime_r()**, **gmtime_r()**, and **localtime_r()** are thread-safe. These interfaces are not async-cancel-safe.

WARNINGS

Users of **asctime_r()**, **ctime_r()**, **gmtime_r()**, and **localtime_r()** should also note that these functions now conform to POSIX.1c. The old prototypes of these functions are supported for compatibility with existing DCE applications only.

The range of **tm_sec([0,61])** extends to 61 to allow for the occasional one or two leap seconds. However, the "seconds since the Epoch" value returned by *time*(2) and passed as the *timer* argument does not include accumulated leap seconds. The **tm** structure generated by **localtime()** and **gmtime()** will never reflect any leap seconds. Upon successful completion, **mktime()** forces the value of the **tm_sec** component to the range [0,59].

AUTHOR

ctime() was developed by AT&T and HP.

SEE ALSO

time(2), getdate(3C), setlocale(3C), strftime(3C), tztab(4), environ(5), hpnls(5), lang(5), langinfo(5).

STANDARDS CONFORMANCE

ctime(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

asctime(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

daylight: AES, SVID2, SVID3, XPG2, XPG3, XPG4

difftime(): AES, SVID3, XPG4, ANSI C

gmtime(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

localtime(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

mktime(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

timezone: AES, SVID3, XPG2, XPG3, XPG4

tzname: AES, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

tzset(): AES, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

asctime_r(),ctime_r(),localtime_r(),gmtime_r(): POSIX.1c

NAME

isalpha(), isupper(), islower(), isdigit(), isxdigit(), isalnum(), isspace(), ispunct(), isprint(), isgraph(), iscntrl(), isascii() - classify characters

SYNOPSIS

```
#include <ctype.h>

int isalnum(int c);
int isalpha(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
int isascii(int c);
```

DESCRIPTION

These functions classify character-coded integer values according to the rules of the coded character set identified by the last successful call to **setlocale()** (see *setlocale*(3C)). Each function is a predicate returning non-zero for true, zero for false.

If **setlocale()** has not been called successfully, characters are classified according to the rules of the default ASCII 7-bit coded character set (see *setlocale*(3C)).

isascii() is defined on all integer values; the other functions are defined for the range **-1** (EOF) through **255**.

The functions return non-zero under the following circumstances; zero otherwise:

isalpha(c**)** *c* is a letter.

isupper(c**)** *c* is an uppercase letter.

islower(c**)** *c* is a lowercase letter.

isdigit(c**)** *c* is a decimal digit (in ASCII: characters [0-9]).

isxdigit(c**)** *c* is a hexadecimal digit (in ASCII: characters [0-9], [A-F] or [a-f]).

isalnum(c**)** *c* is an alphanumeric (letters or digits).

isspace(c**)** *c* is a character that creates "white space" in displayed text (in ASCII: space, tab, carriage return, new-line, vertical tab, and form-feed).

ispunct(c**)** *c* is a punctuation character (in ASCII: any printing character except the space character (040), digits, letters).

isprint(c**)** *c* is a printing character.

isgraph(c**)** *c* is a visible character (in ASCII: printing characters, excluding the space character (040)).

iscntrl(c**)** *c* is a control character (in ASCII: character codes less than 040 and the delete character (0177)).

isascii(c**)** *c* is any ASCII character code between 0 and 0177, inclusive.

If the argument to any of these functions is outside the domain of the function, the result is undefined.

APPLICATION USAGE

The interfaces **isalnum()**, **isalpha()**, **iscntrl()**, **isdigit()**, **isgraph()**, **islower()**, **isprint()**, **ispunct()**, **isspace()**, **isupper()**, **isxdigit()** and **isascii()** are thread-safe and async-cancel-safe.

EXTERNAL INFLUENCES

Locale

The **LC_CTYPE** category determines the classification of character type.

International Code Set Support

Single-byte character code sets are supported.

WARNINGS

These functions are supplied both as library functions and as macros defined in the <**ctype.h**> header. Normally, the macro versions are used. To obtain the library function, either use a **#undef** to remove the macro definition or, if compiling in ANSI-C mode, enclose the function name in parenthesis or take its address. The following example uses the library functions for **isalpha()**, **isdigit()**, and **isspace()**:

```
#include <ctype.h>
#undef isalpha
   ...
main()
{
    int (*ctype_func)();
   ...
    if ( isalpha(c) )
   ...
    if ( (isdigit)(c) )
   ...
    ctype_func = isspace;
   ...
}
```

AUTHOR

ctype() was developed by IBM, OSF, and HP.

SEE ALSO

setlocale(3C), ascii(5).

STANDARDS CONFORMANCE

isalnum(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

isalpha(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

isascii(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

iscntrl(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

isdigit(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

isgraph(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

islower(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

isprint(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

ispunct(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

isspace(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

isupper(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

isxdigit(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

(ENHANCED CURSES)

NAME

cur_term — current terminal information

SYNOPSIS

```
#include <term.h>
extern TERMINAL *cur_term;
```

DESCRIPTION

The external variable *cur_term* identifies the record in the terminfo database associated with the terminal currently in use.

SEE ALSO

del_curterm(3X), tigetflag(3X), <term.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME
curs_set — set the cursor mode

SYNOPSIS
```
#include <curses.h>
int curs_set(int visibility);
```

DESCRIPTION
The **curs_set()** function sets the appearance of the cursor based on the value of *visibility*:

Value of *visibility*	**Appearance of Cursor**
0	Invisible
1	Terminal-specific normal mode
2	Terminal-specific high visibility mode

The terminal does not necessarily support all the above values.

RETURN VALUE
If the terminal supports the cursor mode specified by *visibility*, then **curs_set()** returns the previous cursor state. Otherwise, the function returns ERR.

ERRORS
No errors are defined.

SEE ALSO
<curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 4.

(ENHANCED CURSES)

NAME

curscr — current window

SYNOPSIS

```
#include <curses.h>
extern WINDOW *curscr;
```

DESCRIPTION

The external variable *curscr* points to an internal data structure. It can be specified as an argument to certain functions, such as **clearok()**, where permitted in this specification.

SEE ALSO

clearok(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

(X/Open CURSES)

NAME

curses( ) - Terminal and printer handling and optimization package

DESCRIPTION

Use and Implementation of Interfaces

These routines provide a method for updating screens with reasonable optimization in a terminal independent manner. Each of the following statements applies unless explicitly stated otherwise in the detailed descriptions that follow. If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer outside the address space of the program, or a null pointer), the behaviour is undefined. Any function declared in a header may also be implemented as a macro defined in the header, so a library function should not be declared explicitly if its header is included. Any macro definition of a function can be suppressed locally by enclosing the name of the function in parentheses, because the name is then not followed by the left parenthesis that indicates expansion of a macro function name. For the same syntactic reason, it is permitted to take the address of a library function even if it is also defined as a macro. The use of the C-language **#undef** construct to remove any such macro definition will also ensure that an actual function is referred to. Any invocation of a library function that is implemented as a macro will expand to code that evaluates each of its arguments exactly once, fully protected by parentheses where necessary, so it is generally safe to use arbitrary expressions as arguments. Likewise, those function-like macros described in the following sections may be invoked in an expression anywhere a function with a compatible return type could be called.

Provided that a library function can be declared without reference to any type defined in a header, it is also permissible to declare the function, either explicitly or implicitly, and use it without including its associated header. If a function that accepts a variable number of arguments is not declared (explicitly or by including its associated header), the behaviour is undefined.

As a result of changes introduced in this version of the *Curses Specification*, application writers are only required to include the minimum number of headers. Implementations of XSI-conformant systems will make all necessary symbols visible as described in the Headers section of this document.

C Language Definition

The C language that is the basis for the synopses and code examples in this document is *ISO C*, as specified in the referenced ISO C standard. *Common Usage C*, which refers to the C language before standardisation, was the basis for previous editions of this specification.

The Compilation Environment

Applications should ensure that the feature test macro _XOPEN_SOURCE is defined before inclusion of any header. This is needed to enable the functionality described in this document, and possibly to enable functionality defined elsewhere in the Common Applications Environment.

The _XOPEN_SOURCE macro may be defined automatically by the compilation process, but to ensure maximum portability, applications should make sure that _XOPEN_SOURCE is defined by using either compiler options or **#define** directives in the source files, before any **#include** directives. Identifiers in this document may only be undefined using the **#undef** directive. These **#undef** directives must follow all **#include** directives of any headers.

Most strictly conforming POSIX and ISO C applications will compile on systems compliant to this specification. However, an application which uses any of the items marked as an extension to POSIX and ISO C, for any purpose other than that shown here, may not compile. In such cases, it may be necessary to alter those applications to use alternative identifiers.

Since this document is aligned with the ISO C standard, and since all functionality enabled by the _POSIX_C_SOURCE set equal to 2 should be enabled by _XOPEN_SOURCE, there should be no need to define either _POSIX_SOURCE or _POSIX_C_SOURCE if _XOPEN_SOURCE is defined. Therefore if _XOPEN_SOURCE is defined and _POSIX_SOURCE is defined, or _POSIX_C_SOURCE is set equal to 1 or 2, the behaviour is the same as if only _XOPEN_SOURCE is defined. However should _POSIX_C_SOURCE be set to a value greater than 2, the behaviour is undefined.

The *c89* and *cc* utilities recognise the additional **–l** operand for standard libraries:

–l curses This operand makes visible all library functions referenced in this specification, (except for those labelled ENHANCED CURSES and except for portions marked with the EC margin legend).

If the implementation defines _XOPEN_CURSES and if the application defines the _XOPEN_SOURCE_EXTENDED feature test macro, then **-l curses** also makes

visible all library functions referenced in this specification and labelled ENHANCED CURSES.

An application that uses any API specified as ENHANCED CURSES must define _XOPEN_SOURCE_EXTENDED = 1 in each source file or as part of its compilation environment. When _XOPEN_SOURCE_EXTENDED = 1 is defined in a source file, it must appear before any header is included.

If the implementation supports the utilities marked *DEVELOPMENT* in the *Curses Specification*, the *lint* utility recognises the additional **–l curses** operand for standard libraries:

–l curses Names the library **llib-lcurses.ln ,** which will contain functions specified in this document.

It is unspecified whether the library **llib-lcurses.ln** exists as a regular file.

The X/Open Name Space (ENHANCED CURSES)

The requirements in this section are in effect only for implementations that claim Enhanced Curses compliance.

All identifiers in this document are defined in at least one of the headers **<curses.h>**, **<term.h>**, **<unctrl.h>**. When _XOPEN_SOURCE is defined, each header defines or declares some identifiers, potentially conflicting with identifiers used by the application. The set of identifiers visible to the application consists of precisely those identifiers from the header pages of the included headers, as well as additional identifiers reserved for the implementation. In addition, some headers may make visible identifiers from other headers as indicated on the relevant header pages.

The identifiers reserved for use by the implementation are described below.

1 Each identifier with external linkage described in the header section is reserved for use as an identifier with external linkage if the header is included.

2 Each macro name described in the header section is reserved for any use if the header is included.

3 Each identifier with file scope described in the header section is reserved for use as an identifier with file scope in the same name space if the header is included.

4 All identifiers consisting of exactly 2 upper-case letters.

If any header in the following table is included, identifiers with the prefixes, suffixes or complete names shown are reserved for use by the implementation.

Header	Prefix	Suffix	Complete Name
<curses.h>	**add, attr, get, in, mousr, mv, scr, slk, un, wadd, wattr, wbkg, win**		
ANY header		**_t**	

If any header in the following table is included, macros with the prefixes shown may be defined. After the last inclusion of a given header, an application may use identifiers with the corresponding prefixes for its own purpose, provided their use is preceded by an **#undef** of the corresponding macro.

Header	Prefix
<curses.h>	A_, ACS_, ALL_, BUTTON, COLOR_, KEY_, MOUSE, REPORT_, WA_, WACS_
<term.h>	ext_

The following identifiers are reserved regardless of the inclusion of headers:

1 All identifiers that begin with an underscore and either an upper-case letter or another underscore are always reserved for any use by the implementation.

2 All identifiers that begin with an underscore are always reserved for use as identifiers with file scope in both the ordinary identifier and tag name spaces.

3 All identifiers listed as reserved in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification are reserved for use as identifiers with external linkage.

All the identifiers defined in this document that have external linkage are always reserved for use as identifiers with external linkage.

No other identifiers are reserved.

Applications must not declare or define identifiers with the same name as an identifier reserved in the same context. Since macro names are replaced whenever found, independent of scope and name space, macro names matching any of the reserved identifier names must not be defined if any associated header is included.

Headers may be included in any order, and each may be included more than once in a given scope, with no difference in effect from that of being included only once.

C

If used, a header must be included outside of any external declaration or definition, and it must be first included before the first reference to any type or macro it defines, or to any function or object it declares. However, if an identifier is declared or defined in more than one header, the second and subsequent associated headers may be included after the initial reference to the identifier. Prior to the inclusion of a header, the program must not define any macros with names lexically identical to symbols defined by that header.

Interfaces Implemented as Macros (ENHANCED CURSES)

The requirements in this section are in effect only for implementations that claim Enhanced Curses compliance.

The following interfaces with arguments must be implemented as macros. The relevance to the application programmer is that the '**&**' character cannot be used before the arguments.

Macros	Man Page
COLOR_PAIR(), PAIR_NUMBER()	*can_change_color()*
getbegyx(), *getmaxyx*(), *getparyx*(), *getyx*()	*getbegyx()*

The descriptions in **<curses.h>**, **<term.h>**, **<unctrl.h>** list other macros, like COLOR_BLACK, that do not take arguments.

Relationship to the X/Open System Interfaces and Headers, Issue 4, Version 2 specification

Error Numbers

Most functions provide an error number in *errno* , which is either a variable or macro defined in **<errno.h>**; the macro expands to a modifiable **lvalue** of type **int**.

A list of valid values for *errno* and advice to application writers on the use of *errno* appears in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification .

Data Types

All of the data types used by Curses functions are defined by the implementation. The following list describes these types:

attr_t An integral type that can contain at least an **unsigned short**. The type **attr_t** is used to hold an OR-ed set of attributes defined in **<curses.h>** that begin with the prefix WA_.

bool Boolean data type

chtype An integral type that can contain at least an **unsigned char** and attributes. Values of type **chtype** are formed by OR-ing together an **unsigned char** value and zero or more of the base attribute flags defined in **<curses.h>** that have the A_ prefix. The application can extract these components of a **chtype** value using the base masks defined in **<curses.h>** for this purpose.

The **chtype** data type also contains a colour-pair. Values of type **chtype** are formed by OR-ing together an **unsigned char** value, a colour pair, and zero or more of the attributes defined in **<curses.h>** that begin with the prefix A_. The application can extract these components of a **chtype** value using the masks defined in **<curses.h>** for this purpose.

SCREEN An opaque terminal representation.

wchar_t As described in **<stddef.h>**.

cchar_t A type that can reference a string of wide characters of up to an implementation-dependent length, a colour-pair, and zero or more attributes from the set of all attributes defined in this document. A null **cchar_t** object is an object that references a empty wide-character string. Arrays of **cchar_t** objects are terminated by a null **cchar_t** object.

WINDOW An opaque window representation.

Interface Overview

Components

A Curses initialisation function, usually **initscr()**, determines the terminal model in use, by reference to either an argument or an environment variable. If that model is defined in **terminfo**, then the same **terminfo** entry tells Curses exactly how to operate the terminal.

In this case, a comprehensive API lets the application perform terminal operations. The Curses run-time system receives each terminal request and sends appropriate commands to the terminal to achieve the desired effect.

C

Relationship to the X/Open System Interface Definitions, Issue 4, Version 2 specification

Applications using Curses should not also control the terminal using capabilities of the general terminal interface defined in the *X/Open System Interface Definitions, Issue 4, Version 2* specification, Chapter 9, *General Terminal Interface*.

There is no requirement that the paradigms that exist while in Curses mode be carried over outside the Curses environment (see the **def_prog_mode()** page).

Relationship to Signals

Curses implementations may provide for special handling of the SIGINT, SIGQUIT and SIGTSTP signals if their disposition is SIGDFL at the time **initscr()** is called (see the **initscr()** page).

Any special handling for these signals may remain in effect for the life of the process or until the process changes the disposition of the signal.

None of the Curses functions are required to be safe with respect to signals (see **sigaction()** in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification).

The behaviour of Curses with respect to signals not defined by the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification is unspecified.

Screens, Windows and Terminals

Screen

A screen is the physical output device of the terminal. In Curses, a **SCREEN** data type is an opaque data type associated with a terminal. Each window (see below) is associated with a **SCREEN**.

Windows

The Curses functions permit manipulation of *window* objects, which can be thought of as two-dimensional arrays of characters and their renditions. A default window called *stdscr* , which is the size of the terminal screen, is supplied. Others may be created with **newwin()**.

Variables declared as **WINDOW *** refer to windows (and to subwindows, derived windows, and pads, as described below). These data structures are manipulated with functions described on the reference manual pages in terminfo(4). Among the most basic functions are **move()** and **addch()**. More general versions of these functions are included that allow a process to specify a window.

After using functions to manipulate a window, **refresh()** is called, telling Curses to make the CRT screen look like *stdscr* .

Line drawing characters may be specified to be output. On input, Curses is also able to translate arrow and function keys that transmit escape sequences into single values. The line drawing characters and input values use names defined in **<curses.h>**.

Each window has a flag that indicates that the information in the window could differ from the information displayed on the terminal device. Making any change to the contents of the window, moving or modifying the window, or setting the window's cursor position, sets this flag (*touches* the window).

Subwindows

A *subwindow* is a window, created within another window (called the *parent window)*, and positioned relative to the parent window. A subwindow can be created by calling **derwin()**, **newpad()** or **subwin()**. Changes made to a subwindow do not affect its parent window.

Subwindows can be created from a parent window by calling **subwin()**. The position and size of subwindows on the screen must be identical to or totally within the parent window. Changes to either the parent window or the subwindow affect both. Window clipping is not a property of subwindows.

Ancestors
The term *ancestor* refers to a window's parent, or its parent, or so on.

Derived Windows
Derived windows are subwindows whose position is defined by reference to the parent window rather than in absolute screen coordinates. Derived windows are otherwise no different from subwindows.

Pads
A pad is a specialised case of subwindow that is not necessarily associated with a viewable part of a screen. Functions that deal with pads are all discussed in **newpad()**.

Terminal
A terminal is the logical input and output device through which character-based applications interact with the user. **TERMINAL** is an opaque data type associated with a terminal. A **TERMINAL** data structure primarily contains information about the capabilities of the terminal, as defined by **terminfo**. A **TERMINAL** also contains information about the terminal modes and current state for input and output operations. Each screen (see above) is associated with a **TERMINAL**.

Characters

Character Storage Size
Historically, a position on the screen has corresponded to a single stored byte. This correspondence is no longer true for several reasons:

- Some characters may occupy several columns when displayed on the screen (see *Multi-column Characters*).
- Some characters may be non-spacing characters, defined only in association with a spacing character (see *Non-spacing Characters (ENHANCED CURSES)*).
- The number of bytes to hold a character from the extended character sets depends on the LC_CTYPE locale category.

The internal storage format of characters and renditions is unspecified. There is no implied correspondence between the internal storage format and the external representation of characters and renditions in objects of type **chtype** and **cchar_t**.

Multi-column Characters
Some character sets define *multi-column characters* that occupy more than one column position when displayed on the screen.

Writing a character whose width is greater than the width of the destination window is an error.

Attributes
Each character can be displayed with *attributes* such as underlining, reverse video or colour on terminals that support such display enhancements. Current attributes of a window are applied to all characters that are written into the window with **waddch()**, **wadd_wch()**, **waddstr()**, **waddchstr()**, **waddwstr()**, **waddwchstr()** and **wprintw()**. Attributes can be combined.

Attributes can be specified using constants with the A_ prefix specified in **<curses.h>**. The A_ constants manipulate attributes in objects of type **chtype**. Additional attributes can be specified using constants with the WA_ prefix. The WA_ constants manipulate attributes in objects of type **attr_t**.

Two constants that begin with A_ and WA_ and that represent the same terminal capability refer to the same attribute in the **terminfo** database and in the window data structure. The effect on a window does not differ depending on whether the application specifies A_ or WA_ constants. For example, when an application updates window attributes using the interfaces that support the A_ values, a query of the window attribute using the function that returns WA_ values reflects this update. When it updates window attributes using the interfaces that support the WA_ values, for which corresponding A_ values exist, a query of the window attribute using the function that returns A_ values reflects this update.

Rendition
The *rendition* of a character displayed on the screen is its attributes and a colour pair.

The rendition of a character written to the screen becomes a property of the character and moves with the character through any scrolling and insert/delete line/character operations. To the extent possible on a particular terminal, a character's rendition corresponds to the graphic rendition of the character put on the

screen.

If a given terminal does not support a rendition that an application program is trying to use, Curses may substitute a different rendition for it.

Colours are always used in pairs (referred to as colour-pairs). A colour-pair consists of a foreground colour (for characters) and a background colour (for the field on which the characters are displayed).

Non-spacing Characters

The requirements in this section are in effect only for implementations that claim Enhanced Curses compliance.

Some character sets may contain *non-spacing* characters. (Non-spacing characters are those for which **wcwidth()** returns a width of zero.) The application may write non-spacing characters to a window. Every non-spacing character in a window is associated with a spacing character and modifies the spacing character. Non-spacing characters in a window cannot be addressed separately. A non-spacing character is implicitly addressed whenever a Curses operation affects the spacing character with which the non-spacing character is associated.

Non-spacing characters do not support attributes. For interfaces that use wide characters and attributes, the attributes are ignored if the wide character is a non-spacing character. Multi-column characters have a single set of attributes for all columns. The association of non-spacing characters with spacing characters can be controlled by the application using the wide character interfaces. The wide character string functions provide codeset-dependent association.

Two typical effects of a non-spacing character associated with a spacing character called *c*, are as follows:

- The non-spacing character may modify the appearance of *c*. (For instance, there may be non-spacing characters that add diacritical marks to characters. However, there may also be spacing characters with built-in diacritical marks.)
- The non-spacing character may bridge *c* to the character following *c*. (Examples of this usage are the formation of ligatures and the conversion of characters into compound display forms, words, or ideograms.)

Implementations may limit the number of non-spacing characters that can be associated with a spacing character, provided any limit is at least 5.

Complex Characters

A *complex character* is a set of associated characters, which may include a spacing character and may include any non-spacing characters associated with it. A *spacing complex character* is a spacing character followed by any non-spacing characters associated with it. That is, a spacing complex character is a complex character that includes one spacing character. An example of a code set that has complex characters is ISO/IEC 10646-1:1993.

A complex character can be written to the screen; if it does not include a spacing character, any non-spacing characters are associated with the spacing complex character that exists at the specified screen position. When the application reads information back from the screen, it obtains spacing complex characters.

The **cchar_t** data type represents a complex character and its rendition. When a **cchar_t** represents a non-spacing complex character (that is, when there is no spacing character within the complex character), then its rendition is not used; when it is written to the screen, it uses the rendition specified by the spacing character already displayed.

An object of type **cchar_t** can be initialised using **setcchar()** and its contents can be extracted using **getcchar()**. The behaviour of functions that take a **cchar_t** input argument is undefined if the application provides a **cchar_t** value that was not initialised in this way or obtained from a Curses function that has a **cchar_t** output argument.

Window Properties

Associated with each window are the following properties that affect the placing of characters into the window (see *Rendition of Characters Placed into a Window* in curses_intro).

Window Rendition

Each window has a rendition, which is separate from the rendition component of the window's background property described below.

Window Background

Each window has a background property. The background property specifies:

- A spacing complex character (the background character) that will be used in a variety of situations where visible information is deleted from the screen.
- A rendition to use in displaying the background character in those situations, and in other situations specified in *Rendition of Characters Placed into a Window* in curses_intro.

Conceptual Operations

Screen Addressing

Many Curses functions use a coordinate pair. In the *DESCRIPTION*, coordinate locations are represented as (*y*, *x*) since the *y* argument always precedes the *x* argument in the function call. These coordinates denote a line/column position, not a character position.

The coordinate *y* always refers to the row (of the window), and *x* always refers to the column. The first row and the first column is number 0, not 1. The position (0, 0) is the window's *origin* .

For example, for terminals that display the ISO 8859-1 character set (with left-to-right writing), (0, 0) represents the upper left-hand corner of the screen.

Functions that start with *mv* take arguments that specify a (*y*, *x*) position and move the cursor (as though **move()** were called) before performing the requested action. As part of the requested action, further cursor movement may occur, specified on the respective reference manual page.

Basic Character Operations

Adding (Overwriting)

The Curses functions that contain the word add, such as **addch()**, actually specify one or more characters to replace (overwrite) characters already in the window. If these functions specify only non-spacing characters, they are appended to a spacing character already in the window; see also *Non-spacing Characters (ENHANCED CURSES)*.

When replacing a multi-column character with a character that requires fewer columns, the new character is added starting at the specified or implied column position. All columns that the former multi-column character occupied that the new character does not require are *orphaned columns*, which are filled using the background character and rendition.

Replacing a character with a character that requires more columns also replaces one or more subsequent characters on the line. This process may also produce orphaned columns.

Truncation, Wrapping and Scrolling

If the application specifies a character or a string of characters such that writing them to a window would extend beyond the end of the line (for example, if the application tries to deposit any multi-column character at the last column in a line), the behaviour depends on whether the function supports line wrapping:

- If the function does not wrap, it fails.
- If the function wraps, then it places one or more characters in the window at the start of the next line, beginning with the first character that would not completely fit on the original line.

If the final character on the line is a multi-column character that does not completely fit on the line, the entire character wraps to the next line and columns at the end of the original line may be orphaned.

If the original line was the last line in the window, the wrap may cause a scroll to occur:

— If scrolling is enabled, a scroll occurs. The contents of the first line of the window are lost. The contents of each remaining line in the window move to the previous line. The last line of the window is filled with any characters that wrapped. Any remaining space on the last line is filled with the background character and rendition.

— If scrolling is disabled, any characters that would extend beyond the last column of the last line are truncated.

The **scrollok()** function enables and disables scrolling.

Some *add* functions move the cursor just beyond the end of the last character added. If this position is beyond the end of a line, it causes wrapping and scrolling under the conditions specified in the second bullet above.

Insertion

Insertion functions (such as **insch()**) insert characters immediately before the character at the specified or implied cursor position.

The insertion shifts all characters that were formerly at or beyond the cursor position on the cursor line toward the end of that line. The disposition of the characters that would thus extend beyond the end of the line depends on whether the function supports wrapping:

- If the function does not wrap, those characters are removed from the window. This may produce orphaned columns.
- If the function supports wrapping, the effect is as described above in **truncation** (except that the overwriting discussed in the final dash is an insertion).

If multi-column characters are displayed, some cursor positions are within a multi-column character but not at the beginning of a character. Any request to insert data at a position that is not the beginning of a multi-column character will be adjusted so that the actual cursor position is at the beginning of the multi-column character in which the requested position occurs.

There are no warning indications relative to cursor relocation. The application should not maintain an image of the cursor position, since this constitutes placing terminal-specific information in the application and defeats the purpose of using Curses.

Portable applications cannot assume that a cursor position specified in an insert function is a reusable indication of the actual cursor position.

Deletion

Deletion functions (such as **delch()**) delete the simple or complex character at the specified or implied cursor position, no matter which column of the character this is. All column positions are replaced by the background character and rendition and the cursor is not relocated. If a character-deletion operation would cause a previous wrapping operation to be undone, then the results are unspecified.

Window Operations

Overlapping a window (that is, placing one window on top of another) and overwriting a window (that is, copying the contents of one window into another) follows the operation of overwriting multi-column glyphs around its edge. Any orphaned columns are handled as in the character operations.

Characters that Straddle the Subwindow Border

A subwindow can be defined such that multi-column characters straddle the subwindow border. The character operations deal with these straddling characters as follows:

- Reading the subwindow with a function such as **in_wch()** reads the entire straddling character.
- Adding, inserting or deleting in the subwindow deletes the entire straddling character before the requested operation begins and does not relocate the cursor.
- Scrolling lines in the subwindow has the following effects:
  - — A straddling character at the start of the line is completely erased before the scroll operation begins.
  - — A straddling character at the end of the line moves in the direction of the scroll and continues to straddle the subwindow border. Column positions outside the subwindow at the straddling character's former position are orphaned unless another straddling character scrolls into those positions.

If the application calls a function such as **border()**, the above situations do not occur because writing the border on the subwindow deletes any straddling characters.

In the above cases involving multi-column characters, operations confined to a subwindow can modify the screen outside the subwindow. Therefore, saving a subwindow, performing operations within the subwindow, and then restoring the subwindow may disturb the appearance of the screen. To overcome these effects (for example, for pop-up windows), the application should refresh the entire screen.

Special Characters

Some functions process special characters as specified below.

In functions that do not move the cursor based on the information placed in the window, these special characters would only be used within a string in order to affect the placement of subsequent characters; the cursor movement specified below does not persist in the visible cursor beyond the end of the operation. In functions that do move the cursor, these special characters can be used to affect the placement of subsequent characters and to achieve movement of the visible cursor.

<backspace> Unless the cursor was already in column 0, <backspace> moves the cursor one column toward the start of the current line and any characters after the <backspace> are added or inserted starting there.

<carriage return> Unless the cursor was already in column 0, <carriage return> moves the cursor to the start of the current line. Any characters after the <carriage return> are added or inserted starting there.

<newline> In an add operation, Curses adds the background character into successive columns until reaching the end of the line. Scrolling occurs as described in *Truncation, Wrapping and Scrolling* in curses_intro. Any characters after the <newline> character are added at the start of the new line.

In an insert operation, <newline> moves the cursor to the start of a new line (causing scrolling as described in *Truncation, Wrapping and Scrolling* in curses_intro). Any characters after the <newline> character are placed at the start of the new line.

The **filter()** function may inhibit this processing.

<tab> Tab characters in text move subsequent characters to the next horizontal tab stop. By default, tab stops are in column 0, 8, 16, and so on.

In an insert or add operation, Curses inserts or adds, respectively, the background character into successive columns until reaching the next tab stop. If there are no more tab stops in the current line, wrapping and scrolling occur as described in *Truncation, Wrapping and Scrolling* in curses_intro.

Control Characters

The Curses functions that perform special-character processing conceptually convert control characters to the caret ('^') character followed by a second character (which is an upper-case letter if it is alphabetic) and write this string to the window in place of the control character. The functions that retrieve text from the window will not retrieve the original control character.

Rendition of Characters Placed into a Window

When the application adds or inserts characters into a window, the effect is as follows:

If the character is not the space character, then the window receives:

- the character that the application specifies
- the colour that the application specifies; or the window colour, if the application does not specify a colour
- the attributes specified, OR-ed with the window attributes.

If the character is the space character, then the window receives:

- the background character
- the colour that the application specifies; or the background colour, if the application does not specify a colour
- the attributes specified, OR-ed with the background attributes.

Input Processing

The Curses input model provides a variety of ways to obtain input from the keyboard.

Keypad Processing

The application can enable or disable *keypad translation* by calling **keypad()**. When translation is enabled, Curses attempts to translate a sequence of terminal input that represents the pressing of a function key into a single key code. When translation is disabled, Curses passes terminal input to the application without such translation, and any interpretation of the input as representing the pressing of a keypad key must be done by the application.

The complete set of key codes for keypad keys that Curses can process is specified by the constants defined in **<curses.h>** whose names begin with "KEY_". Each terminal type described in the **terminfo** database may support some or all of these key codes. The **terminfo** database specifies the sequence of input characters from the terminal type that correspond to each key code (see *Keypad* in *terminfo*(4)).

The Curses implementation cannot translate keypad keys on terminals where pressing the keys does not transmit a unique sequence.

When translation is enabled and a character that could be the beginning of a function key (such as escape) is received, Curses notes the time and begins accumulating characters. If Curses receives additional characters that represent the pressing of a keypad key, within an unspecified interval from the time the first character was received, then Curses converts this input to a key code for presentation to the application. If such characters are not received during this interval, translation of this input does not occur and the individual characters are presented to the application separately. (Because Curses waits for this interval to accumulate a key code, many terminals experience a delay between the time a user presses the escape key and the time the escape is returned to the application.)

In addition, No Timeout Mode provides that in any case where Curses has received part of a function key sequence, it waits indefinitely for the complete key sequence. The "unspecified interval" in the previous paragraph becomes infinite in No Timeout Mode. No Timeout Mode allows the use of function keys over slow communication lines. No Timeout Mode lets the user type the individual characters of a function key sequence, but also delays application response when the user types a character (not a function key) that begins a function key sequence. For this reason, in No Timeout Mode many terminals will appear to hang between the time a user presses the escape key and the time another key is pressed. No Timeout Mode is switchable by calling **notimeout()**.

If any special characters (see *Special Characters* in curses_intro) are defined or redefined to be characters that are members of a function key sequence, then Curses will be unable to recognise and translate those function keys.

Several of the modes discussed below are described in terms of availability of input. If keypad translation is enabled, then input is not available once Curses has begun receiving a keypad sequence until the sequence is completely received or the interval has elapsed.

Input Mode

The *X/Open System Interface Definitions, Issue 4, Version 2* specification (*Special Characters*) defines flow-control characters, the interrupt character, the erase character, and the kill character. Four mutually-exclusive Curses modes let the application control the effect of these input characters:

Input Mode	Effect
Cooked Mode	This achieves normal line-at-a-time processing with all special characters handled outside the application. This achieves the same effect as canonical-mode input processing as specified in the *X/Open System Interface Definitions, Issue 4, Version 2* specification . The state of the ISIG and IXON flags are not changed upon entering this mode by calling **cbreak**(), and are set upon entering this mode by calling **noraw**(). The implementation supports erase and kill characters from any supported locale, no matter what the width of the character is.
cbreak Mode	Characters typed by the user are immediately available to the application and Curses does not perform special processing on either the erase character or the kill character. An application can select *cbreak* mode to do its own line editing but to let the abort character be used to abort the task. This mode achieves the same effect as non-canonical-mode, Case B input processing (with MIN set to 1 and ICRNL cleared) as specified in the *X/Open System Interface Definitions, Issue 4, Version 2* specification. The state of the ISIG and IXON flags are not changed upon entering this mode.
Half-Delay Mode	The effect is the same as *cbreak*, except that input functions wait until a character is available or an interval defined by the application elapses, whichever comes first. This mode achieves the same effect as non-canonical-mode, Case C input processing (with TIME set to the value specified by the application) as specified in the *X/Open System Interface Definitions, Issue 4, Version 2* specification . The state of the ISIG and IXON flags are not changed upon entering this mode.
Raw Mode	Raw mode gives the application maximum control over terminal input. The application sees each character as it is typed. This achieves the same effect as non-canonical mode, Case D input processing as specified in the *X/Open System Interface Definitions, Issue 4, Version 2* specification . The ISIG and IXON flags are cleared upon entering this mode.

The terminal interface settings are recorded when the process calls **initscr()** or **newterm()** to initialise Curses and restores these settings when **endwin()** is called. The initial input mode for Curses operations is unspecified unless the implementation supports Enhanced Curses compliance, in which the initial input mode is *cbreak* mode.

The behaviour of the BREAK key depends on other bits in the display driver that are not set by Curses.

Delay Mode

Two mutually-exclusive delay modes specify how quickly certain Curses functions return to the application when there is no terminal input waiting when the function is called:

No Delay The function fails.

Delay The application waits until the implementation passes text through to the application. If *cbreak* or Raw Mode is set, this is after one character. Otherwise, this is after the first <newline> character, end-of-line character, or end-of-file character.

The effect of No Delay Mode on function key processing is unspecified.

Echo Processing

Echo mode determines whether Curses echoes typed characters to the screen. The effect of Echo mode is analogous to the effect of the ECHO flag in the local mode field of the **termios** structure associated with the terminal device connected to the window. However, Curses always clears the ECHO flag while it is operating, to inhibit the operating system from performing echoing. The method of echoing characters is not identical to the operating system's method of echoing characters, because Curses performs additional processing of terminal input.

If in Echo mode, Curses performs its own echoing: Any visible input character is stored in the current or specified window by the input function that the application called, at that window's cursor position, as though **addch( )** were called, with all consequent effects such as cursor movement and wrapping.

If not in Echo mode, any echoing of input must be performed by the application. Applications often perform their own echoing in a controlled area of the screen, or do not echo at all, so they disable Echo mode.

The Set of Curses Functions

The Curses functions allow: overall screen, window and pad manipulation; output to windows and pads; reading terminal input; control over terminal and Curses input and output options; environment query functions; colour manipulation; use of soft label keys; access to the **terminfo** database of terminal capabilities; and access to low-level functions.

Function Name Conventions

The reference manual pages present families of multiple Curses functions. Most function families have different functions that give the programmer the following options:

- A function with the basic name operates on the window *stdscr* . A function with the same name plus the *w* prefix operates on a window specified by the *win* argument.

  When the reference manual page for a function family refers to the **current or specified window**, it means *stdscr* for the basic functions and the window specified by *win* for any *w* function.

  Functions whose names have the *p* prefix require an argument that is a pad instead of a window.

- A function with the basic name operates based on the current cursor position (of the current or specified window, as described above). A function with the same name plus the *mv* prefix moves the cursor to a position specified by the *y* and *x* arguments before performing the specified operation.

  When the reference manual page for a function family refers to the **current or specified position**, it means the cursor position for the basic functions and the position (*y*, *x*) for any *mv* function.

  The *mvw* prefix exists and combines the *mv* semantics discussed here with the *w* semantics discussed above. The window argument is always specified before the coordinates.

- A function with the basic name is often provided for historical compatibility and operates only on single-byte characters. A function with the same name plus the *w* infix operates on wide (multibyte) characters. A function with the same name plus the *_w* infix operates on complex characters and their renditions.

- When a function with the basic name operates on a single character, there is sometimes a function with the same name plus the *n* infix that operates on multiple characters. An *n* argument specifies the number of characters to process. The respective manual page specifies the outcome if the value of *n* is inappropriate.

(X/Open CURSES)

Function Families Provided

Function Names	Description	s	w	c	Refer to
	Add (Overwrite)				
[*mv*][*w*]*addch*(3X)	add a character	Y	Y	Y	*addch(3X)*
[*mv*][*w*]*addch*[*n*]*str*(3X)	add a character string	N	N	N	*addchstr(3X)*
[*mv*][*w*]*add*[*n*]*str*(3X)	add a string	Y	Y	Y	*addnstr(3X)*
[*mv*][*w*]*add*[*n*]*wstr*(3X)	add a wide character string	Y	Y	Y	*addnwstr(3X)*
[*mv*][*w*]*add_wch*(3X)	add a wide character and rendition	Y	Y	Y	*add_wch(3X)*
[*mv*][*w*]*add_wch*[*n*]*str*(3X)	add an array of wide characters and renditions	?	N	N	*add_wchnstr(3X)*
	Change Renditions				
[*mv*][*w*]*chgat*(3X)	change renditions of characters in a window	-	N	N	*chgat(3X)*
	Delete				
[*mv*][*w*]*delch*(3X)	delete a character	-	-	N	*delch(3X)*
	Get (Input from Keyboard to Window)				
[*mv*][*w*]*getch*(3X)	get a character	Y	Y	Y	*getch(3X)*
[*mv*][*w*]*get*[*n*]*str*(3X)	get a character string	Y	Y	Y	*getnstr(3X)*
[*mv*][*w*]*get_wch*(3X)	get a wide character	Y	Y	Y	*get_wch(3X)*
[*mv*][*w*]*get*[*n*]*_wstr*(3X)	get an array of wide characters and key codes	Y	Y	Y	*get_wstr(3X)*
	Explicit Cursor Movement				
[*w*]*move*(3X)	move the cursor	-	-	-	*move(3X)*
	Input (Read Back from Window)				
[*mv*][*w*]*inch*(3X)	input a character	-	-	-	*inch(3X)*
[*mv*][*w*]*inch*[*n*]*str*(3X)	input an array of characters and attributes	-	-	-	*inchnstr(3X)*
[*mv*][*w*]*in*[*n*]*str*(3X)	input a string	-	-	-	*innstr(3X)*
[*mv*][*w*]*in*[*n*]*wstr*(3X)	input a string of wide characters	-	-	-	*innwstr(3X)*
[*mv*][*w*]*in_wch*(3X)	input a wide character and rendition	-	-	-	*in_wch(3X)*
[*mv*][*w*]*in_wch*[*n*]*str*(3X)	input an array of wide characters and renditions	-	-	-	*inwchnstr(3X)*
	Insert				
[*mv*][*w*]*insch*(3X)	insert a character	Y	N	N	*insch(3X)*
[*mv*][*w*]*ins*[*n*]*str*(3X)	insert a character string	Y	N	N	*insnstr(3X)*
[*mv*][*w*]*ins_*[*n*]*wstr*(3X)	insert a wide-character string	Y	N	N	*ins_nwstr(3X)*
[*mv*][*w*]*ins_wch*(3X)	insert a wide character	Y	N	N	*ins_wch(3X)*
	Print and Scan				
[*mv*][*w*]*printw*(3X)	print formatted output	-	-	-	*mvprintw(3X)*
[*mv*][*w*]*scanw*(3X)	convert formatted output	-	-	-	*mvscanw(3X)*

Legend

The following notation indicates the effect when characters are moved to the screen. (For the Get functions, this applies only when echoing is enabled.)

S Y means these functions perform special-character processing (see *Special Characters* in curses_intro). N means they do not. ? means the results are unspecified when these functions are applied to special characters.

W Y means these functions perform wrapping (see *Truncation, Wrapping and Scrolling* in curses_intro). N means they do not.

C Y means these functions advance the cursor (see *Truncation, Wrapping and Scrolling* in curses_intro). N means they do not.

- The attribute specified by this column does not apply to these functions.

Interfaces Implemented as Macros

The following interfaces with arguments must be implemented as macros. The relevance to the application programmer is that the '**&**' character cannot be used before the arguments.

Macros	Man page Entry
COLOR_PAIR()	*can_change_color*()
getbegyx(), *getmaxyx*(), *getparyx*(), *getyx*()	*getbegyx*()

The header file reference manual pages list other macros, like COLOR_BLACK, that do not take arguments.

Initialised Curses Environment

Before executing an application that uses Curses, the terminal must be prepared as follows:

- If the terminal has hardware tab stops, they should be set.
- Any initialisation strings defined for the terminal must be output to the terminal.

The resulting state of the terminal must be compatible with the model of the terminal that Curses has, as reflected in the terminal's entry in the **terminfo** database (see *terminfo*(4)).

To initialise Curses, the application must call **initscr()** or **newterm()** before calling any of the other functions that deal with windows and screens, and it must call **endwin()** before exiting. To get character-at-a-time input without echoing (most interactive, screen-oriented programs want this), the following sequence should be used:

```
initscr();
cbreak();
noecho();
```

Most programs would additionally use the sequence:

```
nonl();
intrflush(stdscr, FALSE);
keypad(stdscr, TRUE);
```

Synchronous and Networked Asynchronous Terminals

This section indicates to the application writer some considerations to be borne in mind when driving synchronous, networked asynchronous (NWA) or non-standard directly-connected asynchronous terminals.

Such terminals are often used in a mainframe environment and communicate to the host in block mode. That is, the user types characters at the terminal then presses a special key to initiate transmission of the characters to the host.

Frequently, although it may be possible to send arbitrary sized blocks to the host, it is not possible or desirable to cause a character to be transmitted with only a single keystroke.

This can cause severe problems to an application wishing to make use of single-character input; see *Input Processing* in curses_intro.

Output

The Curses interface can be used in the normal way for all operations pertaining to output to the terminal, with the possible exception that on some terminals the **refresh()** routine may have to redraw the entire screen contents in order to perform any update.

If it is additionally necessary to clear the screen before each such operation, the result could be undesirable.

Input

Because of the nature of operation of synchronous (block-mode) and NWA terminals, it might not be possible to support all or any of the Curses input functions. In particular, the following points should be noted:

- Single-character input might not be possible. It may be necessary to press a special key to cause all characters typed at the terminal to be transmitted to the host.
- It is sometimes not possible to disable echo. Character echo may be performed directly by the terminal. On terminals that behave in this way, any Curses application that performs input should be aware that any characters typed will appear on the screen at wherever the cursor is positioned. This does not necessarily correspond to the position of the cursor in the window.

SEE ALSO

curses(5).

NAME

cuserid( ) - get character login name of the user

SYNOPSIS

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

Remarks:

Because this function behaved differently in previous releases of HP-UX, and behaves differently on other systems, its use is not recommended. It is provided only for conformance to current industry standards, and is subject to withdrawal in future releases of HP-UX.

For portability and security, application writers and maintainers should search their existing code and replace references to **cuserid()** with equivalent calls to **getpwuid** (**getuid()**), **getpwuid** (**geteuid()**), or **getlogin()**, depending on which user name is desired.

DESCRIPTION

cuserid() generates a character-string representation of the user name corresponding to the effective 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.

For multi-thread applications, if *s* is a NULL pointer, the operation is not performed and a NULL pointer is returned.

APPLICATION USAGE

cuserid() is thread-safe. It is not async-cancel-safe. A cancellation point may occur when a thread is executing **cuserid()**.

DIAGNOSTICS

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

SEE ALSO

geteuid(2), getlogin(3C), getpwuid(3C).

STANDARDS CONFORMANCE

cuserid(): AES, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

datalock() - lock process into memory after allocating data and stack space

SYNOPSIS

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

int datalock(size_t datsiz, size_t stsiz);
```

DESCRIPTION

datalock() allocates at least *datsiz* bytes of data space and *stsiz* bytes of stack space, then locks the program in memory. The data space is allocated by **malloc()** (see *malloc*(3C)). After the program is locked, this space is released by **free()** (see *malloc*(3C)), making it available for use. This allows the calling program to use that much space dynamically without receiving the **SIGSEGV** signal.

The effective user ID of the calling process must be super-user or be a member of or have an effective group ID of a group having PRIV_MLOCK access to use this call (see *setprivgrp*(2)).

EXAMPLES

The following call to **datalock()** allocates 4096 bytes of data space and 2048 bytes of stack space, then locks the process in memory:

```
datalock (4096, 2048);
```

APPLICATION USAGE

datalock() is thread-safe. It is not async-cancel-safe.

RETURN VALUE

datalock() returns -1 if **malloc()** cannot allocate enough memory or if **plock()** returned an error (see *plock*(2)).

WARNINGS

Multiple datalocks cannot be the same as one big one.

Methods for calculating the required size are not yet well developed.

AUTHOR

datalock() was developed by HP.

SEE ALSO

getprivgrp(2), plock(2).

NAME

dbminit, fetch, store, delete, firstkey, nextkey, dbmclose - database subroutines

SYNOPSIS

```
#include <dbm.h>
int dbminit(const char *file);
datum fetch(datum key);
int store(datum key, datum content);
int delete(datum key);
datum firstkey(void);
datum nextkey(datum key);
int dbmclose(void);
```

DESCRIPTION

These functions maintain key/content pairs in a database. They handle very large (a billion blocks (block = 1024 bytes)) databases and can locate a keyed item in one or two file system accesses.

key and *content* parameters are described by the **datum** type. A **datum** specifies a string of *dsize* bytes pointed to by *dptr*. Arbitrary binary data, as well as normal ASCII strings, are allowed. The database is stored in two files. One file is a directory containing a bit map of keys and has **.dir** as its suffix. The second file contains all data and has **.pag** as its suffix.

Before a database can be accessed, it must be opened by **dbminit**. At the time of this call, the files *file***.dir** and *file***.pag** must exist. (An empty database is created by creating zero-length **.dir** and **.pag** files.)

Once open, data stored under a key is accessed by *fetch***,** and data is placed under a key by *store***.** Storing data on an existing key replaces the existing data. A key (and its associated contents) is deleted by **delete**. A linear pass through all keys in a database can be made, in (apparently) random order by using **firstkey** and **nextkey**. **firstkey** returns the first key in the database. With any key, **nextkey** returns the next key in the database. The following code can be used to traverse the database:

```
for (key = firstkey(); key.dptr != NULL; key = nextkey(key))
```

A database can be closed by calling **dbmclose.** A currently open database must be closed before opening a new one.

DIAGNOSTICS

All functions that return an **int** indicate errors with negative values and success with zero. Functions that return a **datum** indicate errors with a null *dptr*.

WARNINGS

The *dbm* functions provided in this library should not be confused in any way with those of a general-purpose database management system such as ALLBASE/HP-UX SQL. These functions *do not* provide for multiple search keys per entry, they *do not* protect against multi-user access (in other words they do not lock records or files), and they *do not* provide the many other useful data base functions that are found in more robust database management systems. Creating and updating databases by use of these functions is relatively slow because of data copies that occur upon hash collisions. These functions *are useful* for applications requiring fast lookup of relatively static information that is to be indexed by a single key.

The **.pag** file will contain holes so that its apparent size is about four times its actual content. Some older UNIX systems create real file blocks for these holes when touched. These files cannot be copied by normal means (such as *cp*(1), *cat*(1), *tar*(1), or *ar*(1)) without expansion.

dptr pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 1024 bytes). Moreover, all key/content pairs that hash together must fit on a single block. **store** returns an error if a disk block fills with inseparable data.

delete does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by **firstkey** and **nextkey** depends on a hashing function, not on anything interesting.

A **store** or **delete** during a pass through the keys by **firstkey** and **nextkey** may yield unexpected results.

AUTHOR

dbm(3C) was developed by the University of California, Berkeley.

SEE ALSO

ndbm(3X).

NAME

def_prog_mode, def_shell_mode, reset_prog_mode, reset_shell_mode — save or restore program or shell terminal modes

SYNOPSIS

```
#include <curses.h>
int def_prog_mode(void);
int def_shell_mode(void);
int reset_prog_mode(void);
int reset_shell_mode(void);
```

DESCRIPTION

The **def_prog_mode()** function saves the current terminal modes as the "program" (in Curses) state for use by **reset_prog_mode()**.

The **def_shell_mode()** function saves the current terminal modes as the "shell" (not in Curses) state for use by **reset_shell_mode()**.

The **reset_prog_mode()** function restores the terminal to the "program" (in Curses) state.

The **reset_shell_mode()** function restores the terminal to the "shell" (not in Curses) state.

These functions affect the mode of the terminal associated with the current screen.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

The **initscr()** function achieves the effect of calling **def_shell_mode()** to save the prior terminal settings so they can be restored during the call to **endwin()**, and of calling **def_prog_mode()** to specify an initial definition of the program terminal mode.

Applications normally do not need to refer to the shell terminal mode. Applications may find it useful to save and restore the program terminal mode.

SEE ALSO

doupdate(3X), endwin(3X), initscr(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4

The **reset_prog_mode()** and **reset_shell_mode()** functions are merged with this entry. In previous issues, they appeared in entries of their own.

The entry is rewritten for clarity. The argument list for all these functions is explicitly declared as **void**.

NAME

del_curterm, restartterm, set_curterm, setupterm — interfaces to the **terminfo** database

SYNOPSIS

```
#include <term.h>
int del_curterm(TERMINAL *oterm);
int restartterm(char *term, int fildes, int *errret);
TERMINAL *set_curterm(TERMINAL *nterm);
int setupterm(char *term, int fildes, int *errret);
extern TERMINAL *cur_term;
```

DESCRIPTION

These functions retrieve information from the **terminfo** database.

To gain access to the **terminfo** database, **setupterm()** must be called first. It is automatically called by **initscr()** and **newterm()**. The **setupterm()** function initializes the other functions to use the **terminfo** record for a specified terminal (which depends on whether **use_env()** was called). It sets the *cur_term* external variable to a **TERMINAL** structure that contains the record from the **terminfo** database for the specified terminal.

The terminal type is the character string *term*; if *term* is a null pointer, the environment variable TERM is used. If TERM is not set or if its value is an empty string, then **"unknown"** is used as the terminal type. The application must set *fildes* to a file descriptor, open for output, to the terminal device, before calling **setupterm()**. If *errret* is not null, the integer it points to is set to one of the following values to report the function outcome:

−1	The **terminfo** database was not found (function fails).
0	The entry for the terminal was not found in **terminfo** (function fails).
1	Success.

If **setupterm()** detects an error and *errret* is a null pointer, **setupterm()** writes a diagnostic message and exits.

A simple call to **setupterm()** that uses all the defaults and sends the output to *stdout* is:

```
setupterm((char *)0, fileno(stdout), (int *)0);
```

The **set_curterm()** function sets the variable *cur_term* to *nterm*, and makes all of the **terminfo** boolean, numeric, and string variables use the values from *nterm*.

The **del_curterm()** function frees the space pointed to by *oterm* and makes it available for further use. If *oterm* is the same as *cur_term*, references to any of the **terminfo** boolean, numeric, and string variables thereafter may refer to invalid memory locations until **setupterm()** is called again.

The **restartterm()** function assumes a previous call to **setupterm()** (perhaps from **initscr()** or **newterm()**). It lets the application specify a different terminal type in *term* and updates the information returned by **baudrate()** based on *fildes*, but does not destroy other information created by **initscr()**, **newterm()** or **setupterm()**.

RETURN VALUE

Upon successful completion, **set_curterm()** returns the previous value of *cur_term*. Otherwise, it returns a null pointer.

Upon successful completion, the other functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

An application would call **setupterm()** if it required access to the **terminfo** database but did not otherwise need to use Curses.

SEE ALSO

Selecting a Terminal in terminfo(4), baudrate(3X), erasechar(3X), has_ic(3X), longname(3X), putc(3X), termattrs(3X), termname(3X), tgetent(3X), tigetflag(3X), use_env(3X), <term.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME
delay_output — delay output

SYNOPSIS
```
#include <curses.h>
int delay_output(int ms);
```

DESCRIPTION
On terminals that support pad characters, **delay_output()** pauses the output for at least *ms* milliseconds. Otherwise, the length of the delay is unspecified.

RETURN VALUE
Upon successful completion, **delay_output()** returns OK. Otherwise, it returns ERR.

ERRORS
No errors are defined.

APPLICATION USAGE
Whether or not the terminal supports pad characters, the **delay_output()** function is not a precise method of timekeeping.

SEE ALSO
Defined Capabilities in terminfo(4), napms(), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The entry is rewritten for clarity.

NAME
delch, mvdelch, mvwdelch, wdelch — delete a character from a window.

SYNOPSIS
```
#include <curses.h>
int delch(void);
int mvdelch(int y, int x);
int mvwdelch(WINDOW *win, int y, int x);
int wdelch(WINDOW *win);
```

DESCRIPTION
These functions delete the character at the current or specified position in the current or specified window. This function does not change the cursor position.

RETURN VALUE
Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS
No errors are defined.

SEE ALSO
<curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The entry is rewritten for clarity. The argument list for the **delch()** function is explicitly declared as **void**.

NAME
deleteln, wdeleteln — delete lines in a window

SYNOPSIS
```
#include <curses.h>
int deleteln(void);
int wdeleteln(WINDOW *win);
```

DESCRIPTION
The **deleteln()** and **wdeleteln()** functions delete the line containing the cursor in the current or specified window and move all lines following the current line one line toward the cursor. The last line of the window is cleared. The cursor position does not change.

RETURN VALUE
Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS
No errors are defined.

SEE ALSO
insdelln(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The entry is rewritten for clarity. The argument list for the **deleteln()** function is explicitly declared as **void**.

NAME
delscreen — free storage associated with a screen

SYNOPSIS
```
#include <curses.h>
void delscreen(SCREEN *sp);
```

DESCRIPTION
The **delscreen()** function frees storage associated with the **SCREEN** pointed to by *sp*.

RETURN VALUE
The **delscreen()** function does not return a value.

ERRORS
No errors are defined.

SEE ALSO
endwin(3X), initscr(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 4.

NAME
delwin — delete a window

SYNOPSIS
```
#include <curses.h>
int delwin(WINDOW *win);
```

DESCRIPTION
The **delwin()** function deletes *win*, freeing all memory associated with it. The application must delete subwindows before deleting the main window.

RETURN VALUE
Upon successful completion, **delwin()** returns OK. Otherwise, it returns ERR.

ERRORS
No errors are defined.

SEE ALSO
derwin(3X), dupwin(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The entry is rewritten for clarity.

NAME
derwin— relative window creation function

SYNOPSIS
```
#include <curses.h>

WINDOW *derwin(WINDOW *orig, int nlines, int ncols, int begin_y,
int begin_x);
```

DESCRIPTION
The **derwin()** function creates a new window with *nlines* lines and *ncols* columns, positioned so that the origin is at (*begin_y*, *begin_x*) relative to the origin of the window *orig*. If any part of the new window is outside *orig*, the function fails and the window is not created.

RETURN VALUE
Upon successful completion, these functions return a pointer to the new window. Otherwise, they return a null pointer.

ERRORS
No errors are defined.

APPLICATION USAGE
Before performing the first refresh of a subwindow, portable applications should call **touchwin()** or **touchline()** on the parent window.

Each window maintains internal descriptions of the screen image and status. The screen image is shared among all windows in the window hierarchy. Refresh operations rely on information on what has changed within a window, which is private to each window. Refreshing a window, when updates were made to a different window, may fail to perform needed updates because the windows do not share this information.

SEE ALSO
delwin(3X), newwin(3X), is_linetouched(3X), doupdate(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 4.

NAME

devnm - map device ID to file path

SYNOPSIS

```
#include <devnm.h>

int devnm (
   mode_t devtype,
   dev_t devid,
   char *path,
   size_t pathlen,
   int cache
);
```

DESCRIPTION

Given a device type, a device ID, and a string in which to return the result, **devnm()** maps the type and ID to a block or character special file (device file) name by searching **/dev**. It returns in *path* the full path name of the first special file encountered with a matching device type and ID. It searches **/dev** and all its subdirectories recursively in unspecified order.

The parameters are:

devtype One of the file type values **S_IFBLK** or **S_IFCHR** documented in *stat*(5). Bits other than those in the **S_IFMT** set are ignored. Hence the value can be, for example, an **st_mode** value returned by **stat()** (see *stat*(2)).

devid A device ID (the combined major and minor numbers) such as returned by **stat()** in the **st_dev** or **st_rdev** field.

path Pointer to the buffer in which to return the path name result.

pathlen Tells the available length of the *path* string, including the NUL terminator character. If *path* is too short to hold the full path name, only the first *pathlen*–1 characters are returned in a null-terminated string, and the return value is altered (see below).

cache A flag that tells **devnm()** whether to save file information in memory allocated by **malloc()**, and later, whether to use that saved information instead of searching **/dev** again. A subsequent call with *cache* non-zero can be much faster, especially if **/dev** is a large tree. However, the first call with *cache* true might be slower because **devnm()** must read all of the **/dev** tree once to create the cache, rather than returning immediately upon finding a matching file. Any call with *cache* set to zero ignores the cache, if any, and reads the directory.

To allow for possible future enhancements, *cache* should be restricted to the values 0 and 1.

There is no way to tell **devnm()** to free its cached memory.

devnm() ignores unreadable directories and files for which **stat()** fails.

APPLICATION USAGE

devnm() is thread-safe. It is not async-cancel-safe. A cancellation point may occur when a thread is executing **devnm()**.

RETURN VALUE

devnm() returns one of the following values:

0 Successful. The result is in *path*.

–1 **ftw()** failed. **errno** contains the value returned from **ftw()**. *path* might be altered if *cache* is set. If *cache* was set for the first time, **devnm()** freed any memory allocated by the current call.

–2 No matching special file was found. **errno** is undefined. *path* is unaltered.

–3 A matching special file was found, but the name was truncated to fit in *path*. **errno** is undefined.

If **malloc()** fails, **devnm()** silently abandons the attempt to do caching in the current or any later call with *cache* true, and frees any memory allocated by the current call.

AUTHOR

devnm() was created by HP.

SEE ALSO

devnm(1M), stat(2), ftw(3C), malloc(3C), ttyname(3C), stat(5).

NAME

dial(), undial() - establish an outgoing terminal line connection

SYNOPSIS

```
#include <dial.h>

int dial(CALL call);

void undial(int fd);
```

DESCRIPTION

The **dial()** function 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;
```

CALL elements are as follows:

speed 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 transmits at any rate between 0 and 300 bits per second. However, the high-speed setting of the 212A modem transmits and receives at 1200 bits per second only.

baud Desired transmission baud rate. For example, one might set *baud* to 110 and *speed* to 300 (or 1200). However, if *speed* is set to 1200, *baud* must be set to high (1200).

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

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

Permissible Codes	
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

modem Specifies modem control for direct lines. Set to non-zero if modem control is required.

attr Pointer to a **termio** structure, as defined in the <**termio.h**> header file. A NULL value for this pointer element can be passed to the **dial()** function, but if such a structure is included, the elements specified in it are set for the outgoing terminal line before the connection is established. This is often important for certain attributes such as parity and baud rate.

device Holds the device name that establishes the connection.

dev_len Length of the device name that is copied into the array device.

APPLICATION USAGE

dial() and **undial()** are thread-safe. These interfaces are not cancel-safe. A cancellation point may occur when a thread is executing **dial()** or **undial()**.

RETURN VALUE

On failure, a negative value indicating the reason for the failure is 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    /* automatic call unit (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 unexpected increases in the size of programs that otherwise do not use standard I/O.

The **dial()** function will modify the values of some of the fields of the the **CALL** structure so if **dial()**, is reinvoked, reinitialize the values of the **CALL** structure.

FILES

/etc/uucp/Devices

SEE ALSO

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

UUCP tutorial in *Remote Access User's Guide*.

NAME

opendir(), readdir(), readdir_r(), telldir(), seekdir(), rewinddir(), closedir() - directory operations

SYNOPSIS

```
#include <dirent.h>
DIR *opendir(const char *dirname);
struct dirent *readdir(DIR *dirp);
int readdir_r(DIR *dirp, struct dirent *entry, struct dirent **result);
long int telldir(DIR *dirp);
void seekdir(DIR *dirp, long int loc);
void rewinddir(DIR *dirp);
int closedir(DIR *dirp);
```

DESCRIPTION

This library package provides functions that allow programs to read directory entries without having to know the actual directory format associated with the file system. Because these functions allow programs to be used portably on file systems with different directory formats, this is the recommended way to read directory entries.

opendir() opens the directory *dirname* and associates a directory stream with it. **opendir()** returns a pointer used to identify the directory stream in subsequent operations. **opendir()** uses *malloc*(3C) to allocate memory.

readdir() returns a pointer to the next directory entry. It returns a NULL pointer upon reaching the end of the directory or detecting an invalid **seekdir()** operation. See *dirent*(5) for a description of the fields available in a directory entry.

readdir_r()
initializes the **dirent** structure refrenced by **entry** to represent the current position in the directory stream referenced by **dirp**, and stores a pointer to this structure at the location referenced by **result**.

telldir() returns the current location (encoded) associated with the directory stream to which *dirp* refers.

seekdir() sets the position of the next **readdir()** operation on the directory stream to which *dirp* refers. The *loc* argument is a location within the directory stream obtained from **telldir()**. The position of the directory stream is restored to where it was when **telldir()** returned that *loc* value. Values returned by **telldir()** are valid only while the **DIR** pointer from which they are derived remains open. If the directory stream is closed and then reopened, the **telldir()** value might be invalid.

rewinddir()
resets the position of the directory stream to which *dirp* refers to the beginning of the directory. It also causes the directory stream to refer to the current state of the corresponding directory, as a call to **opendir()** would have done.

closedir() closes the named directory stream, then frees the structure associated with the **DIR** pointer.

RETURN VALUE

opendir() upon successful completion, returns a pointer to an object of type **DIR** referring to an open directory stream. Otherwise, it returns a NULL pointer and sets the global variable **errno** to indicate the error.

readdir() upon successful completion, returns a pointer to an object of type **struct** dirent describing a directory entry. Upon reaching the end of the directory, **readdir()** returns a NULL pointer and does not change the value of **errno**. Otherwise, it returns a NULL pointer and sets **errno** to indicate the error.

readdir_r()
upon successful completion returns a 0. On successful return, the pointer returned at **result** has the same value as the argument **entry.** Upon reaching end of the directory

stream, **result** has the value **NULL.** An error number is returned upon error.

telldir() upon successful completion, returns a long value indicating the current position in the directory. Otherwise it returns **-1** and sets **errno** to indicate the error.

seekdir() does not return any value, but if an error is encountered, **errno** is set to indicate the error.

closedir() upon successful completion, returns a value of **0**. Otherwise, it returns a value of **-1** and sets **errno** to indicate the error.

ERRORS

opendir() fails if any of the following conditions are encountered:

[EACCES]	Search permission is denied for a component of *dirname*, or read permission is denied for *dirname*.
[EFAULT]	*dirname* points outside the allocated address space of the process. The reliable detection of this error is implementation dependent.
[ELOOP]	Too many symbolic links were encountered in translating the path name.
[EMFILE]	Too many open file descriptors are currently open for the calling process.
[ENAMETOOLONG]	A component of *dirname* exceeds **PATH_MAX** bytes, or the entire length of *dirname* exceeds **PATH_MAX** – 1 bytes while **_POSIX_NO_TRUNC** is in effect.
[ENFILE]	Too many open file descriptors are currently open on the system.
[ENOENT]	A component of *dirname* does not exist.
[ENOMEM]	**malloc()** failed to provide sufficient memory to process the directory.
[ENOTDIR]	A component of *dirname* is not a directory.
[ENOENT]	The *dirname* argument points to an empty string.

readdir() or **readdir_r()** might fail if any of the following conditions are encountered:

[EBADF]	*dirp* does not refer to an open directory stream.
[ENOENT]	The directory stream to which *dirp* refers is not located at a valid directory entry.
[EFAULT]	*dirp* points outside the allocated address space of the process.

telldir() might fail if any of the following conditions are encountered:

[EBADF]	*dirp* does not refer to an open directory stream.
[ENOENT]	*dirp* specifies an improper file system block size.

seekdir() might fail if the following condition is encountered:

[ENOENT]	*dirp* specifies an improper file system block size.

closedir() might fail if any of the following conditions are encountered:

[EBADF]	*dirp* does not refer to an open directory stream.
[EFAULT]	*dirp* points outside the allocated address space of the process.

rewinddir() might fail if any of the following conditions are encountered:

[EBADF]	*dirp* does not refer to an open directory stream.
[EFAULT]	*dirp* points outside the allocated address space of the process.

EXAMPLES

The following code searches the current directory for an entry *name*:

```
DIR *dirp;
struct dirent *dp;

dirp = opendir(".");
while ((dp = readdir(dirp)) != NULL) {
     if (strcmp(dp->d_name, name) == 0) {
```

```
            (void) closedir(dirp);
            return FOUND;
        }
    }
    (void) closedir(dirp);
    return NOT_FOUND;
```

WARNINGS

readdir() and **getdirentries()** (see *getdirentries*(2) are the only ways to access remote NFS directories. Attempting to read a remote directory via NFS by using **read()** returns **-1** and sets **errno** to EISDIR (see *read*(2)).

APPLICATION USAGE

The header file required for these functions and the type of the return value from **readdir()** has been changed for compatibility with System V Release 3 and the *X/Open Portability Guide*. See *ndir*(5) for a description of the header file <**ndir.h**>, which is provided to allow existing HP-UX applications to compile unmodified. <**ndir.h**> header file is obsoleted starting from HP-UX 10.30 and is available in **/usr/old/usr/include** directory.

New applications should use the <**dirent.h**> header file for portability to System V and X/Open systems. The <**ndir.h**> header file is obsoleted starting from HP-UX 10.30 and is going to be removed in future releases.

readdir() is unsafe in multithread applications. **closedir()**, **opendir()**, **readdir()**, **rewinddir()**, **seekdir()** and **readdir_r()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing **closedir()**, **opendir()**, **readdir()**, **rewinddir()**, **seekdir()** or **readdir_r()**.

Users of **readdir_r()** should note that readdir_r() now conforms with the POSIX.1c Threads standard. The old prototype of **readdir_r()** is supported for compatibility with existing DCE applications only.

AUTHOR

directory was developed by AT&T, HP, and the University of California, Berkeley.

SEE ALSO

close(2), getdirentries(2), lseek(2), open(2), read(2), dir(4), dirent(5), ndir(5).

STANDARDS CONFORMANCE

closedir(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

opendir(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

readdir(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

readdir_r(): POSIX.1c

rewinddir(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

seekdir(): AES, SVID3, XPG2, XPG3, XPG4

telldir(): AES, XPG2, XPG3, XPG4

NAME

div(), ldiv() - integer division and remainder

SYNOPSIS

```
#include <stdlib.h>
div_t div(int numer, int denom);
ldiv_t ldiv(long int numer, long int denom);
```

DESCRIPTION

div() Computes the quotient and remainder of the division of the numerator *numer* by the denominator *denom*. If the division is inexact, the sign of the resulting quotient is that of the algebraic quotient, and the magnitude of the resulting quotient is the largest integer less than the magnitude of the algebraic quotient. If the result can be represented, the result is returned in a structure of type **div_t** (defined in <**stdlib.h**>) having members *quot* and *rem* for the quotient and remainder respectively. Both members have type **int** and values such that *quot* × *denom* + *rem* = *numer*. If the result cannot be represented, the behavior is undefined.

ldiv() Similar to **div()**, except that the arguments each have type **long int** and the result is returned in a structure of type **ldiv_t** (defined in <**stdlib.h**>) having **long int** members *quot* and *rem* for the quotient and remainder respectively.

APPLICATION USAGE

div() and **ldiv()** are thread-safe and async-cancel-safe.

WARNINGS

Behavior is undefined if *denom* is zero.

SEE ALSO

floor(3M).

STANDARDS CONFORMANCE

div(): AES, SVID3, XPG4, ANSI C

ldiv(): AES, SVID3, XPG4, ANSI C

NAME

dlclose - close a shared object

SYNOPSIS

cc [*flag* ...] *file* ... **-ldl** [*library*] ...

```
#include <dlfcn.h>
int dlclose(void *handle);
```

DESCRIPTION

dlclose disassociates a shared object previously opened by **dlopen** from the current process. Once an object has been closed using **dlclose**, its symbols are no longer available to **dlsym**. All objects loaded automatically as a result of invoking **dlopen** on the referenced object (see *dlopen*(3C)) are also closed. *handle* is the value returned by a previous invocation of **dlopen**.

MULTITHREAD USAGE

This routine is thread-safe.

RETURN VALUE

If the referenced object was successfully closed, **dlclose** returns 0. If the object could not be closed, or if handle does not refer to an open object, **dlclose** returns a non-0 value. More detailed diagnostic information is available through **dlerror**.

WARNINGS

A successful invocation of **dlclose** does not guarantee that the objects associated with *handle* have actually been removed from the address space of the process. Objects loaded by one invocation of **dlopen** may also be loaded by another invocation of **dlopen**. The same object may also be opened multiple times. An object is not removed from the address space until all references to that object through an explicit **dlopen** invocation have been closed and all other objects implicitly referencing that object have also been closed.

Once an object has been closed by **dlclose**, referencing symbols contained in that object can cause undefined behavior.

SEE ALSO

dlerror(3C), dlopen"(3C), dlsym(3C).

NAME

dlerror - get diagnostic information

SYNOPSIS

cc [*flag* ...] *file* ... **-ldl** [*library*] ...

```
#include <dlfcn.h>
char *dlerror(void);
```

DESCRIPTION

dlerror returns a null-terminated character string (with no trailing newline) that describes the last error that occurred during dynamic linking processing. If no dynamic linking errors have occurred since the last invocation of **dlerror**, **dlerror** returns NULL. Thus, invoking **dlerror** a second time, immediately following a prior invocation, results in NULL being returned.

MULTITHREAD USAGE

This routine is thread-safe.

WARNINGS

The messages returned by **dlerror** may reside in a static buffer that is overwritten on each call to **dlerror**. Application code should not write to this buffer. Programs wishing to preserve an error message should make their own copies of that message.

SEE ALSO

dlclose(3C), dlopen(3C), dlsym(3C).

NAME

dlget - retrieve information on a loaded module (program or shared library)

SYNOPSIS

cc [*flag* ...] *file* ... **-ldl** [*library*] ...

```
#include <dlfcn.h>
void *dlget(unsigned int index,
            struct load_module_desc *desc,
            size_t desc_size);
```

DESCRIPTION

dlget is one of a family of routines that give the user direct access to the dynamic linking facilities. **dlget** returns information about a loaded module for a process. *index* specifies the requested shared library by its placement on the dynamic loader's search list. An *index* of zero requests information about the program file itself. An *index* of -1 requests info about the dynamic loader.

If successful, **dlget** returns a handle for the shared library as defined by the return value from **dlopen()**.

desc must be preallocated by the user. The structure members are filled in by the dynamic loader with information about the requested shared library.

A *load_module_desc* structure has the following members:

```
struct load_module_desc {
   unsigned long text_base;
   unsigned long text_size;
   unsigned long data_base;
   unsigned long data_size;
   unsigned long unwind_base;
   unsigned long linkage_ptr;
   unsigned long phdr_base;
   unsigned long tls_size;
   unsigned long tls_start_addr;
   }
```

desc_size specifies the size in bytes of the *load_module_desc* structure sent in by the user.

If a call to **dlget** is unsuccessful, a NULL pointer is returned and *desc* remains unchanged.

MULTITHREAD USAGE

This routine is thread-safe.

AUTHOR

dlget() was developed by HP.

SEE ALSO

System Tools:

cc(1)	Invokes the HP-UX C compiler.
exec(2)	System loader.
ld(1)	Invokes the link editor.

Miscellaneous:

a.out(4)	Assembler, compiler, and linker output.
dlclose(3C)	Unloads a shared library previously loaded by **dlopen()**.
dlerror(3C)	Prints the last error message recorded by **dld**.
dlgetname(3C)	Returns the name of the storage containing a load module.
dlmodinfo(3C)	Returns information about a loaded module.
dlopen(3C)	Loads a shared library.
dlsym(3C)	Gets the address of a symbol in a shared library.

NAME

dlgetname - retrieve the name of a dll given a load module descriptor

SYNOPSIS

cc [*flag* ...] *file* ... **-ldl** [*library*] ...

```
#include <dlfcn.h>
char *dlgetname(struct load_module_desc *desc,
            size_t desc_size,
            void *(*read_tgt_mem)(void* buffer,
                                  unsigned long long ptr,
                                  size_t bufsiz,
                                  int ident),
            int ident_parm,
            unsigned long long load_map_parm);
```

DESCRIPTION

dlgetname is one of a family of routines that give the user direct access to the dynamic linking facilities. **dlgetname** returns the pathname of a load module represented by *desc*. The *read_tgt_mem*, *ident_parm*, and *load_map_parm* parameters are identical to those for **dlmodinfo**.

The caller of **dlgetname** must copy the return value to insure that it is not corrupted. If *desc* does not describe a loaded module, then NULL is returned.

MULTITHREAD USAGE

This routine is thread-safe.

AUTHOR

dlgetname() was developed by HP.

SEE ALSO

System Tools:

cc(1)	Invokes the HP-UX C compiler.
exec(2)	System loader.
ld(1)	Invokes the link editor.

Miscellaneous:

a.out(4)	Assembler, compiler, and linker output.
dlclose(3C)	Unloads a shared library previously loaded by **dlopen()**.
dlerror(3C)	Prints the last error message recorded by **dld**.
dlget(3C)	Returns information on a loaded module.
dlmodinfo(3C)	Returns information about a loaded module.
dlopen(3C)	Loads a shared library.
dlsym(3C)	Gets the address of a symbol in a shared library.

NAME

dlmodinfo - retrieve information on a loaded module (program or shared library)

SYNOPSIS

cc [*flag* ...] *file* ... **-ldl** [*library*] ...

```
#include <dlfcn.h>
unsigned long dlmodinfo(unsigned long ip_value,
                  struct load_module_desc *desc,
                  size_t desc_size,
                  void *(*read_tgt_mem)(void* buffer,
                                  unsigned long ptr,
                                  size_t bufsiz,
                                  int ident),
                  int ident_parm,
                  uint64_t load_map_parm);
```

DESCRIPTION

dlmodinfo is one of a family of routines that give the user direct access to the dynamic linking facilities. **dlmodinfo** returns information about a loaded module for a process. *ip_value* is the instruction pointer value of the requested library. If the value is NULL, then *desc* will contain the module info of dld itself. *desc* is a buffer of memory allocated by the user program. The dynamic loader will fill this in with module information. *desc_size* is the size in bytes of the *desc* buffer. *read_tgt_mem* is a pointer to a function used by **dlmodinfo** to retrieve needed information. If the value is NULL, the dynamic loader will use its own internal data structures to find the correct load module and the following two parameters are ignored.

ident_parm Is only used to pass the fourth parameter to *read_tgt_mem*.

load_map_parm
Is only used when calling through *read_tgt_mem*. It contains the starting address of the load map.

Otherwise, the function pointer will be used to read memory during its search, using these parameters:

buffer a buffer supplied by **dlmodinfo** to read into

ptr the virtual memory address to read from

bufsiz the size of buffer in bytes

ident the value of the ident_parm parameter to **dlmodinfo**

On success, *read_tgt_mem* will return the value of its buffer parameter, otherwise, it will return NULL. *read_tgt_mem* allows **dlmodinfo** to find a load module in one process on behalf of another. The calling process passes a callback via *read_tgt_mem* in order to read memory in a different process address space from the one in which **dlmodinfo** resides. *ip_value*, *load_map_parm*, and *ptr* from *read_tgt_mem* can be pointers to objects in another process. For example, when a 32 bit program wants to enquire about a 64 bit program, the *ip_value* and *load_map_parm* should be 64bit values. Any 32 bit pointers should be type casted to 64 bits by the user program when passed in the *ip_value* or *load_map_parm* parameters.

If the calling process calls **dlmodinfo** with a callback registered via *read_tgt_mem*, it must supply the starting address of the target process' load map in the *load_map_parm* parameter to **dlmodinfo**. This can be retrieved by using the **DT_HP_LOAD_MAP** dynamic table entry in the target program file.

A cross-process load module operation can be done via **dlmodinfo**, for example, by issuing a call to ttrace().

RETURN VALUE

If successful, **dlmodinfo** returns a handle for the shared library as defined by the return value from **dlopen()**. NULL is returned otherwise.

MULTITHREAD USAGE

Thread safe in **libdl.sl** but not in **libxpdl.sl**.

AUTHOR

dlmodinfo was developed by HP.

SEE ALSO

System Tools:

cc(1)	Invokes the HP-UX C compiler.
exec(2)	System loader.
ld(1)	Invokes the link editor.

Miscellaneous:

a.out(4)	Assembler, compiler, and linker output.
dlclose(3C)	Unloads a shared library previously loaded by **dlopen()**.
dlerror(3C)	Prints the last error message recorded by **dld**.
dlget(3C)	Returns information about a loaded module.
dlgetname(3C)	Returns the name of the storage containing a load module.
dlopen(3C)	Loads a shared library.
dlsym(3C)	Gets the address of a symbol in a shared library.

NAME

dlopen - open a shared object

SYNOPSIS

cc [*flag* ...] *file* ... **-ldl** [*library*] ...

```
#include <dlfcn.h>
void *dlopen(const char *file, int mode);
```

DESCRIPTION

dlopen is one of a family of routines that give the user direct access to the dynamic linking facilities. **dlopen** makes a shared object specified by a *file* available to a running process. A shared object may specify other objects that it "needs" in order to execute properly. These dependencies are specified by **DT_NEEDED** entries int the **.dynamic** section of the original object. Each needed object may, in turn, specify other needed objects. All such objects are loaded along with the original object as a result of the call to **dlopen**.

A successful **dlopen** call returns to the process a *handle* which the process may use on subsequent calls to **dlsym** and **dlclose.** This value should not be interpreted in any way by the process.

file is used to construct a pathname to the object file. If *file* contains a slash character, the *file* argument itself is used as the pathname. Otherwise a series of directories is searched for *file.* First, any directories specified by a **DT_RPATH** entry in the **.dynamic** section of the original program object are searched. Then, any directories specified by the environment variable **LD_LIBRARY_PATH** are searched. Finally, the directory **/usr/lib/pa20_64** and **usr/ccs/lib/pa20_64** are searched.

If the value of *file* is 0, **dlopen** provides a *handle* on a "global symbol object." This object provides access to the symbols from an ordered set of objects consisting of the original **a.out**, all of the objects that were loaded at program startup along with the **a.out**, and all objects loaded using a **dlopen** operation along with the **RTLD_GLOBAL** flag. As the latter set of objects can change during execution, the set identified by *handle* can also change dynamically.

Only a single copy of an object file is brought into the address space, even if **dlopen** is invoked multiple times in reference to the file, and even if different pathnames are used to reference the file.

When a shared object is brought into the address space of a process, it may contain references to symbols whose addresses are not known until the object is loaded. These references must be relocated before the symbols can be accessed. The *mode* parameter governs when these relocations take place and may have the following values:

RTLD_LAZY Under this *mode*, only references to data symbols are relocated when the object is loaded. References to functions are not relocated until a given function is invoked for the first time. This *mode* should result in better performance, since a process may not reference all of the functions in any given shared object.

RTLD_NOW Under this *mode*, all necessary relocations are performed when the object is first loaded. This may result in some wasted effort, if relocations are performed for functions that are never referenced, but is useful for applications that need to know as soon as an object is loaded that all symbols referenced during execution will be available.

Any object loaded by **dlopen** that requires relocations against global symbols can reference the symbols in the original **a.out**, any objects loaded at program startup, from the object itself as well as any other object included in the same **dlopen** invocation, and any objects that were loaded in any **dlopen** invocation that specified the **RTLD_GLOBAL** flag. To determine the scope of visibility for the symbols loaded with a **dlopen** invocation, the *mode* parameter should be bitwise or'ed with one of the following values:

RTLD_GLOBAL
The object's symbols are made available for the relocation processing of any other object. In addition, symbol lookup using **dlopen**(0, *mode*) and an associated **dlsym()** allows objects loaded with **RTLD_GLOBAL** to be searched.

RTLD_LOCAL The object's symbols are made available for relocation processing only to objects loaded in the same **dlopen** invocation.

If neither **RTLD_GLOBAL** nor **RTLD_LOCAL** are specified, the default is **RTLD_LOCAL**.

If a *file* is specified in multiple **dlopen** invocations, *mode* is interpreted at each invocation. Note, however, that once **RTLD_NOW** has been specified, all relocations will have been completed, rendering any further **RTLD_NOW** operations redundant and any further **RTLD_LAZY** operations irrelevant. Similarly note that

once **RTLD_GLOBAL** has been specified, the object will maintain the **RTLD_GLOBAL** status regardless of any previous or future specification of **RTLD_LOCAL**, so long as the object remains in the address space (see *dlclose*(3C)).

Symbols introduced into a program through calls to **dlopen** may be used in relocation activities. Symbols so introduced may duplicate symbols already defined by the program or previous **dlopen** operations. To resolve the ambiguities such a situation might present, the resolution of a symbol reference to a symbol definition is based on a symbol resolution order. Two such resolution orders are defined: *load* and *dependency* ordering. Load order establishes an ordering among symbol definitions using the temporal order in which the objects containing the definitions were loaded, such that the definition first loaded has priority over definitions added later. Load ordering is used in relocation processing. Dependency ordering uses a "breadth-first" order starting with a given object, then all of its dependencies, then any dependents of those, iterating until all dependencies are satisfied. With the exception of the global symbol object obtained via a **dlopen** operation on a *file* with a value 0, dependency ordering is used by the **dlsym** function. Load ordering is used in **dlsym** operations on the global symbol object.

When an object is first made accessible via **dlopen**, it and its dependent objects are added in dependency order. Once all objects are added, relocations are performed using load order. Note that if an object and its dependencies have been loaded by a previous **dlopen** invocation or on startup, the load and dependency order may yield different resolutions.

The symbols introduced by **dlopen** operations and available through **dlsym** are those which are "exported" as symbols of global scope by the object. For shared objects, such symbols will typically be those that were specified in (for example) C source code as having *extern* linkage. For **a.out**'s, only a subset of externally visible symbols are typically exported: specifically those referenced by the shared objects with which the **a.out** is linked. The exact set of exported symbols for any shared object or the **a.out** can be controlled using the linker (see *ld*(1)).

MULTITHREAD USAGE

This routine is thread-safe.

RETURN VALUE

If *file* cannot be found, cannot be opened for reading, is not a shared object, or if an error occurs during the process of loading *file* or relocating its symbolic references, **dlopen** returns NULL. More detailed diagnostic information is available through **dlerror**.

WARNINGS

The environment variable **LD_LIBRARY_PATH** should contain a colon-separated list of directories, in the same format as the **PATH** variable (see *sh*(1)). **LD_LIBRARY_PATH** will be ignored if the process' real user id is different from its effective user id or its real group id is different from its effective group id (see *exec*(2)) or if the process has acquired any privileges (see *tfadmin*(1M)).

SEE ALSO

cc(1), ld(1), sh(1), exec(2), dlclose(3C), dlerror(3C), dlsym(3C).

NAME

dlsym - get the address of a symbol in shared object

SYNOPSIS

cc [*flag* ...] *file* ... **-ldl** [*library*] ...

```
#include <dlfcn.h>
void *dlsym(void *handle, const char *name);
```

DESCRIPTION

dlsym allows a process to obtain the address of a symbol defined within a shared object previously opened by **dlopen**. *handle* is a either the value returned by a call to **dlopen** or is the special flag **RTLD_NEXT**. In the former case, the corresponding shared object must not have been closed using **dlclose**. *name* is the symbol's name as a character string.

dlsym searches for the named symbol in all shared objects loaded automatically as a result of loading the object referenced by *handle* (see *dlopen*(3C)).

If *handle* is **RTLD_NEXT**, the search begins with the "next" object after the object from which **dlsym** was invoked. Objects are searched using a *load* order symbol resolution algorithm (see *dlopen*(3C)). The "next" object, and all other objects searched, are either of global scope (because they were loaded at startup or as part of a **dlopen** operation with the **RTLD_GLOBAL** flag) or are objects loaded by the same **dlopen** operation that loaded the caller of **dlsym**.

MULTITHREAD USAGE

This routine is thread-safe.

RETURN VALUE

If *handle* does not refer to a valid object opened by **dlopen**, or if the named symbol cannot be found within any of the objects associated with *handle*, **dlsym** will return NULL. More detailed diagnostic information will be available through **dlerror**.

APPLICATION USAGE

RTLD_NEXT can be used to navigate an intentionally created hierarchy of multiply defined symbols created through *interposition*. For example, if a program wished to create an implementation of **malloc** that embedded some statistics gathering about memory allocations, such an implementation could define its own **malloc** which would gather the necessary information, and use **dlsym** with **RTLD_NEXT** to find the "real" **malloc**, which would perform the actual memory allocation. Of course, this "real" **malloc** could be another user-defined interface that added its own value and then used **RTLD_NEXT** to find the system **malloc**.

EXAMPLES

The following example shows how one can use **dlopen** and **dlsym** to access either function or data objects. For simplicity, error checking has been omitted.

```
void *handle;
int i, *iptr;
int (*fptr)(int);

/* open the needed object */
handle = dlopen("/usr/mydir/mylib.so", RTLD_LAZY);

/* find address of function and data objects */
fptr = (int (*)(int))dlsym(handle, "some_function");

iptr = (int *)dlsym(handle, "int_object");

/* invoke function, passing value of integer as a parameter */

i = (*fptr)(*iptr);
```

The next example shows how one can use dlsym with RTLD_NEXT to add functionality to an existing interface. Again, error checking has been omitted.

```
extern void record_malloc(void *, size_t);
```

```
void *
malloc(size_t sz)
{
      void *ptr;
      void *(*real_malloc)(size_t);

      real_malloc = (void * (*) (size_t))
                     dlsym(RTLD_NEXT, "malloc");
      ptr = (*real_malloc)(sz);
      record_malloc(ptr, sz);
      return ptr;
}
```

SEE ALSO

dlclose(3C), dlerror(3C), dlopen(3C).

NAME

doupdate, refresh, wnoutrefresh, wrefresh — refresh windows and lines

SYNOPSIS

```
#include <curses.h>
int doupdate(void);
int refresh(void);
int wnoutrefresh(WINDOW *win);
int wrefresh(WINDOW *win);
```

DESCRIPTION

The **refresh()** and **wrefresh()** functions refresh the current or specified window. The functions position the terminal's cursor at the cursor position of the window, except that if the **leaveok()** mode has been enabled, they may leave the cursor at an arbitrary position.

The **wnoutrefresh()** function determines which parts of the terminal may need updating. The **doupdate()** function sends to the terminal the commands to perform any required changes.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise they return ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

Refreshing an entire window is typically more efficient than refreshing several subwindows separately. An efficient sequence is to call **wnoutrefresh()** on each subwindow that has changed, followed by a call to **doupdate()**, which updates the terminal.

The **refresh()** or **wrefresh()** function (or **wnoutrefresh()** followed by **doupdate()**) must be called to send output to the terminal, as other Curses functions merely manipulate data structures.

SEE ALSO

clearok(3X), redrawwin(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

This entry is a merger of the X/Open Curses, Issue 3 entries **refresh()** and **wnoutrefresh()**. The *DESCRIPTION* is rewritten for clarity and the argument list for the **doupdate()** and **refresh()** functions is explicitly declared as **void**. Otherwise the functionality is identical to that defined in X/Open Curses, Issue 3.

NAME

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

SYNOPSIS

```
#include <stdlib.h>
double drand48(void);
double erand48(unsigned short int xsubi[3]);
long int lrand48(void);
long int nrand48(unsigned short int xsubi[3]);
long int mrand48(void);
long int jrand48(unsigned short int xsubi[3]);
void srand48(long int seedval);
unsigned short int *seed48(unsigned short int seed16v[3]);
void lcong48(unsigned short int param[7]);
```

Obsolescent Interfaces

```
int drand48_r(struct drand48_data *dp, double *randval);
int erand48_r(
   unsigned short int xsubi[3],
   struct drand48_data *dp,
   double *randval);
int lrand48_r(struct drand48_data *dp, long int *randval);
int nrand48_r(
   unsigned short int xsubi[3],
   struct drand48_data *dp,
   long int *randval);
int mrand48_r(struct drand48_data *dp, long int *randval);
int jrand48_r(
   unsigned short int xsubi[3],
   struct drand48_data *dp,
   long int *randval);
int srand48_r(long int seedval, struct drand48_data *dp);
int seed48_r(unsigned short int seed16v[3], struct drand48_data *dp);
int lcong48_r(unsigned short int param[7], struct drand48_data *dp);
```

DESCRIPTION

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

In the following description, the formal mathematical notation [*low*,*high*) indicates an interval including *low* but not including *high*.

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

lrand48() and **nrand48()** return nonnegative long integers uniformly distributed over the interval [0,2^31).

mrand48() and **jrand48()** return signed long integers uniformly distributed over the interval [–2^31,2^31).

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 are supplied automatically if **drand48()**, **lrand48()**, or **mrand48()** is called without a prior call to an initialization entry point.) **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] = (*a***X*[*n*] + *c*) modulo *m* *n*>=0

The parameter *m* = 2^48; hence 48-bit integer arithmetic is performed.

Unless **lcong48()** has been invoked, the default multiplier value *a* and the default addend value *c* are given by

a = 0x5DEECE66D (base 16) = 0273673163155 (base 8)
c = 0xB (base 16) = 013 (base 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, **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 do *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 0x330E (base 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 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()** restores the default multiplier and addend values for *a* and *c*, specified above.

Obsolescent Interfaces

drand48_r(), **erand48_r()**, **lrand48_r()**, **nrand48_r()**, **mrand48_r()**, **jrand48_r()**, **srand48_r()**, **seed48_r()**, **lcong48_r()** generate uniformly distributed pseudo-random numbers.

APPLICATION USAGE

drand48(), **erand48()**, **lrand48()**, **nrand48()**, **mrand48()**, **jrand48()**, **srand48()**, **seed48()** and **lcong48()** are thread-safe. These interfaces are not async-cancel-safe.

WARNINGS

drand48_r(), **erand48_r()**, **lrand48_r()**, **nrand48_r()**, **mrand48_r()**, **jrand48_r()**, **srand48_r()**, **seed48_r()** and **lcong48_r()** are obsolescent interfaces supported only for compatibility with existing DCE applications. New multithreaded applications should use **drand48()**, **erand48()**, **lrand48()**, **nrand48()**, **mrand48()**, **jrand48()**, **srand48()**, **seed48()**, and **lcong48()**.

SEE ALSO

rand(3C), random(3M).

STANDARDS CONFORMANCE

drand48(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

erand48(): AES, SVID2, SVID3, XPG2, XPG3, XPG4
jrand48(): AES, SVID2, SVID3, XPG2, XPG3, XPG4
lcong48(): AES, SVID2, SVID3, XPG2, XPG3, XPG4
lrand48(): AES, SVID2, SVID3, XPG2, XPG3, XPG4
mrand48(): AES, SVID2, SVID3, XPG2, XPG3, XPG4
nrand48(): AES, SVID2, SVID3, XPG2, XPG3, XPG4
seed48(): AES, SVID2, SVID3, XPG2, XPG3, XPG4
srand48(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

(ENHANCED CURSES)

NAME

dupwin — duplicate a window

SYNOPSIS

```
#include <curses.h>
WINDOW *dupwin(WINDOW *win);
```

DESCRIPTION

The **dupwin()** function creates a duplicate of the window *win*.

RETURN VALUE

Upon successful completion, **dupwin()** returns a pointer to the new window. Otherwise, it returns a null pointer.

ERRORS

No errors are defined.

SEE ALSO

derwin(3X), doupdate(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

echo, noecho — enable/disable terminal echo

SYNOPSIS

```
#include <curses.h>
int echo(void);
int noecho(void);
```

DESCRIPTION

The **echo()** function enables Echo mode for the current screen. The **noecho()** function disables Echo mode for the current screen. Initially, curses software echo mode is enabled and hardware echo mode of the tty driver is disabled.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

SEE ALSO

Input Processing in curses_intro(3X), getch(3X), <curses.h>, *X/Open System Interface Definitions, Issue 4, Version 2* specification, Section 9.2, *Parameters That Can Be Set*.

CHANGE HISTORY

First released in Issue .

X/Open Curses, Issue 4

The entry is rewritten for clarity. The argument list for the **echo()** and **noecho()** functions is explicitly declared as **void**.

NAME
echo_wchar, wecho_wchar — write a complex character and immediately refresh the window

SYNOPSIS
```
#include <curses.h>
int echo_wchar(const cchar_t *wch);
int wecho_wchar(WINDOW *win, const cchar_t *wch);
```

DESCRIPTION
The **echo_wchar()** function is equivalent to calling **add_wch()** and then calling **refresh()**.

The **wecho_wchar()** function is equivalent to calling **wadd_wch()** and then calling **wrefresh()**.

RETURN VALUE
Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS
No errors are defined.

SEE ALSO
addch(3X), add_wch(3X), doupdate(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 4.

(ENHANCED CURSES)

NAME

echochar, wechochar — echo single-byte character and rendition to a window and refresh

SYNOPSIS

```
#include <curses.h>
int echochar(const chtype ch);
int wechochar(WINDOW *win, const chtype ch);
```

DESCRIPTION

The **echochar()** function is equivalent to a call to **addch()** followed by a call to **refresh()**.

The **wechochar()** function is equivalent to a call to **waddch()** followed by a call to **wrefresh()**.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise they return ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the A_ prefix.

SEE ALSO

addch(3X), doupdate(3X), echo_wchar(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

ecvt(), fcvt(), gcvt() - convert floating-point number to string

SYNOPSIS

```
#include <stdlib.h>
char *ecvt(double value, int ndigit, int *decpt, int *sign);
char *fcvt(double value, int ndigit, int *decpt, int *sign);
char *gcvt(double value, int ndigit, char *buf);
```

Obsolescent Interfaces

```
int ecvt_r(
   double value,
   int ndigit,
   int *decpt,
   int *sign,
   char *buffer,
   int buflen);
int fcvt_r(
   double value,
   int ndigit,
   int *decpt,
   int *sign,
   char *buffer,
   int buflen);
```

DESCRIPTION

ecvt() Converts *value* to a null-terminated string of *ndigit* digits and returns a pointer to the string. The high-order digit is non-zero, unless the value is zero. The low-order digit is rounded. The position of the radix character relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). The radix character 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.

One of three non-digit characters strings could be returned if the converted value is out of range. A **- -** or **++** is returned if the value is larger than the exponent can contain, and is negative, or positive, respectively. The third string is returned if the number is illegal, a zero divide for example. The result value is Not A Number (NAN) and would return a **?** character.

fcvt() 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 produces *ndigit* significant digits in FORTRAN F-format if possible, or E-format otherwise. A minus sign, if required, and a radix character is included in the returned string. Trailing zeros are suppressed. The radix character is determined by the currently loaded NLS environment (see *setlocale*(3C)). If **setlocale()** has not been called successfully, the default NLS environment, "C", is used (see *lang*(5)). The default environment specifies a period (**.**) as the radix character.

Obsolescent Interfaces

ecvt_r() and **fcvt_r()** convert floating-point number to string.

APPLICATION USAGE

ecvt(), **gcvt()** and **fcvt()** are thread-safe. These interfaces are not async-cancel-safe.

EXTERNAL INFLUENCES

Locale

The **LC_NUMERIC** category determines the value of the radix character within the current NLS environment.

WARNINGS

The values returned by **ecvt()** and **fcvt()** point to an array whose content is overwritten by subsequent calls to these interfaces by the same thread.

ecvt_r() and **fcvt_r()** are obsolescent interfaces supported only for compatibility with existing DCE applications. New multi-threaded applications should use **ecvt()** and **fcvt()**.

AUTHOR

ecvt() and **fcvt()** were developed by AT&T. **gcvt()** was developed by AT&T and HP.

SEE ALSO

setlocale(3C), printf(3S), hpnls(5), lang(5).

STANDARDS CONFORMANCE

ecvt(): XPG2

fcvt(): XPG2

gcvt(): XPG2

NAME

elf - object file access library

SYNOPSIS

cc [*flag*...] *file*... **-lelf** [*library*] ...

```
#include <libelf.h>
```

DESCRIPTION

Functions in the ELF access library let a program manipulate ELF (Executable and Linking Format) object files, archive files, and archive members. The header file provides type and function declarations for all library services.

Programs communicate with many of the higher-level routines using an **ELF descriptor**. That is, when the program starts working with a file, **elf_begin** creates an ELF descriptor through which the program manipulates the structures and information in the file. These ELF descriptors can be used both to read and to write files. After the program establishes an ELF descriptor for a file, it may then obtain *section descriptors* to manipulate the sections of the file (see *elf_getscn*(3E)). Sections hold the bulk of an object file's real information, such as text, data, the symbol table, and so on. A section descriptor "belongs" to a particular ELF descriptor, just as a section belongs to a file. Finally, *data descriptors* are available through section descriptors, allowing the program to manipulate the information associated with a section. A data descriptor "belongs" to a section descriptor.

Descriptors provide private handles to a file and its pieces. In other words, a data descriptor is associated with one section descriptor, which is associated with one ELF descriptor, which is associated with one file. Although descriptors are private, they give access to data that may be shared. Consider programs that combine input files, using incoming data to create or update another file. Such a program might get data descriptors for an input and an output section. It then could update the output descriptor to reuse the input descriptor's data. That is, the descriptors are distinct, but they could share the associated data bytes. This sharing avoids the space overhead for duplicate buffers and the performance overhead for copying data unnecessarily.

File Classes

ELF provides a framework in which to define a family of object files, supporting multiple processors and architectures. An important distinction among object files is the *class* , or capacity, of the file. The 32-bit class supports architectures in which a 32-bit object can represent addresses, file sizes, and so forth, as in the following.

Name	Purpose
Elf32_Addr	Unsigned address
Elf32_Half	Unsigned medium integer
Elf32_Off	Unsigned file offset
Elf32_Sword	Signed large integer
Elf32_Word	Unsigned large integer
unsigned char	Unsigned small integer

Other classes will be defined as necessary, to support larger (or smaller) machines. Some library services deal only with data objects for a specific class, while others are class- independent. To make this distinction clear, library function names reflect their status, as described below.

Data Representations

Conceptually, two parallel sets of objects support cross compilation environments. One set corresponds to file contents, while the other set corresponds to the native memory image of the program manipulating the file. Type definitions supplied by the header files work on the native machine, which may have different data encodings (size, byte order, and so forth) than the target machine. Although native memory objects should be at least as big as the file objects (to avoid information loss), they may be bigger if that is more natural for the host machine.

Translation facilities exist to convert between file and memory representations. Some library routines convert data automatically, while others leave conversion as the program's responsibility. Either way, programs that create object files must write file-typed objects to those files; programs that read object files must take a similar view. See *elf_xlate*(3E) and *elf_fsize*(3E) for more information.

Programs may translate data explicitly, taking full control over the object file layout and semantics. If the program prefers not to have and exercise complete control, the library provides a higher-level interface that hides many object file details. elf_begin and related functions let a program deal with the native memory types, converting between memory objects and their file equivalents automatically when reading or writing

an object file.

ELF Versions

Object file versions allow ELF to adapt to new requirements. Three-independent-versions can be important to a program. First, an application program knows about a particular version by virtue of being compiled with certain header files. Second, the access library similarly is compiled with header files that control what versions it understands. Third, an ELF object file holds a value identifying its version, determined by the ELF version known by the file's creator. Ideally, all three versions would be the same, but they may differ.

If a program's version is newer than the access library, the program might use information unknown to the library. Translation routines might not work properly, leading to undefined behavior. This condition merits installing a new library.

The library's version might be newer than the program's and the file's. The library understands old versions, thus avoiding compatibility problems in this case.

Finally, a file's version might be newer than either the program or the library understands. The program might or might not be able to process the file properly, depending on whether the file has extra information and whether that information can be safely ignored. Again, the safe alternative is to install a new library that understands the file's version.

To accommodate these differences, a program must use **elf_version** to pass its version to the library, thus establishing the **working version** for the process. Using this, the library accepts data from and presents data to the program in the proper representations. When the library reads object files, it uses each file's version to interpret the data. When writing files or converting memory types to the file equivalents, the library uses the program's working version for the file data.

System Services

As mentioned above, **elf_begin** and related routines provide a higher-level interface to ELF files, performing input and output on behalf of the application program. These routines assume a program can hold entire files in memory, without explicitly using temporary files. When reading a file, the library routines bring the data into memory and perform subsequent operations on the memory copy. Programs that read or write large object files with this model must execute on a machine with a large process virtual address space. If the underlying operating system limits the number of open files, a program can use **elf_cntl** to retrieve all necessary data from the file, allowing the program to close the file descriptor and reuse it.

Although the **elf_begin** interfaces are convenient and efficient for many programs, they might be inappropriate for some. In those cases, an application may invoke the **elf_xlate** data translation routines directly. These routines perform no input or output, leaving that as the application's responsibility. By assuming a larger share of the job, an application controls its input and output model.

Library Names

Names associated with the library take several forms.

elf_name These class-independent names perform some service, *name*, for the program.

elf32_name Service names with an embedded class, **32** here,indicate they work only for the designated class of files.

elf64_name Service names with an embedded class, **64** here,indicate they work only for the designated class of files.

Elf_Type Data types can be class-independent as well, distinguished by *Type*.

Elf32_Type Class-dependent data types have an embedded class name, **32** here.

Elf64_Type Class-dependent data types have an embedded class name, **64** here.

ELF_C_CMD Several functions take commands that control their actions. These values are members of the **Elf_Cmd** enumeration; they range from zero through **ELF_C_NUM-1**.

ELF_F_FLAG Several functions take flags that control library status and/or actions. Flags are bits that may be combined.

ELF32_FSZ_TYPE
These constants give the file sizes in bytes of the basic ELF types for the 32-bit class of files. See **elf_fsize** for more information.

ELF64_FSZ_TYPE
These constants give the file sizes in bytes of the basic ELF types for the 64-bit class of files. See **elf_fsize** for more information.

ELF_K_KIND The function **elf_kind** identifies the *KIND* of file associated with an ELF descriptor. These values are members of the **Elf_Kind** enumeration; they range from zero through **ELF_K_NUM-1**.

ELF_T_TYPE When a service function, such as **elf_xlate**, deals with multiple types, names of this form specify the desired *TYPE*. Thus, for example, **ELF_T_EHDR** is directly related to **Elf32_Ehdr** These values are members of the **Elf_Type** enumeration; they range from zero through **ELF_T_NUM-1**.

NOTICES

Information in the ELF header files is separated into common parts and processor-specific parts. A program can make a processor's information available by including the appropriate header file: **sys/elf_NAME.h** where *NAME* matches the processor name as used in the ELF file header.

Symbol	Processor
parisc	PA RISC

Other processors will be added to the table as necessary. To illustrate, a program could use the following code to "see" the processor-specific information for the WE 32100.

```
#include <libelf.h>
#include <sys/elf_M32.h>
```

Without the **sys/elf_M32.h** definition, only the common ELF information would be visible.

SEE ALSO

a.out(4), ar(4), elf_begin(3E), elf_cntl(3E), elf_end(3E), elf_error(3E), elf_fill(3E), elf_flag(3E), elf_fsize(3E), elf_getarhdr(3E), elf_getarsym(3E), elf_getbase(3E), elf_getdata(3E), elf_getehdr(3E), elf_getident(3E), elf_getphdr(3E), elf_getscn(3E), elf_getshdr(3E), elf_hash(3E), elf_kind(3E), elf_next(3E), elf_rand(3E), elf_rawfile(3E), elf_strptr(3E), elf_update(3E), elf_version(3E), elf_xlate(3E).

NAME

elf_begin - make a file descriptor

SYNOPSIS

cc [*flag*...] *file*... **-lelf** [*library*] ...

```
#include <libelf.h>
Elf *elf_begin(int fildes, Elf_Cmd cmd, Elf *ref);
```

DESCRIPTION

elf_begin, **elf_next**, **elf_rand**, and **elf_end** work together to process ELF object files, either individually or as members of archives. After obtaining an ELF descriptor from **elf_begin**, the program may read an existing file, update an existing file, or create a new file. *fildes* is an open file descriptor that **elf_begin** uses for reading or writing. The initial file offset (see *lseek*(2)) is unconstrained, and the resulting file offset is undefined. *cmd* may have the following values.

ELF_C_NULL When a program sets *cmd* to this value, **elf_begin** returns a null pointer, without opening a new descriptor. *ref* is ignored for this command. See *elf_next*(3E) and the examples below for more information.

ELF_C_READ When a program wishes to examine the contents of an existing file, it should set *cmd* to this value. Depending on the value of *ref*, this command examines archive members or entire files. Three cases can occur.

First, if *ref* is a null pointer, **elf_begin** allocates a new ELF descriptor and prepares to process the entire file. If the file being read is an archive, **elf_begin** also prepares the resulting descriptor to examine the initial archive member on the next call to **elf_begin**, as if the program had used **elf_next** or **elf_rand** to "move" to the initial member.

Second, if *ref* is a non-null descriptor associated with an archive file, **elf_begin** lets a program obtain a separate ELF descriptor associated with an individual member. The program should have used **elf_next** or **elf_rand** to position *ref* appropriately (except for the initial member, which **elf_begin** prepares; see the example below). In this case, *fildes* should be the same file descriptor used for the parent archive.

Finally, if *ref* is a non-null ELF descriptor that is not an archive, **elf_begin** increments the number of activations for the descriptor and returns *ref*, without allocating a new descriptor and without changing the descriptor's read/write permissions. To terminate the descriptor for *ref*, the program must call **elf_end** once for each activation. See *elf_next*(3E) and the examples below for more information.

ELF_C_RDWR This command duplicates the actions of **ELF_C_READ** and additionally allows the program to update the file image (see *elf_update*(3E)). That is, using **ELF_C_READ** gives a read-only view of the file, while **ELF_C_RDWR** lets the program read and write the file. **ELF_C_RDWR** is not valid for archive members. If *ref* is non-null, it must have been created with the **ELF_C_RDWR** command.

ELF_C_WRITE If the program wishes to ignore previous file contents, presumably to create a new file, it should set *cmd* to this value. *ref* is ignored for this command.

elf_begin "works" on all files (including files with zero bytes), providing it can allocate memory for its internal structures and read any necessary information from the file. Programs reading object files thus may call **elf_kind** or **elf_getehdr** to determine the file type (only object files have an ELF header). If the file is an archive with no more members to process, or an error occurs, **elf_begin** returns a null pointer. Otherwise, the return value is a non-null ELF descriptor. Before the first call to **elf_begin**, a program must call **elf_version** to coordinate versions.

System Services

When processing a file, the library decides when to read or write the file, depending on the program's requests. Normally, the library assumes the file descriptor remains usable for the life of the ELF descriptor. If, however, a program must process many files simultaneously and the underlying operating system limits the number of open files, the program can use **elf_cntl** to let it reuse file descriptors. After calling **elf_cntl** with appropriate arguments, the program may close the file descriptor without interfering with the library.

All data associated with an ELF descriptor remain allocated until **elf_end** terminates the descriptor's last activation. After the descriptors have been terminated, the storage is released; attempting to reference such data gives undefined behavior. Consequently, a program that deals with multiple input (or output) files must keep the ELF descriptors active until it finishes with them.

EXAMPLES

A prototype for reading a file appears below. If the file is a simple object file, the program executes the loop one time, receiving a null descriptor in the second iteration. In this case, both **elf** and **arf** will have the same value, the activation count will be two, and the program calls **elf_end** twice to terminate the descriptor. If the file is an archive, the loop processes each archive member in turn, ignoring those that are not object files.

```
if (elf_version(EV_CURRENT) == EV_NONE)
{
     /* library out of date */
     /* recover from error */
}
cmd = ELF_C_READ;
arf = elf_begin(fildes, cmd, (Elf *)0);
while ((elf = elf_begin(fildes, cmd, arf)) != 0)
{
     if ((ehdr = elf32_getehdr(elf)) != 0)
     {
          /* process the file . . . */
     }
     cmd = elf_next(elf);
     elf_end(elf);
}
elf_end(arf);
```

Alternatively, the next example illustrates random archive processing. After identifying the file as an archive, the program repeatedly processes archive members of interest. For clarity, this example omits error checking and ignores simple object files. Additionally, this fragment preserves the ELF descriptors for all archive members, because it does not call **elf_end** to terminate them.

```
elf_version(EV_CURRENT);
arf = elf_begin(fildes, ELF_C_READ, (Elf *)0);
if (elf_kind(arf) != ELF_K_AR)
{
     /* not an archive */
}
/* initial processing */
/* set offset = . . . for desired member header */
while (elf_rand(arf, offset) == offset)
{
     if ((elf = elf_begin(fildes, ELF_C_READ, arf)) == 0)
          break;
     if ((ehdr = elf32_getehdr(elf)) != 0)
     {
          /* process archive member . . . */
     }
     /* set offset = . . . for desired member header */
}
```

The following outline shows how one might create a new ELF file. This example is simplified to show the overall flow.

```
elf_version(EV_CURRENT);
fildes = open("path/name", O_RDWR|O_TRUNC|O_CREAT, 0666);
if ((elf = elf_begin(fildes, ELF_C_WRITE, (Elf *)0)) == 0)
return;
ehdr = elf32_newehdr(elf);
phdr = elf32_newphdr(elf, count);
scn = elf_newscn(elf);
shdr = elf32_getshdr(scn);
```

```
data = elf_newdata(scn);
elf_update(elf, ELF_C_WRITE);
elf_end(elf);
```

Finally, the following outline shows how one might update an existing ELF file. Again, this example is simplified to show the overall flow.

```
elf_version(EV_CURRENT);
fildes = open("path/name", O_RDWR);
elf = elf_begin(fildes, ELF_C_RDWR, (Elf *)0);

/* add new or delete old information . . . */

close(creat("path/name", 0666));
elf_update(elf, ELF_C_WRITE);
elf_end(elf);
```

In the example above, the call to **creat** truncates the file, thus ensuring the resulting file will have the "right" size. Without truncation, the updated file might be as big as the original, even if information were deleted. The library truncates the file, if it can, with **ftruncate** (see *truncate*(2)). Some systems, however, do not support **ftruncate**, and the call to **creat** protects against this. Notice that both file creation examples open the file with write *and* read permissions. On systems that support **mmap**, the library uses it to enhance performance, and **mmap** requires a readable file descriptor. Although the library can use a write-only file descriptor, the application will not obtain the performance advantages of **mmap**.

SEE ALSO

ar(4), creat(2), elf(3E), elf_cntl(3E), elf_end(3E), elf_getarhdr(3E), elf_getbase(3E), elf_getdata(3E), elf_getehdr(3E), elf_getphdr(3E), elf_getscn(3E), elf_kind(3E), elf_next(3E), elf_rand(3E), elf_rawfile(3E), elf_update(3E), elf_version(3E), lseek(2), mmap(2), open(2), truncate(2).

NAME

elf_cntl - control a file descriptor

SYNOPSIS

cc *[*flag*...] file...* **-lelf** [*library*] ...

```
#include <libelf.h>
int elf_cntl(Elf *elf, Elf_Cmd cmd);
```

DESCRIPTION

elf_cntl instructs the library to modify its behavior with respect to an ELF descriptor, *elf*. As *elf_begin*(3E) describes, an ELF descriptor can have multiple activations, and multiple ELF descriptors may share a single file descriptor. Generally, **elf_cntl** commands apply to all activations of *elf*. Moreover, if the ELF descriptor is associated with an archive file, descriptors for members within the archive will also be affected as described below. Unless stated otherwise, operations on archive members do not affect the descriptor for the containing archive.

The *cmd* argument tells what actions to take and may have the following values.

ELF_C_FDDONE This value tells the library not to use the file descriptor associated with *elf*. A program should use this command when it has requested all the information it cares to use and wishes to avoid the overhead of reading the rest of the file. The memory for all completed operations remains valid, but later file operations, such as the initial **elf_getdata** for a section, will fail if the data is not in memory already.

ELF_C_FDREAD This command is similar to **ELF_C_FDDONE**, except it forces the library to read the rest of the file. A program should use this command when it must close the file descriptor but has not yet read everything it needs from the file. After **elf_cntl** completes the **ELF_C_FDREAD** command, future operations, such as **elf_getdata,** will use the memory version of the file without needing to use the file descriptor.

If **elf_cntl** succeeds, it returns zero. Otherwise, *elf* was null or an error occurred, and the function returns -1.

NOTICES

If the program wishes to use the "raw" operations (see **elf_rawdata**, described in *elf_getdata*(3E), and *elf_rawfile*(3E)) after disabling the file descriptor with **ELF_C_FDDONE** or **ELF_C_FDREAD**, it must execute the raw operations explicitly beforehand. Otherwise, the raw file operations will fail. Calling **elf_rawfile** makes the entire image available, thus supporting subsequent **elf_rawdata** calls.

SEE ALSO

elf(3E), elf_begin(3E), elf_getdata(3E), elf_rawfile(3E).

NAME

elf_end - finish using an object file

SYNOPSIS

cc [*flag*...] *file*... **-lelf** [*library*] ...

```
#include <libelf.h>
int elf_end(Elf *elf);
```

DESCRIPTION

A program uses **elf_end** to terminate an ELF descriptor, *elf*, and to deallocate data associated with the descriptor. Until the program terminates a descriptor, the data remain allocated. *elf* should be a value previously returned by **elf_begin**; a null pointer is allowed as an argument, to simplify error handling. If the program wishes to write data associated with the ELF descriptor to the file, it must use **elf_update** before calling **elf_end**.

As *elf_begin*(3E) explains, a descriptor can have more than one activation. Calling **elf_end** removes one activation and returns the remaining activation count. The library does not terminate the descriptor until the activation count reaches zero. Consequently, a zero return value indicates the ELF descriptor is no longer valid.

SEE ALSO

elf(3E), elf_begin(3E), elf_update(3E).

NAME

elf_errmsg, elf_errno - error handling

SYNOPSIS

cc [*flag*...] *file*... **-lelf** [*library*] ...

```
#include <libelf.h>
const char *elf_errmsg(int err);
int elf_errno(void);
```

DESCRIPTION

If an ELF library function fails, a program may call **elf_errno** to retrieve the library's internal error number. As a side effect, this function resets the internal error number to zero, which indicates no error.

elf_errmsg takes an error number, *err*, and returns a null-terminated error message (with no trailing new-line) that describes the problem. A zero *err* retrieves a message for the most recent error. If no error has occurred, the return value is a null pointer (not a pointer to the null string). Using *err* of -1 also retrieves the most recent error, except it guarantees a non-null return value, even when no error has occurred. If no message is available for the given number, **elf_errmsg** returns a pointer to an appropriate message. This function does not have the side effect of clearing the internal error number.

EXAMPLES

The following fragment clears the internal error number and checks it later for errors. Unless an error occurs after the first call to **elf_errno**, the next call will return zero.

```
(void)elf_errno();
while (more_to_do)
{
        /* processing ... */
        if ((err = elf_errno()) != 0)
        {
                msg = elf_errmsg(err);
                /* print msg */
        }
}
```

SEE ALSO

elf(3E), elf_version(3E).

NAME

elf_fill - set fill byte

SYNOPSIS

cc [*flag*...] *file*... **-lelf** [*library*] ...

```
#include <libelf.h>
void elf_fill(int fill);
```

DESCRIPTION

Alignment constraints for ELF files sometimes require the presence of "holes." For example, if the data for one section are required to begin on an eight-byte boundary, but the preceding section is too "short," the library must fill the intervening bytes. These bytes are set to the *fill* character. The library uses zero bytes unless the application supplies a value. See *elf_getdata*(3E) for more information about these holes.

NOTICES

An application can assume control of the object file organization by setting the **ELF_F_LAYOUT** bit (see *elf_flag*(3E)). When this is done, the library does *not* fill holes.

SEE ALSO

elf(3E), elf_getdata(3E), elf_flag(3E), elf_update(3E).

NAME

elf_flagdata, elf_flagehdr, elf_flagelf, elf_flagphdr, elf_flagscn, elf_flagshdr - manipulate flags

SYNOPSIS

cc [*flag*...] *file*... **-lelf** [*library*] ...

```
#include <libelf.h>

unsigned elf_flagdata(Elf_Data *data, Elf_Cmd cmd, unsigned flags);
unsigned elf_flagehdr(Elf *elf, Elf_Cmd cmd, unsigned flags);
unsigned elf_flagelf(Elf *elf, Elf_Cmd cmd, unsigned flags);
unsigned elf_flagphdr(Elf *elf, Elf_Cmd cmd, unsigned flags);
unsigned elf_flagscn(Elf_Scn *scn, Elf_Cmd cmd, unsigned flags);
unsigned elf_flagshdr(Elf_Scn *scn, Elf_Cmd cmd, unsigned flags);
```


DESCRIPTION

These functions manipulate the flags associated with various structures of an ELF file. Given an ELF descriptor *elf*, a data descriptor *data*, or a section descriptor *scn*, the functions may set or clear the associated status bits, returning the updated bits. A null descriptor is allowed, to simplify error handling; all functions return zero for this degenerate case.

cmd may have the following values:

ELF_C_CLR The functions clear the bits that are asserted in *flags*. Only the non-zero bits in *flags* are cleared; zero bits do not change the status of the descriptor.

ELF_C_SET The functions set the bits that are asserted in *flags*. Only the non-zero bits in *flags* are set; zero bits do not change the status of the descriptor.

Descriptions of the defined flags bits appear below.

ELF_F_DIRTY When the program intends to write an ELF file, this flag asserts the associated information needs to be written to the file. Thus, for example, a program that wished to update the ELF header of an existing file would call **elf_flagehdr** with this bit set in *flags* and *cmd* equal to **ELF_C_SET**. A later call to **elf_update** would write the marked header to the file.

ELF_F_LAYOUT Normally, the library decides how to arrange an output file. That is, it automatically decides where to place sections, how to align them in the file, etc. If this bit is set for an ELF descriptor, the program assumes responsibility for determining all file positions. This bit is meaningful only for **elf_flagelf** and applies to the entire file associated with the descriptor.

When a flag bit is set for an item, it affects all the subitems as well. Thus, for example, if the program sets the **ELF_F_DIRTY** bit with **elf_flagelf**, the entire logical file is "dirty."

EXAMPLES

The following fragment shows how one might mark the ELF header to be written to the output file.

```
ehdr = elf32_getehdr(elf);
/* dirty ehdr . . . */
elf_flagehdr(elf, ELF_C_SET, ELF_F_DIRTY);
```

SEE ALSO

elf(3E), elf_end(3E), elf_getdata(3E), elf_getehdr(3E), elf_update(3E).

NAME

elf32_fsize, elf64_fsize - return the size of an object file type for elf32 files, return the size of an object file type for elf64 files, respectively.

SYNOPSIS

cc [*flag*...] *file*... **-lelf** [*library*] ...

```
#include <libelf.h>
size_t elf32_fsize(Elf_Type type, size_t count, unsigned ver);
size_t elf64_fsize(Elf_Type type, size_t count, unsigned ver);
```

DESCRIPTION

elf32_fsize gives the size in bytes of the 32-bit file representation of *count* data objects with the given *type*. The library uses version *ver* to calculate the size (see *elf*(3E) and *elf_version*(3E)).

Constant values are available for the sizes of fundamental types.

Elf_Type	File Size	Memory Size
ELF_T_ADDR	ELF32_FSZ_ADDR	sizeof(Elf32_Addr)
ELF_T_BYTE	1	sizeof(unsigned char)
ELF_T_HALF	ELF32_FSZ_HALF	sizeof(Elf32_Half)
ELT_T_OFF	ELF32_FSZ_OFF	sizeof(Elf32_Off)
ELF_T_SWORD	ELF32_FSZ_SWORD	sizeof(Elf32_Sword)
ELF_T_WORD	ELF32_FSZ_WORD	sizeof(Elf32_Word)

elf32_fsize returns zero if the value of *type* or *ver* is unknown. See *elf_xlate*(3E) for a list of the *type* values.

elf64_fsize gives the size in bytes of the 64-bit file representation of *count* data objects with the given *type*. The library uses version *ver* to calculate the size (see *elf*(3E) and *elf_version*(3E)).

Constant values are available for the sizes of fundamental types.

Elf_Type	File Size	Memory Size
ELF_T_ADDR	ELF64_FSZ_ADDR	sizeof(Elf64_Addr)
ELF_T_BYTE	1	sizeof(unsigned char)
ELF_T_HALF	ELF64_FSZ_HALF	sizeof(Elf64_Half)
ELT_T_OFF	ELF64_FSZ_OFF	sizeof(Elf64_Off)
ELF_T_SWORD	ELF64_FSZ_SWORD	sizeof(Elf64_Sword)
ELF_T_WORD	ELF64_FSZ_WORD	sizeof(Elf64_Word)

elf64_fsize returns zero if the value of *type* or *ver* is unknown. See *elf_xlate*(3E) for a list of the *type* values.

SEE ALSO

elf(3E), elf_version(3E), elf_xlate(3E).

NAME

elf_getarhdr - retrieve archive member header

SYNOPSIS

cc [*flag*...] *file*... **-lelf** [*library*] ...

```
#include <libelf.h>

Elf_Arhdr *elf_getarhdr(Elf *elf);
```

DESCRIPTION

elf_getarhdr returns a pointer to an archive member header, if one is available for the ELF descriptor *elf*. Otherwise, no archive member header exists, an error occurred, or *elf* was null; **elf_getarhdr** then returns a null value. The header includes the following members.

```
char            *ar_name;
time_t          ar_date;
long            ar_uid;
long            ar_gid;
unsigned long   ar_mode;
off_t           ar_size;
char            *ar_rawname;
```

An archive member name, available through **ar_name**, is a null-terminated string, with the **ar** format control characters removed. The **ar_rawname** member holds a null-terminated string that represents the original name bytes in the file, including the terminating slash and trailing blanks as specified in the archive format.

In addition to "regular" archive members, the archive format defines some special members. All special member names begin with a slash (**/**), distinguishing them from regular members (whose names may not contain a slash). These special members have the names (**ar_name**) defined below.

/ This is the archive symbol table. If present, it will be the first archive member. A program may access the archive symbol table through **elf_getarsym**. The information in the symbol table is useful for random archive processing (see *elf_rand*(3E)).

// This member, if present, holds a string table for long archive member names. An archive member's header contains a 16-byte area for the name, which may be exceeded in some file systems. The library automatically retrieves long member names from the string table, setting **ar_name** to the appropriate value.

Under some error conditions, a member's name might not be available. Although this causes the library to set **ar_name** to a null pointer, the **ar_rawname** member will be set as usual.

SEE ALSO

ar(4), elf(3E), elf_begin(3E), elf_getarsym(3E), elf_rand(3E).

NAME

elf_getarsym - retrieve archive symbol table

SYNOPSIS

cc [*flag*...] *file*... **-lelf** [*library*] ...

```
#include <libelf.h>

Elf_Arsym *elf_getarsym(Elf *elf, size_t *ptr);
```

DESCRIPTION

elf_getarsym returns a pointer to the archive symbol table, if one is available for the ELF descriptor **elf**. Otherwise, the archive doesn't have a symbol table, an error occurred, or **elf** was null; **elf_getarsym** then returns a null value. The symbol table is an array of structures that include the following members.

```
char            *as_name;
size_t          as_off;
unsigned long   as_hash;
```

These members have the following semantics.

as_name A pointer to a null-terminated symbol name resides here.

as_off This value is a byte offset from the beginning of the archive to the member's header. The archive member residing at the given offset defines the associated symbol. Values in **as_off** may be passed as arguments to **elf_rand** to access the desired archive member.

as_hash This is a hash value for the name, as computed by **elf_hash.**

If *ptr* is non-null, the library stores the number of table entries in the location to which *ptr* points. This value is set to zero when the return value is null. The table's last entry, which is included in the count, has a null **as_name**, a zero value for **as_off**, and **~0UL** for **as_hash**.

SEE ALSO

ar(4), elf(3E), elf_getarhdr(3E), elf_hash(3E), elf_rand(3E).

NAME

elf_getbase - get the base offset for an object file

SYNOPSIS

cc [*flag*...] *file*... **-lelf** [*library*] ...

```
#include <libelf.h>
off_t elf_getbase(Elf *elf);
```

DESCRIPTION

elf_getbase returns the file offset of the first byte of the file or archive member associated with *elf*, if it is known or obtainable, and -1 otherwise. A null *elf* is allowed, to simplify error handling; the return value in this case is -1. The base offset of an archive member is the beginning of the member's information, not the beginning of the archive member header.

SEE ALSO

ar(4), elf(3E), elf_begin(3E).

NAME

elf_getdata, elf_newdata, elf_rawdata - get section data

SYNOPSIS

cc [*flag*...] *file*... **-lelf** [*library*] ...

```
#include <libelf.h>
Elf_Data *elf_getdata(Elf_Scn *scn, Elf_Data *data);
Elf_Data *elf_newdata(Elf_Scn *scn);
Elf_Data *elf_rawdata(Elf_Scn *scn, Elf_Data *data);
```

DESCRIPTION

These functions access and manipulate the data associated with a section descriptor, *scn*. When reading an existing file, a section will have a single data buffer associated with it. A program may build a new section in pieces, however, composing the new data from multiple data buffers. For this reason, "the" data for a section should be viewed as a list of buffers, each of which is available through a data descriptor.

elf_getdata lets a program step through a section's data list. If the incoming data descriptor, *data*, is null, the function returns the first buffer associated with the section. Otherwise, *data* should be a data descriptor associated with *scn*, and the function gives the program access to the next data element for the section. If *scn* is null or an error occurs, **elf_getdata** returns a null pointer.

elf_getdata translates the data from file representations into memory representations (see *elf_xlate*(3E)) and presents objects with memory data types to the program, based on the file's *class* (see *elf*(3E)). The working library version (see *elf_version*(3E)) specifies what version of the memory structures the program wishes **elf_getdata** to present.

elf_newdata creates a new data descriptor for a section, appending it to any data elements already associated with the section. As described below, the new data descriptor appears empty, indicating the element holds no data. For convenience, the descriptor's type (**d_type** below) is set to **ELF_T_BYTE**, and the version (**d_version** below) is set to the working version. The program is responsible for setting (or changing) the descriptor members as needed. This function implicitly sets the **ELF_F_DIRTY** bit for the section's data (see *elf_flag*(3E)). If *scn* is null or an error occurs, **elf_newdata** returns a null pointer.

elf_rawdata differs from **elf_getdata** by returning only uninterpreted bytes, regardless of the section type. This function typically should be used only to retrieve a section image from a file being read, and then only when a program must avoid the automatic data translation described below. Moreover, a program may not close or disable (see *elf_cntl*(3E)) the file descriptor associated with *elf* before the initial raw operation, because **elf_rawdata** might read the data from the file to ensure it doesn't interfere with **elf_getdata**. See *elf_rawfile*(3E) for a related facility that applies to the entire file. When **elf_getdata** provides the right translation, its use is recommended over **elf_rawdata**. If *scn* is null or an error occurs, **elf_rawdata** returns a null pointer.

The **Elf_Data** structure includes the following members.

```
void            *d_buf;
Elf_Type        d_type;
size_t          d_size;
off_t           d_off;
size_t          d_align;
unsigned        d_version;
```

These members are available for direct manipulation by the program. Descriptions appear below.

d_buf A pointer to the data buffer resides here. A data element with no data has a null pointer.

d_type This member's value specifies the type of the data to which **d_buf** points. A section's type determines how to interpret the section contents, as summarized below. **d_size** This member holds the total size, in bytes, of the memory occupied by the data. This may differ from the size as represented in the file. The size will be zero if no data exist. (See the discussion of **SHT_NOBITS** below for more information.)

d_off This member gives the offset, within the section, at which the buffer resides. This offset is relative to the file's section, not the memory object's.

d_align This member holds the buffer's required alignment, from the beginning of the section. That is, **d_off** will be a multiple of this member's value. For example, if this member's value is four, the beginning of the buffer will be four-byte aligned within the section. Moreover, the

entire section will be aligned to the maximum of its constituents, thus ensuring appropriate alignment for a buffer within the section and within the file.

d_version This member holds the version number of the objects in the buffer. When the library originally read the data from the object file, it used the working version to control the translation to memory objects.

Data Alignment

As mentioned above, data buffers within a section have explicit alignment constraints. Consequently, adjacent buffers sometimes will not abut, causing "holes" within a section. Programs that create output files have two ways of dealing with these holes.

First, the program can use **elf_fill** to tell the library how to set the intervening bytes. When the library must generate gaps in the file, it uses the fill byte to initialize the data there. The library's initial fill value is zero, and **elf_fill** lets the application change that.

Second, the application can generate its own data buffers to occupy the gaps, filling the gaps with values appropriate for the section being created. A program might even use different fill values for different sections. For example, it could set text sections' bytes to no-operation instructions, while filling data section holes with zero. Using this technique, the library finds no holes to fill, because the application eliminated them.

Section and Memory Types

elf_getdata interprets sections' data according to the section type, as noted in the section header available through *elf_getshdr*(3E). The following table shows the section types and how the library represents them with memory data types for the 32-bit file class. Other classes would have similar tables. By implication, the memory data types control translation by *elf_xlate*(3E).

Section Type	Elf_Type	32-Bit Type
SHT_DYNAMIC	ELF_T_DYN	Elf32_Dyn
SHT_DYNSYM	ELF_T_SYM	Elf32_Sym
SHT_HASH	ELF_T_WORD	Elf32_Word
SHT_NOBITS	ELF_T_BYTE	unsigned char
SHT_NOTE	ELF_T_BYTE	unsigned char
SHT_NULL	none	none
SHT_PROGBITS	ELF_T_BYTE	unsigned char
SHT_REL	ELF_T_REL	Elf32_Rel
SHT_RELA	ELF_T_RELA	Elf32_Rela
SHT_STRTAB	ELF_T_BYTE	unsigned char
SHT_SYMTAB	ELF_T_SYM	Elf32_Sym
other	ELF_T_BYTE	unsigned char

elf_rawdata creates a buffer with type **ELF_T_BYTE**.

As mentioned above, the program's working version controls what structures the library creates for the application. The library similarly interprets section types according to the versions. If a section type "belongs" to a version newer than the application's working version, the library does not translate the section data. Because the application cannot know the data format in this case, the library presents an untranslated buffer of type **ELF_T_BYTE**, just as it would for an unrecognized section type.

A section with a special type, **SHT_NOBITS**, occupies no space in an object file, even when the section header indicates a non-zero size. **elf_getdata** and **elf_rawdata** "work" on such a section, setting the *data* structure to have a null buffer pointer and the type indicated above. Although no data is present, the **d_size** value is set to the size from the section header. When a program is creating a new section of type **SHT_NOBITS**, it should use **elf_newdata** to add data buffers to the section. These "empty" data buffers should have the **d_size** members set to the desired size and the **d_buf** members set to null.

EXAMPLES

The following fragment obtains the string table that holds section names (ignoring error checking). See *elf_strptr*(3E) for a variation of string table handling.

```
ehdr = elf32_getehdr(elf);
scn = elf_getscn(elf, (size_t)ehdr->e_shstrndx);
shdr = elf32_getshdr(scn);
if (shdr->sh_type != SHT_STRTAB)
{
```

```
        /* not a string table */
}
data = 0;
if ((data = elf_getdata(scn, data)) == 0
                || data->d_size == 0)
{
        /* error or no data */
}
```

The **e_shstrndx** member in an ELF header holds the section table index of the string table. The program gets a section descriptor for that section, verifies it is a string table, and then retrieves the data. When this fragment finishes, **data->d_buf** points at the first byte of the string table, and **data->d_size** holds the string table's size in bytes.

e

SEE ALSO

elf(3E), elf_cntl(3E), elf_fill(3E), elf_flag(3E), elf_getehdr(3E), elf_getscn(3E), elf_getshdr(3E), elf_rawfile(3E), elf_strptr(3E), elf_version(3E), elf_xlate(3E).

NAME

elf32_getehdr, elf32_newehdr, elf64_getehdr, elf64_newehdr - retrieve class-dependent object file header for elf32 and elf64 files, respectively

SYNOPSIS

cc [*flag*...] *file*... **-lelf** [*library*] ...

```
#include <libelf.h>

Elf32_Ehdr *elf32_getehdr(Elf *elf);
Elf32_Ehdr *elf32_newehdr(Elf *elf);
Elf64_Ehdr *elf64_getehdr(Elf *elf);
Elf64_Ehdr *elf64_newehdr(Elf *elf);
```

DESCRIPTION

For a 32-bit class file, **elf32_getehdr** returns a pointer to an ELF header, if one is available for the ELF descriptor *elf*. If no header exists for the descriptor, **elf32_newehdr** allocates a "clean" one, but it otherwise behaves the same as **elf32_getehdr**. It does not allocate a new header if one exists already. If no header exists (for **elf_getehdr**), one cannot be created (for **elf_newehdr**), a system error occurs, the file is not a 32-bit class file, or *elf* is null, both functions return a null pointer. The header includes the following members.

```
unsigned char   e_ident[EI_NIDENT];
Elf32_Half e_type;
Elf32_Half e_machine;
Elf32_Word e_version;
Elf32_Addr e_entry;
Elf32_Off       e_phoff;
Elf32_Off       e_shoff;
Elf32_Word e_flags;
Elf32_Half e_ehsize;
Elf32_Half e_phentsize;
Elf32_Half e_phnum;
Elf32_Half e_shentsize;
Elf32_Half e_shnum;
Elf32_Half e_shstrndx;
```

elf32_newehdr automatically sets the **ELF_F_DIRTY** bit (see *elf_flag*(3E)). A program may use **elf_getident** to inspect the identification bytes from a file.

For a 64-bit class file, **elf64_getehdr** returns a pointer to an ELF header, if one is available for the ELF descriptor *elf*. If no header exists for the descriptor, **elf64_newehdr** allocates a "clean" one, but it otherwise behaves the same as **elf64_getehdr**. It does not allocate a new header if one exists already. If no header exists (for **elf_getehdr**), one cannot be created (for **elf_newehdr**), a system error occurs, the file is not a 64-bit class file, or *elf* is null, both functions return a null pointer. The header includes the following members.

```
unsigned char   e_ident[EI_NIDENT];
Elf64_Half e_type;
Elf64_Half e_machine;
Elf64_Word e_version;
Elf64_Addr e_entry;
Elf64_Off       e_phoff;
Elf64_Off       e_shoff;
Elf64_Word e_flags;
Elf64_Half e_ehsize;
Elf64_Half e_phentsize;
Elf64_Half e_phnum;
Elf64_Half e_shentsize;
Elf64_Half e_shnum;
Elf64_Half e_shstrndx;
```

elf64_newehdr automatically sets the **ELF_F_DIRTY** bit (see *elf_flag*(3E)). A program may use **elf_getident** to inspect the identification bytes from a file.

SEE ALSO
elf(3E), elf_begin(3E), elf_flag(3E), elf_getident(3E).

NAME

elf_getident - retrieve file identification data

SYNOPSIS

cc [*flag*...] *file*... **-lelf** [*library*] ...

```
#include <libelf.h>
char *elf_getident(Elf *elf, size_t *ptr);
```

DESCRIPTION

As *elf*(3E) explains, ELF provides a framework for various classes of files, where basic objects may have 32 bits, 64 bits, and so forth. To accommodate these differences, without forcing the larger sizes on smaller machines, the initial bytes in an ELF file hold identification information common to all file classes. Every ELF header's **e_ident** has **EI_NIDENT** bytes with the following interpretation.

e_ident Index	Value	Purpose
EI_MAG0	ELFMAG0	File identification
EI_MAG1	ELFMAG1	
EI_MAG2	ELFMAG2	
EI_MAG3	ELFMAG3	
EI_CLASS	ELFCLASSNONE	File class
	ELFCLASS32	
	ELFCLASS64	
EI_DATA	ELFDATANONE	Data encoding
	ELFDATA2LSB	
	ELFDATA2MSB	
EI_VERSION	EV_CURRENT	File version
7-15	0	Unused, set to zero

Other kinds of files (see *elf_kind*(3E)) also may have identification data, though they would not conform to **e_ident**.

elf_getident returns a pointer to the file's "initial bytes." If the library recognizes the file, a conversion from the file image to the memory image may occur. In any case, the identification bytes are guaranteed not to have been modified, though the size of the unmodified area depends on the file type. If *ptr* is non-null, the library stores the number of identification bytes in the location to which *ptr* points. If no data is present, *elf* is null, or an error occurs, the return value is a null pointer, with zero optionally stored through *ptr*.

SEE ALSO

elf(3E), elf_begin(3E), elf_getehdr(3E), elf_kind(3E), elf_rawfile(3E).

NAME

elf32_getphdr, elf32_newphdr, elf64_getphdr, elf64_newphdr - retrieve class-dependent program header table for elf32 and elf64 files, respectively

SYNOPSIS

cc [*flag*...] *file*... **-lelf** [*library*] ...

```
#include <libelf.h>

Elf32_Ehdr *elf32_getphdr(Elf *elf);
Elf32_Ehdr *elf32_newphdr(Elf *elf, size_t count);
Elf64_Ehdr *elf64_getphdr(Elf *elf);
Elf64_Ehdr *elf64_newphdr(Elf *elf, size_t count);
```

DESCRIPTION

For a 32-bit class file, **elf32_getphdr** returns a pointer to the program execution header table, if one is available for the ELF descriptor *elf*.

elf32_newphdr allocates a new table with *count* entries, regardless of whether one existed previously, and sets the **ELF_F_DIRTY** bit for the table (see *elf_flag*(3E)). Specifying a zero *count* deletes an existing table. Note this behavior differs from that of **elf32_newehdr** (see *elf_getehdr*(3E)), allowing a program to replace or delete the program header table, changing its size if necessary.

If no program header table exists, the file is not a 32-bit class file, an error occurs, or *elf* is null, both functions return a null pointer. Additionally, **elf32_newphdr** returns a null pointer if *count* is zero.

The table is an array of **Elf32_Phdr** structures, each of which includes the following members.

```
Elf32_Word      p_type;
Elf32_Off       p_offset;
Elf32_Addr      p_vaddr;
Elf32_Addr      p_paddr;
Elf32_Word      p_filesz;
Elf32_Word      p_memsz;
Elf32_Word      p_flags;
Elf32_Word      p_align;
```

The ELF header's e_phnum member tells how many entries the program header table has (see *elf_getehdr*(3E)). A program may inspect this value to determine the size of an existing table; **elf32_newphdr** automatically sets the member's value to *count*. If the program is building a new file, it is responsible for creating the file's ELF header before creating the program header table.

For a 64-bit class file, **elf64_getphdr** returns a pointer to the program execution header table, if one is available for the ELF descriptor *elf*.

elf64_newphdr allocates a new table with *count* entries, regardless of whether one existed previously, and sets the **ELF_F_DIRTY** bit for the table (see **elf_flag**(3E)). Specifying a zero *count* deletes an existing table. Note this behavior differs from that of **elf64_newehdr** (see *elf_getehdr*(3E)), allowing a program to replace or delete the program header table, changing its size if necessary.

If no program header table exists, the file is not a 64-bit class file, an error occurs, or *elf* is null, both functions return a null pointer. Additionally, **elf64_newphdr** returns a null pointer if *count* is zero.

The table is an array of **Elf64_Phdr** structures, each of which includes the following members.

```
Elf64_Word      p_type;
Elf64_Off       p_offset;
Elf64_Addr      p_vaddr;
Elf64_Addr      p_paddr;
Elf64_Word      p_filesz;
Elf64_Word      p_memsz;
Elf64_Word      p_flags;
Elf64_Word      p_align;
```

The ELF header's e_phnum member tells how many entries the program header table has (see *elf_getehdr*(3E)). A program may inspect this value to determine the size of an existing table; **elf64_newphdr** automatically sets the member's value to *count*. If the program is building a new file, it is responsible for creating the file's ELF header before creating the program header table.

SEE ALSO
elf(3E), elf_begin(3E), elf_flag(3E), elf_getehdr(3E).

NAME

elf_getscn, elf_ndxscn, elf_newscn, elf_nextscn - get section information

SYNOPSIS

cc [*flag*...] *file*... **-lelf** [*library*] ...

```
#include <libelf.h>

Elf_Scn *elf_getscn(Elf *elf, size_t index);
size_t elf_ndxscn(Elf_Scn *scn);
Elf_Scn *elf_newscn(Elf *elf);
Elf_Scn *elf_nextscn(Elf *elf, Elf_Scn *scn);
```

DESCRIPTION

These functions provide indexed and sequential access to the sections associated with the ELF descriptor *elf*. If the program is building a new file, it is responsible for creating the file's ELF header before creating sections; see *elf_getehdr*(3E).

elf_getscn returns a section descriptor, given an *index* into the file's section header table. Note the first "real" section has index 1. Although a program can get a section descriptor for the section whose *index* is 0 (**SHN_UNDEF**, the undefined section), the section has no data and the section header is "empty" (though present). If the specified section does not exist, an error occurs, or *elf* is null, **elf_getscn** returns a null pointer.

elf_newscn creates a new section and appends it to the list for *elf*. Because the **SHN_UNDEF** section is required and not "interesting" to applications, the library creates it automatically. Thus the first call to **elf_newscn** for an ELF descriptor with no existing sections returns a descriptor for section 1. If an error occurs or *elf* is null, **elf_newscn** returns a null pointer.

After creating a new section descriptor, the program can use **elf_getshdr** to retrieve the newly created, "clean" section header. The new section descriptor will have no associated data (see *elf_getdata*(3E)). When creating a new section in this way, the library updates the **e_shnum** member of the ELF header and sets the **ELF_F_DIRTY** bit for the section (see *elf_flag*(3E)). If the program is building a new file, it is responsible for creating the file's ELF header (see *elf_getehdr*(3E)) before creating new sections.

elf_nextscn takes an existing section descriptor, *scn*, and returns a section descriptor for the next higher section. One may use a null *scn* to obtain a section descriptor for the section whose index is 1 (skipping the section whose index is **SHN_UNDEF**). If no further sections are present or an error occurs, **elf_nextscn** returns a null pointer.

elf_ndxscn takes an existing section descriptor, *scn*, and returns its section table index. If *scn* is null or an error occurs, **elf_ndxscn** returns **SHN_UNDEF**.

EXAMPLES

An example of sequential access appears below. Each pass through the loop processes the next section in the file; the loop terminates when all sections have been processed.

```
scn = 0;
while ((scn = elf_nextscn(elf, scn)) != 0)
{
        /* process section */
}
```

SEE ALSO

elf(3E), elf_begin(3E), elf_flag(3E), elf_getdata(3E), elf_getehdr(3E), elf_getshdr(3E).

NAME

elf32_getshdr, elf64_getshdr - retrieve class-dependent section header for elf32 and elf64 files, respectively

SYNOPSIS

cc [*flag*...] *file*... **-lelf** [*library*] ...

```
#include <libelf.h>

Elf32_Shdr *elf32_getshdr(Elf_Scn *scn);
Elf64_Shdr *elf64_getshdr(Elf_Scn *scn);
```

DESCRIPTION

For a 32-bit class file, **elf32_getshdr** returns a pointer to a section header for the section descriptor *scn*. Otherwise, the file is not a 32-bit class file, *scn* was null, or an error occurred; **elf32_getshdr** then returns NULL. The header includes the following members.

```
Elf32_Word      sh_name;
Elf32_Word      sh_type;
Elf32_Word      sh_flags;
Elf32_Addr      sh_addr;
Elf32_Off       sh_offset;
Elf32_Word      sh_size;
Elf32_Word      sh_link;
Elf32_Word      sh_info;
Elf32_Word      sh_addralign;
Elf32_Word      sh_entsize;
```

For a 64-bit class file, **elf64_getshdr** returns a pointer to a section header for the section descriptor *scn*. Otherwise, the file is not a 64-bit class file, *scn* was null, or an error occurred; **elf64_getshdr** then returns NULL. The header includes the following members.

```
Elf64_Word      sh_name;
Elf64_Word      sh_type;
Elf64_Word      sh_flags;
Elf64_Addr      sh_addr;
Elf64_Off       sh_offset;
Elf64_Word      sh_size;
Elf64_Word      sh_link;
Elf64_Word      sh_info;
Elf64_Word      sh_addralign;
Elf64_Word      sh_entsize;
```

If the program is building a new file, it is responsible for creating the file's ELF header before creating sections.

SEE ALSO

elf(3E), elf_flag(3E), elf_getscn(3E), elf_strptr(3E).

NAME

elf_hash - compute hash value

SYNOPSIS

cc [*flag*...] *file*... **-lelf** [*library*] ...

```
#include <libelf.h>

unsigned long elf_hash(const char *name);
```

DESCRIPTION

elf_hash computes a hash value, given a null terminated string, *name*. The returned hash value, *h*, can be used as a bucket index, typically after computing *h* mod *x* to ensure appropriate bounds.

Hash tables may be built on one machine and used on another because **elf_hash** uses unsigned arithmetic to avoid possible differences in various machines' signed arithmetic. Although *name* is shown as **char*** above, **elf_hash** treats it as **unsigned char*** to avoid sign extension differences. Using **char*** eliminates type conflicts with expressions such as **elf_hash** ("name").

ELF files' symbol hash tables are computed using this function (see *elf_getdata*(3E) and *elf_xlate*(3E)). The hash value returned is guaranteed not to be the bit pattern of all ones (**~0UL**).

SEE ALSO

elf(3E), elf_getdata(3E), elf_xlate(3E).

NAME

elf_kind - determine file type

SYNOPSIS

cc [*flag*...] *file*... **-lelf** [*library*] ...

```
#include <libelf.h>
Elf_Kind elf_kind(Elf *elf);
```

DESCRIPTION

This function returns a value identifying the kind of file associated with an ELF descriptor *elf*. Currently defined values appear below.

ELF_K_AR The file is an archive (see *ar*(4)). An ELF descriptor may also be associated with an archive *member*, not the archive itself, and then **elf_kind** identifies the member's type.

ELF_K_ELF The file is an ELF file. The program may use **elf_getident** to determine the class. Other functions, such as **elf_getehdr**, are available to retrieve other file information.

ELF_K_NONE This indicates a kind of file unknown to the library.

Other values are reserved, to be assigned as needed to new kinds of files. *elf* should be a value previously returned by **elf_begin**. A null pointer is allowed, to simplify error handling, and causes **elf_kind** to return **ELF_K_NONE**.

SEE ALSO

ar(4), elf(3E), elf_begin(3E), elf_getehdr(3E), elf_getident(3E).

NAME

elf_next - sequential archive member access

SYNOPSIS

cc [*flag*...] *file*... **-lelf** [*library*] ...

```
#include <libelf.h>
Elf_Cmd elf_next(Elf *elf);
```

DESCRIPTION

elf_next, **elf_rand**, and **elf_begin** manipulate simple object files and archives. *elf* is an ELF descriptor previously returned from **elf_begin**.

elf_next provides sequential access to the next archive member. That is, having an ELF descriptor, *elf*, associated with an archive member, **elf_next** prepares the containing archive to access the following member when the program calls **elf_begin**. After successfully positioning an archive for the next member, **elf_next** returns the value **ELF_C_READ**. Otherwise, the open file was not an archive, *elf* was null, or an error occurred, and the return value is **ELF_C_NULL**. In either case, the return value may be passed as an argument to **elf_begin**, specifying the appropriate action.

SEE ALSO

ar(4), elf(3E), elf_begin(3E), elf_getarsym(3E), elf_rand(3E).

NAME

elf_rand - random archive member access

SYNOPSIS

cc [*flag*...] *file*... **-lelf** [*library*] ...

```
#include <libelf.h>

size_t elf_rand(Elf *elf, size_t offset);
```

DESCRIPTION

elf_rand, **elf_next**, and **elf_begin** manipulate simple object files and archives. *elf* is an ELF descriptor previously returned from **elf_begin**.

elf_rand provides random archive processing, preparing *elf* to access an arbitrary archive member. *elf* must be a descriptor for the archive itself, not a member within the archive. *offset* gives the byte offset from the beginning of the archive to the archive header of the desired member. See *elf_getarsym*(3E) for more information about archive member offsets. When **elf_rand** works, it returns *offset*. Otherwise it returns 0, because an error occurred, *elf* was null, or the file was not an archive (no archive member can have a zero offset). A program may mix random and sequential archive processing.

EXAMPLES

An archive starts with a "magic string" that has **SARMAG** bytes; the initial archive member follows immediately. An application could thus provide the following function to rewind an archive (the function returns -1 for errors and 0 otherwise).

```
#include <ar.h>
#include <libelf.h>

int
rewindelf(Elf *elf)
{
        if (elf_rand(elf, (size_t)SARMAG) == SARMAG)
                return 0;
        return -1;
}
```

SEE ALSO

ar(4), elf(3E), elf_begin(3E), elf_getarsym(3E), elf_next(3E).

NAME

elf_rawfile - retrieve uninterpreted file contents

SYNOPSIS

cc [*flag*...] *file*... **-lelf** [*library*] ...

```
#include <libelf.h>
char *elf_rawfile(Elf *elf, size_t *ptr);
```

DESCRIPTION

elf_rawfile returns a pointer to an uninterpreted byte image of the file. This function should be used only to retrieve a file being read. For example, a program might use **elf_rawfile** to retrieve the bytes for an archive member.

A program may not close or disable (see *elf_cntl*(3E)) the file descriptor associated with *elf* before the initial call to **elf_rawfile**, because **elf_rawfile** might have to read the data from the file if it does not already have the original bytes in memory. Generally, this function is more efficient for unknown file types than for object files. The library implicitly translates object files in memory, while it leaves unknown files unmodified. Thus asking for the uninterpreted image of an object file may create a duplicate copy in memory.

elf_rawdata (see *elf_getdata*(3E)) is a related function, providing access to sections within a file.

If *ptr* is non-null, the library also stores the file's size, in bytes, in the location to which *ptr* points. If no data is present, *elf* is null, or an error occurs, the return value is a null pointer, with zero optionally stored through *ptr*.

NOTICES

A program that uses **elf_rawfile** and that also interprets the same file as an object file potentially has two copies of the bytes in memory. If such a program requests the raw image first, before it asks for translated information (through such functions as **elf_getehdr**, **elf_getdata**, and so on), the library "freezes" its original memory copy for the raw image. It then uses this frozen copy as the source for creating translated objects, without reading the file again. Consequently, the application should view the raw file image returned by **elf_rawfile** as a read-only buffer, unless it wants to alter its own view of data subsequently translated. In any case, the application may alter the translated objects without changing bytes visible in the raw image.

Multiple calls to **elf_rawfile** with the same ELF descriptor return the same value; the library does not create duplicate copies of the file.

SEE ALSO

elf(3E), elf_begin(3E), elf_cntl(3E), elf_getdata(3E), elf_getehdr(3E), elf_getident(3E), elf_kind(3E).

NAME

elf_strptr - make a string pointer

SYNOPSIS

cc [*flag*...] *file*... **-lelf** [*library*] ...

```
#include <libelf.h>
char *elf_strptr(Elf *elf, size_t section, size_t offset);
```

DESCRIPTION

This function converts a string section *offset* to a string pointer. *elf* identifies the file in which the string section resides, and *section* gives the section table index for the strings. **elf_strptr** normally returns a pointer to a string, but it returns a null pointer when *elf* is null, *section* is invalid or is not a section of type **SHT_STRTAB**, the section data cannot be obtained, *offset* is invalid, or an error occurs.

EXAMPLES

A prototype for retrieving section names appears below. The file header specifies the section name string table in the **e_shstrndx** member. The following code loops through the sections, printing their names.

```
if ((ehdr = elf32_getehdr(elf)) == 0)
{
        /* handle the error */
        return;
}
ndx = ehdr->e_shstrndx;
scn = 0;
while ((scn = elf_nextscn(elf, scn)) != 0)
{
        char    *name = 0;
        if ((shdr = elf32_getshdr(scn)) != 0)
            name = elf_strptr(elf, ndx,
                              (size_t)shdr->sh_name);
        printf("'%s'\n", name? name: "(null)");
}
```

NOTICES

A program may call **elf_getdata** to retrieve an entire string table section. For some applications, that would be both more efficient and more convenient than using **elf_strptr**.

SEE ALSO

elf(3E), elf_getdata(3E), elf_getshdr(3E), elf_xlate(3E).

NAME

elf_update - update an ELF descriptor

SYNOPSIS

cc [*flag*...] *file*... **-lelf** [*library*] ...

```
#include <libelf.h>
off_t elf_update(Elf *elf, Elf_Cmd cmd);
```

DESCRIPTION

elf_update causes the library to examine the information associated with an ELF descriptor, *elf*, and to recalculate the structural data needed to generate the file's image.

cmd may have the following values.

ELF_C_NULL This value tells **elf_update** to recalculate various values, updating only the ELF descriptor's memory structures. Any modified structures are flagged with the **ELF_F_DIRTY** bit. A program thus can update the structural information and then reexamine them without changing the file associated with the ELF descriptor. Because this does not change the file, the ELF descriptor may allow reading, writing, or both reading and writing (see *elf_begin*(3E)).

ELF_C_WRITE If *cmd* has this value, **elf_update** duplicates its **ELF_C_NULL** actions and also writes any "dirty" information associated with the ELF descriptor to the file. That is, when a program has used **elf_getdata** or the **elf_flag** facilities to supply new (or update existing) information for an ELF descriptor, those data will be examined, coordinated, translated if necessary (see *elf_xlate*(3E)), and written to the file. When portions of the file are written, any **ELF_F_DIRTY** bits are reset, indicating those items no longer need to be written to the file (see *elf_flag*(3E)). The sections' data is written in the order of their section header entries, and the section header table is written to the end of the file.

When the ELF descriptor was created with **elf_begin**, it must have allowed writing the file. That is, the **elf_begin** command must have been either **ELF_C_RDWR** or **ELF_C_WRITE**.

If **elf_update** succeeds, it returns the total size of the file image (not the memory image), in bytes. Otherwise an error occurred, and the function returns -1.

When updating the internal structures, **elf_update** sets some members itself. Members listed below are the application's responsibility and retain the values given by the program.

	Member	Notes
ELF Header	e_ident[EI_DATA]	Library controls other e_ident values
	e_type	
	e_machine	
	e_version	
	e_entry	
	e_phoff	Only when ELF_F_LAYOUT asserted
	e_shoff	Only when ELF_F_LAYOUT asserted
	e_flags	
	e_shstrndx	

	Member	Notes
Program Header	p_type	The application controls all program header entries
	p_offset	
	p_vaddr	
	p_paddr	
	p_filesz	
	p_memsz	
	p_flags	
	p_align	

	Member	Notes
Section Header	sh_name	
	sh_type	
	sh_flags	
	sh_addr	
	sh_offset	Only when ELF_F_LAYOUT asserted
	sh_size	Only when ELF_F_LAYOUT asserted
	sh_link	
	sh_info	
	sh_addralign	Only when ELF_F_LAYOUT asserted
	sh_entsize	

	Member	Notes
Data Descriptor	d_buf	
	d_type	
	d_size	
	d_off	Only when ELF_F_LAYOUT asserted
	d_align	
	d_version	

Note the program is responsible for two particularly important members (among others) in the ELF header. The **e_version** member controls the version of data structures written to the file. If the version is **EV_NONE**, the library uses its own internal version. The **e_ident[EI_DATA]** entry controls the data encoding used in the file. As a special case, the value may be **ELFDATANONE** to request the native data encoding for the host machine. An error occurs in this case if the native encoding doesn't match a file encoding known by the library.

Further note that the program is responsible for the **sh_entsize** section header member. Although the library sets it for sections with known types, it cannot reliably know the correct value for all sections. Consequently, the library relies on the program to provide the values for unknown section type. If the entry size is unknown or not applicable, the value should be set to zero.

When deciding how to build the output file, **elf_update** obeys the alignments of individual data buffers to create output sections. A section's most strictly aligned data buffer controls the section's alignment. The library also inserts padding between buffers, as necessary, to ensure the proper alignment of each buffer.

NOTICES

As mentioned above, the **ELF_C_WRITE** commands translate data as necessary, before writing them to the file. This translation is *not* always transparent to the application program. If a program has obtained pointers to data associated with a file (for example, see *elf_getehdr*(3E) and *elf_getdata*(3E)), the program should reestablish the pointers after calling **elf_update**.

As *elf_begin*(3E) describes, a program may "update" a COFF file to make the image consistent for ELF. (COFF is an object file format that preceded ELF on some computer architectures (Intel, for example). When a program calls elf_begin on a COFF file, the library translates COFF structures to their ELF equivalents, allowing programs to read (but not to write) a COFF file as if it were ELF. This conversion happens only to the memory image and not to the file itself.) The **ELF_C_NULL** command updates only the memory image; one can use the **ELF_C_WRITE** command to modify the file as well. Absolute executable files (**a.out** files) require special alignment, which cannot normally be preserved between COFF and ELF. Consequently, one may not update an executable COFF file with the **ELF_C_WRITE** command (though **ELF_C_NULL** is allowed).

SEE ALSO

elf(3E), elf_begin(3E), elf_flag(3E), elf_fsize(3E), elf_getdata(3E), elf_getehdr(3E), elf_getshdr(3E), elf_xlate(3E).

NAME

elf_version - coordinate ELF library and application versions

SYNOPSIS

cc [*flag*...] *file*... **-lelf** [*library*]...

```
#include <libelf.h>
unsigned elf_version(unsigned ver);
```

DESCRIPTION

As *elf*(3E) explains, the program, the library, and an object file have independent notions of the "latest" ELF version. **elf_version** lets a program determine the ELF library's *internal version*. It further lets the program specify what memory types it uses by giving its own *working version*, *ver*, to the library. Every program that uses the ELF library must coordinate versions as described below.

The header file **libelf.h** supplies the version to the program with the macro **EV_CURRENT**. If the library's internal version (the highest version known to the library) is lower than that known by the program itself, the library may lack semantic knowledge assumed by the program. Accordingly, **elf_version** will not accept a working version unknown to the library.

Passing *ver* equal to **EV_NONE** causes **elf_version** to return the library's internal version, without altering the working version. If *ver* is a version known to the library, **elf_version** returns the previous (or initial) working version number. Otherwise, the working version remains unchanged and **elf_version** returns **EV_NONE**.

EXAMPLES

The following excerpt from an application program protects itself from using an older library.

```
if (elf_version(EV_CURRENT) == EV_NONE)
{
     /* library out of date */
     /* recover from error */
}
```

NOTICES

The working version should be the same for all operations on a particular elf descriptor. Changing the version between operations on a descriptor will probably not give the expected results.

SEE ALSO

elf(3E), elf_begin(3E), elf_xlate(3E).

NAME

elf32_xlatetof, elf32_xlatetom, elf64_xlatetof, elf64_xlatetom - class-dependent data translation for elf32 and elf64 files, respectively

SYNOPSIS

cc [*flag*...] *file*... **-lelf** [*library*] ...

```
#include <libelf.h>
Elf_Data *elf32_xlatetof(Elf_Data *dst, const Elf_Data *src,
                         unsigned encode);
Elf_Data *elf32_xlatetom(Elf_Data *dst, const Elf_Data *src,
                         unsigned encode);
Elf_Data *elf64_xlatetof(Elf_Data *dst, const Elf_Data *src,
                         unsigned encode);
Elf_Data *elf64_xlatetom(Elf_Data *dst, const Elf_Data *src,
                         unsigned encode);
```

DESCRIPTION

elf32_xlatetom translates various data structures from their 32-bit class file representations to their memory representations; **elf64_xlatetom** translates various data structures from their 64-bit class file representations to their memory representations; **elf32_xlatetof** and **elf64_xlatetof** provide the inverse. This conversion is particularly important for cross development environments. *src* is a pointer to the source buffer that holds the original data; *dst* is a pointer to a destination buffer that will hold the translated copy. *encode* gives the byte encoding in which the file objects are (to be) represented and must have one of the encoding values defined for the ELF header's **e_ident[EI_DATA]** entry (see *elf_getident*(3E)). If the data can be translated, the functions return *dst*. Otherwise, they return null because an error occurred, such as incompatible types, destination buffer overflow, and so forth.

elf_getdata(3E) describes the Elf_Data descriptor, which the translation routines use as follows.

d_buf Both the source and destination must have valid buffer pointers.

d_type This member's value specifies the type of the data to which **d_buf** points and the type of data to be created in the destination. The program supplies a **d_type** value in the source; the library sets the destination's **d_type** to the same value. These values are summarized below.

d_size This member holds the total size, in bytes, of the memory occupied by the source data and the size allocated for the destination data. If the destination buffer is not large enough, the routines do not change its original contents. The translation routines reset the destination's **d_size** member to the actual size required, after the translation occurs. The source and destination sizes may differ.

d_version This member holds version number of the objects (desired) in the buffer. The source and destination versions are independent.

Translation routines allow the source and destination buffers to coincide. That is, **dst->d_buf** may equal **src->d_buf**. Other cases where the source and destination buffers overlap give undefined behavior.

For elf32 files:

Elf_Type	32-Bit Memory Type
ELF_T_ADDR	Elf32_Addr
ELF_T_BYTE	unsigned char
ELF_T_DYN	Elf32_Dyn
ELF_T_EHDR	Elf32_Ehdr
ELF_T_HALF	Elf32_Half
ELT_T_OFF	Elf32_Off
ELF_T_PHDR	Elf32_Phdr
ELF_T_REL	Elf32_Rel
ELF_T_RELA	Elf32_Rela
ELF_T_SHDR	Elf32_Shdr
ELF_T_SWORD	Elf32_Sword
ELF_T_SYM	Elf32_Sym
ELF_T_WORD	Elf32_Word

For elf64 files:

Elf_Type	64-Bit Memory Type
ELF_T_ADDR	Elf64_Addr
ELF_T_BYTE	unsigned char
ELF_T_DYN	Elf64_Dyn
ELF_T_EHDR	Elf64_Ehdr
ELF_T_HALF	Elf64_Half
ELT_T_OFF	Elf64_Off
ELF_T_PHDR	Elf64_Phdr
ELF_T_REL	Elf64_Rel
ELF_T_RELA	Elf64_Rela
ELF_T_SHDR	Elf64_Shdr
ELF_T_SWORD	Elf64_Sword
ELF_T_SYM	Elf64_Sym
ELF_T_WORD	Elf64_Word

"Translating" buffers of type **ELF_T_BYTE** does not change the byte order.

SEE ALSO

elf(3E), elf_fsize(3E), elf_getdata(3E), elf_getident(3E).

NAME

end, etext, edata, __data_start, __text_start - last locations in program

SYNOPSIS

```
extern void *_end, *end, *_etext, *etext, *_edata, *edata, *__data_start,
            *__text_start;
```

DESCRIPTION

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

The address of the symbols **__data_start** is the beginning address of the program's data area, and **__text_start** is the beginning address of the program's text area.

The linker defines these symbols with the appropriate values if they are referenced by the program but not defined. The linker issues an error if the user attempts to define **_etext**, **_edata**, **_end**, **__data_start**, or **__text_start**.

When execution begins, the program break (the first location beyond the data) coincides with **_end**, but the program break can 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)).

WARNINGS

In C, these names must look like addresses. Thus, use **&end** instead of **end** to access the current value of *end*.

SEE ALSO

System Tools:

cc(1)	invoke the HP-UX C compiler
ld(1)	invoke the link editor

Miscellaneous:

brk(2)	change data segment space allocation
crt0(3)	execution startup routine
malloc(3C)	main memory allocator
stdio(3S)	standard buffered input/output stream file package

STANDARDS CONFORMANCE

end(): XPG2

edata(): XPG2

etext(): XPG2

NAME
endwin — suspend Curses session

SYNOPSIS
```
#include <curses.h>
int endwin(void);
```

DESCRIPTION
The **endwin()** function restores the terminal after Curses activity by at least restoring the saved shell terminal mode, flushing any output to the terminal and moving the cursor to the first column of the last line of the screen. Refreshing a window resumes program mode. The application must call **endwin()** for each terminal being used before exiting. If **newterm()** is called more than once for the same terminal, the first screen created must be the last one for which **endwin()** is called.

RETURN VALUE
Upon successful completion, **endwin()** returns OK. Otherwise, it returns ERR.

ERRORS
No errors are defined.

APPLICATION USAGE
The **endwin()** function does not free storage associated with a screen, so **delscreen()** should be called after **endwin()** if a particular screen is no longer needed.

To leave Curses mode temporarily, portable applications should call **endwin()**. Subsequently, to return to Curses mode, they should call **doupdate()**, **refresh()** or **wrefresh()**.

SEE ALSO
delscreen(3X), doupdate(3X), initscr(3X), isendwin(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The entry is rewritten for clarity. The argument list is explicitly declared as **void**.

NAME

erasechar, killchar — single-byte terminal environment query functions

SYNOPSIS

```
#include <curses.h>
char erasechar(void);
char killchar(void);
```

DESCRIPTION

The **erasechar()** function returns the current erase character. The **erasewchar()** function stores the current erase character in the object pointed to by *ch*. If no erase character has been defined, the function will fail and the object pointed to by *ch* will not be changed.

The **killchar()** function returns the current line kill character.

RETURN VALUE

The **erasechar()** function returns the erase character and **killchar()** returns the line kill character. The return value is unspecified when these characters are multi-byte characters.

ERRORS

No errors are defined.

APPLICATION USAGE

The **erasechar()** and **killchar()** functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the A_ prefix. Moreover, they do not reliably indicate cases in which when the erase or line kill character, respectively, has not been defined. The **erasewchar()** and **killwchar()** functions overcome these limitations.

SEE ALSO

Attributes in curses_intro(3X), clearok(3X), delscreen(3X), erasewchar(3X), <curses.h>, tcattribute(3C) (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification).

CHANGE HISTORY

First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4

The **killchar()** function is merged with this entry. In previous issues, it appeared in an entry of its own.

The entry is rewritten for clarity. The argument list for the **erasechar()** and **killchar()** functions is explicitly declared as **void**.

NAME

erasewchar, killwchar — terminal environment query functions

SYNOPSIS

```
#include <curses.h>
int erasewchar(wchar_t *ch);
int killwchar(wchar_t *ch);
```

DESCRIPTION

The **erasewchar()** function stores the current erase character in the object pointed to by *ch*. If no erase character has been defined, the function will fail and the object pointed to by *ch* will not be changed.

The **killwchar()** function stores the current line kill character in the object pointed to by *ch*. If no line kill character has been defined, the function will fail and the object pointed to by *ch* will not be changed.

RETURN VALUE

Upon successful completion, **erasewchar()** and **killwchar()** return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

SEE ALSO

Attributes in curses_intro(3X), clearok(3X), delscreen(3X), <curses.h>, tcattribute(3C) (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification).

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

erf(), erfc() - error function and complementary error function

SYNOPSIS

```
#include <math.h>
double erf(double x);
double erfc(double x);
```

DESCRIPTION

erf() returns the error function of *x*, defined as:

(2/**sqrt**(**pi**)) *
(integral with respect to *t* from 0 to *x* of (**exp**(**pow**(-*t*,2))))

erfc() returns the complementary value, 1.0 – **erf(***x***)**. It prevents the extreme loss of relative accuracy if **erf(***x***)** is called for a large *x* and the result is subtracted from 1.0 (for example, for *x* = 5, twelve decimal places are lost).

The ISO/ANSI C committee has approved the **erf()** and **erfc()** functions for inclusion in the C9X draft standard.

To use these functions, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

If *x* is +INFINITY, **erf()** returns 1.0.

If *x* is –INFINITY, **erf()** returns –1.0.

If *x* is +INFINITY, **erfc()** returns zero.

If *x* is –INFINITY, **erfc()** returns 2.0.

If *x* is NaN, **erf()** and **erfc()** return NaN.

If the correct value after rounding would be smaller in magnitude than **MINDOUBLE**, **erf()** and **erfc()** return zero.

ERRORS

No errors are defined.

SEE ALSO

exp(3M), pow(3M), sqrt(3M), math(5), values(5).

STANDARDS CONFORMANCE

erf(): SVID3, XPG4.2

erfc(): SVID3, XPG4.2

NAME

exp(), expf() - exponential functions

SYNOPSIS

```
#include <math.h>
double exp(double x);
float expf(float x);
```

DESCRIPTION

exp() returns e^x.

expf() is a **float** version of **exp()**; it takes a **float** argument and returns a **float** result. To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options.

expf() is not specified by any standard, but it is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, make sure your program includes <**math.h**>, and link in the math library by specifying **-lm** on the compiler or linker command line.

Millicode versions of the **exp()** function are available. Millicode versions of math library functions are usually faster than their counterparts in the standard library. To use these versions, compile your program with the **+Olibcalls** or the **+Oaggressive** optimization option.

If an error occurs, the millicode versions return the value described in the *RETURN VALUE* section, but do not set **errno**.

For more information, see the *HP-UX Floating-Point Guide*.

RETURN VALUE

If *x* is +INFINITY, **exp()** returns +INFINITY.

If *x* is –INFINITY, **exp()** returns zero.

If *x* is NaN, **exp()** returns NaN.

If the correct value after rounding would be smaller in magnitude than **MINDOUBLE**, **exp()** returns zero.

If the correct value would overflow, **exp()** returns **HUGE_VAL** and sets **errno** to [ERANGE].

ERRORS

If **exp()** fails, **errno** is set to the following value.

[ERANGE] The correct value would overflow.

SEE ALSO

cbrt(3M), exp2(3M), expm1(3M), log(3M), pow(3M), sqrt(3M), math(5), values(5).

STANDARDS CONFORMANCE

exp(): SVID3, XPG4.2, ANSI C

NAME

exp2( ) - base-2 exponential function

SYNOPSIS

```
#include <math.h>
double exp2(double x);
```

DESCRIPTION

exp2() returns 2^x.

The ISO/ANSI C committee has approved the **exp2()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

If *x* is +INFINITY, **exp2()** returns +INFINITY.

If *x* is –INFINITY, **exp2()** returns +zero.

If *x* is NaN, **exp2()** returns NaN.

If the correct value after rounding would be smaller in magnitude than **MINDOUBLE**, **exp2()** returns zero.

If the magnitude of *x* is too large, **exp2()** returns **+INFINITY**.

ERRORS

No errors are defined.

SEE ALSO

cbrt(3M), exp(3M), log(3M), pow(3M), sqrt(3M), math(5), values(5).

NAME

expm1() - computes exponential functions

SYNOPSIS

```
#include <math.h>
double expm1(double x);
```

DESCRIPTION

The **expm1()** function is equivalent to **exp(***x***)** - 1, but may be more accurate for very small values of *x*.

The **expm1()** and **log1p()** functions are useful to guarantee that financial calculations of (((1+*x*)**n)-1)/*x*, namely:

expm1(n * **log1p**(*x*))/*x*

are accurate when *x* is very small (for example, when calculating small daily interest rates).

The ISO/ANSI C committee has approved the **expm1()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

If *x* is +INFINITY, **expm1()** returns +INFINITY.

If *x* is –INFINITY, **expm1()** returns -1.0.

If *x* is NaN, **expm1()** returns NaN.

If the correct value after rounding would be smaller in magnitude than **MINDOUBLE**, **expm1()** returns zero.

If the value overflows, **expm1()** returns **HUGE_VAL** and sets **errno** to [ERANGE].

ERRORS

If **expm1()** fails, **errno** is set to the following value.

[ERANGE] The result overflows.

SEE ALSO

exp(3M), log1p(3M), math(5), values(5).

STANDARDS CONFORMANCE

expm1(): XPG4.2

NAME

exportent(), getexportent(), setexportent(), addexportent(), remexportent(), endexportent(), getexportopt() - access exported file system information

SYNOPSIS

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

FILE *setexportent();

struct exportent *getexportent(FILE *fildep);

int addexportent(FILE *filep, char *dirname, char *options);

int remexportent(FILE *filep, char *dirname);

char *getexportopt(struct exportent *xent, char *opt);

void endexportent(FILE *filep);
```

DESCRIPTION

These routines access the exported filesystem information in **/etc/xtab**.

setexportent() Open the export information file and return a file pointer to use with **getexportent()**, **addexportent()**, **remexportent()**, and **endexportent()**. Returns NULL if the file is locked or if an error is encountered in opening the file.

getexportent() Read the next line from *filep* and return a pointer to an object with the following structure containing the broken-out fields of a line in file **/etc/xtab**. The fields have meanings described in *exports*(4).

```
#define ACCESS_OPT "access" /* machines that can mount fs */
#define ROOT_OPT   "root"   /* machines with root access to fs */
#define RO_OPT     "ro"     /* export read-only */
#define ANON_OPT   "anon"   /* uid for anonymous requests */
#define ASYNC_OPT  "async" /* all mounts will be aynchronous */

struct exportent {
       char *xent_dirname; /* directory (or file) to export */
       char *xent_options; /* options, as above */
       };
```

getexportent() returns NULL if it encounters end of file.

addexportent() Add the *exportent* to the end of the open file *filep*. It returns 0 if successful and –1 on failure.

remexportent() Remove the indicated entry from the list. Returns 0 on success and –1 on failure.

getexportopt() Scans the **xent_options** field of the **exportent** structure for a substring that matches *opt*. Returns the string value of *opt*, or NULL if the option is not found.

endexportent() Close the file.

RETURN VALUE

setexportent(), **getexportent()**, and **getexportopt()** return a NULL pointer on EOF or error.

addexportent() and **remexportent()** return –1 if they fail.

WARNINGS

The returned **exportent** structure points to static information that is overwritten in each call.

AUTHOR

exportent, **getexportent()**, **setexportent()**, **addexportent()**, **remexportent()**, **endexportent()**, and **getexportopt()** were developed by Sun Microsystems, Inc.

FILES

`/etc/exports`
`/etc/xtab`

SEE ALSO

exportfs(1M), exports(4).

NAME
fabs(), fabsf() - absolute value functions

SYNOPSIS
```
#include <math.h>
double fabs(double x);
float fabsf(float x);
```

DESCRIPTION
The **fabs()** function returns the absolute value of *x*, $|x|$.

The **fabsf()** function is a **float** version of **fabs()**; it takes a **float** argument and returns a **float** result. To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options.

The **fabsf()** function is not specified by any standard, but it is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, make sure your program includes <**math.h**>, and link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE
If *x* is ±INFINITY, **fabs()** returns +INFINITY.

If if *x* is NaN, **fabs()** returns NaN.

If the correct value after rounding would be smaller in magnitude than **MINDOUBLE**, **fabs()** returns zero.

ERRORS
No errors are defined.

SEE ALSO
abs(3C), ceil(3M), floor(3M), fmod(3M), rint(3M), math(5), values(5).

STANDARDS CONFORMANCE
fabs(): SVID3, XPG4.2, ANSI C

NAME

fattach() - attach a STREAMS file descriptor to an object in the file system name space

SYNOPSIS

```
#include <stropts.h>
int fattach(int fd, const char *path);
```

DESCRIPTION

The **fattach()** function attaches the *fd* file descriptor to an object in the file system name space designated by *path*. *fd* specifies an open file descriptor to a STREAMS device or STREAMS-based pipe. *path* specifies the pathname of an existing object in the file system. A STREAMS device or pipe can be attached to more than one node in the file system name space. In other words, a STREAMS device or pipe is allowed to have several associated names. Until the STREAMS device or pipe is detached from the node (with *fdetach*(3C) or *fdetach*(1M)), all operations on *path* will act on the STREAMS device or pipe instead of the file system object *path*.

The fattached stream's attributes (see the *stat*(2) reference page) are set according to the following scheme:

- The group ID, user ID, times, and permissions are set to those of *path*.
- The size as well as the device number are set to those of the STREAMS device or pipe designated by the *fd* parameter. Note that although the attributes of the fattached STREAMS device or pipe may change (see the *chmod*(2) reference page), the attributes of the underlying file system object *path* will not be changed.
- The number of links is set to 1.

RETURN VALUE

Upon successful completion, the **fattach()** function returns a value of 0 (zero). Otherwise, it returns a value of -1, and **errno** is set to indicate the error.

ERRORS

If any of the following conditions occurs, the **fattach()** function sets **errno** to the value that corresponds to the condition.

[EACCES]	Although the user is the owner of *path*, the user has no write permissions for it.
[EBADF]	The *fd* parameter is an invalid file descriptor.
[EBUSY]	The existing object specified by the *path* parameter is already mounted or has a file descriptor attached to it.
[EFAULT]	The *path* parameter points to a location outside of the allocated address space of the process.
[EINVAL]	The *fd* parameter does not refer to a STREAMS device or STREAMS-based pipe.
[ELOOP]	When *path* was translated, too many symbolic links were found.
[ENOENT]	*path* does not exist.
[ENOTDIR]	The directory portion of the *path* parameter does not exist.
[ENAMETOOLONG]	The size of a pathname component is longer than NAME_MAX when _POSIX_NO_TRUNC is in effect, or the pathname length is longer than PATH_MAX.
[EPERM]	The current effective user ID is not the owner of the existing object specified by the *path* parameter.

SEE ALSO

fdetach(3C), isastream(3C), chmod(2), stat(2), fdetach(1M), streamio(7).

STANDARDS COMPLIANCE

fattach(): SVID3

NAME

fclose(), fflush() - close or flush a stream

SYNOPSIS

```
#include <stdio.h>
int fclose(FILE *stream);
int fflush(FILE *stream);
```

Obsolescent Interfaces

```
int fclose_unlocked(FILE *stream);
int fflush_unlocked(FILE *stream);
```

DESCRIPTION

fclose() causes any buffered data for the named *stream* to be written out, and the *stream* to be closed. Buffers allocated by the standard input/output system may be freed.

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

If *stream* points to an output stream or an update stream in which the most recent operation was output, **fflush()** causes any buffered data for the *stream* to be written to that file; otherwise any buffered data is discarded. The *stream* remains open.

If *stream* is a null pointer, **fflush()** performs this flushing action on all currently open streams.

Obsolescent Interfaces

fclose_unlocked() and **fflush_unlocked()** close or flush a stream.

APPLICATION USAGE

fclose() and **fflush()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing **fclose()** or **fflush()**.

RETURN VALUE

Upon successful completion, **fclose()** and **fflush()** return 0. Otherwise, they return EOF and set **errno** to indicate the error.

ERRORS

fclose(), **fclose_unlocked()**, **fflush()**, and **fflush_unlocked()** fail if:

[EAGAIN]	The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.
[EBADF]	The file descriptor underlying *stream* is not valid.
[EFBIG]	An attempt was made to write a file that exceeds the process's file size limit or the maximum file size (see *ulimit*(2)).
[EINTR]	**fclose()** or **fflush()** was interrupted by a signal.
[EIO]	The process is in a background process group and is attempting to write to its controlling terminal, **TOSTOP** is set, the process is neither ignoring nor blocking the **SIGTTOU** signal, and the process group of the process is orphaned.
[ENOSPC]	There was no free space remaining on the device containing the file.
[EPIPE]	An attempt was made to write to a pipe that is not open for reading by any process. A **SIGPIPE** signal is also sent to the process.

Additional **errno** values may be set by the underlying **write()**, **lseek()**, and **close()** functions (see *write*(2), *lseek*(2) and *close*(2)).

WARNINGS

fclose_unlocked() and **fflush_unlocked()** are obsolescent interfaces supported only for compatibility with existing DCE applications. New multithreaded applications should use **fclose()** and **fflush()**.

SEE ALSO
close(2), exit(2), lseek(2), write(2), flockfile(3S), fopen(3S), setbuf(3S).

STANDARDS CONFORMANCE
`fclose()`: AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
`fflush()`: AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME
fdetach() - detach a name from a STREAMS file descriptor

SYNOPSIS
```
#include <stropts.h>
int fdetach(const char *path);
```

DESCRIPTION
The **fdetach()** function detaches a file descriptor from a name in the file system designated by *path*. *path* specifies the pathname of an existing object in the file system name space that was previously attached (see the **fattach()** reference page). As a result of the **fdetach()** operation, the node's status and permissions return to the state prior to the file attaching to the node. Any later operations on *path* will affect only the file system node and not the attached file. The user must either be superuser or own *path* and have write permission.

RETURN VALUE
Upon successful completion, the **fdetach()** function returns a value of 0 (zero). Otherwise, it returns a value of -1 is returned, and **errno** is set to indicate the error.

ERRORS
If any of the following conditions occurs, the **fdetach()** function sets **errno** to the value that corresponds to the condition.

[EFAULT]	The *path* parameter points outside the process' allocated address space.
[EACCES]	The user is not the owner of the file or does not have the correct permissions to access the file.
[EINVAL]	The object pointed to by the *path* parameter is not fattached to a STREAMS device or pipe.
[ELOOP]	When *path* was translated, too many symbolic links were found.
[ENOENT]	The *path* parameter points to a pathname that does not exist.
[ENOTDIR]	The directory portion of the *path* parameter does not exist.
[ENAMETOOLONG]	The size of a pathname component is longer than [NAME_MAX] when [_POSIX_NO_TRUNC] is in effect, or the pathname length is longer than [PATH_MAX].
[EPERM]	The current effective user ID is not the owner of the existing object specified by the *path* parameter, or the current effective user ID does not specify a user with the correct permissions.

SEE ALSO
fattach(3C), isastream(3C), umount(3), fdetach(1M), streamio(7).

STANDARDS COMPLIANCE
fdetach(): SVID3

NAME
fdim() - positive difference function

SYNOPSIS
```
#include <math.h>
double fdim(double x, double y);
```

DESCRIPTION
The **fdim()** function determines the positive difference between its arguments.

The ISO/ANSI C committee has approved the **fdim()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

f

RETURN VALUE
The **fdim()** function returns the positive difference between *x* and *y*.

If *x* **>** *y*, **fdim()** returns *x* **-** *y*.

If *x* **<=** *y*, **fdim()** returns +zero.

If *x* or *y* is NaN, **fdim()** returns the same NaN.

If both arguments are NaNs, **fdim()** returns one of the arguments.

If the correct value would overflow, **fdim()** returns +INFINITY.

ERRORS
No errors are defined.

SEE ALSO
fmax(3M), fmin(3M), math(5).

NAME

feclearexcept() - clear floating-point exception flags

SYNOPSIS

```
#include <fenv.h>

void feclearexcept(int excepts);
```

DESCRIPTION

The **feclearexcept()** function clears the exception flags represented by its argument. The argument can be constructed as a bitwise OR of the exception macros: **FE_INEXACT**, **FE_DIVBYZERO**, **FE_UNDERFLOW**, **FE_OVERFLOW**, and **FE_INVALID**. **FE_ALL_EXCEPT** represents all the exceptions.

The ISO/ANSI C committee has approved the **feclearexcept()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**fenv.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

For more information, see the *HP-UX Floating-Point Guide*.

RETURN VALUE

None.

ERRORS

No errors are defined.

EXAMPLES

Clear the underflow and inexact exception flags:

```
#include <fenv.h>
/*...*/
   feclearexcept(FE_UNDERFLOW | FE_INEXACT);
```

Clear all exception flags:

```
#include <fenv.h>
/*...*/
   feclearexcept(FE_ALL_EXCEPT);
```

SEE ALSO

fegetexceptflag(3M), fegettrapenable(3M), feraiseexcept(3M), fesetexceptflag(3M), fesettrapenable(3M), fetestexcept(3M), fenv(5).

NAME

fegetenv( ) - get floating-point environment

SYNOPSIS

```
#include <fenv.h>
void fegetenv(fenv_t *envp);
```

DESCRIPTION

The **fegetenv()** function stores the current floating-point environment in the object pointed to by the argument *envp*.

The ISO/ANSI C committee has approved the **fegetenv()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**fenv.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

For more information, see the *HP-UX Floating-Point Guide*.

RETURN VALUE

None.

ERRORS

No errors are defined.

EXAMPLE

Store the current floating-point environment:

```
#include <fenv.h>
/*...*/
   fenv_t env;
   /*...*/
   fegetenv(&env);
```

SEE ALSO

feholdexcept(3M), fesetenv(3M), feupdateenv(3M), fenv(5).

NAME

fegetexceptflag() - get floating-point exception flags

SYNOPSIS

```
#include <fenv.h>
void fegetexceptflag(fexcept_t *flagp, int excepts);
```

DESCRIPTION

The **fegetexceptflag()** function stores the exception flags indicated by the argument *excepts* in the object pointed to by the argument *flagp*. The *excepts* argument can be constructed as a bitwise OR of the exception macros: **FE_INEXACT**, **FE_DIVBYZERO**, **FE_UNDERFLOW**, **FE_OVERFLOW**, and **FE_INVALID**. **FE_ALL_EXCEPT** represents all the exceptions.

Use **fetestexcept()** to determine which exception flags are set.

The ISO/ANSI C committee has approved the **fegetexceptflag()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**fenv.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

For more information, see the *HP-UX Floating-Point Guide*.

RETURN VALUE

None.

ERRORS

No errors are defined.

EXAMPLE

Store the overflow and invalid exception flags:

```
#include <fenv.h>
/*...*/
   fexcept_t flags;
   /*...*/
   fegetexceptflag(&flags, FE_OVERFLOW | FE_INVALID);
```

SEE ALSO

feclearexcept(3M), fegettrapenable(3M), feraiseexcept(3M), fesetexceptflag(3M), fesettrapenable(3M), fetestexcept(3M), fenv(5).

NAME

fegetflushtozero() - get floating-point underflow mode

SYNOPSIS

```
#include <fenv.h>
int fegetflushtozero(void);
```

DESCRIPTION

The **fegetflushtozero()** function retrieves the value representing the current underflow mode, which is either IEEE-754-compliant (gradual) underflow mode or flush-to-zero mode.

The default underflow mode is IEEE-754-compliant.

Flush-to-zero mode, also known as fast underflow mode, is supported on most PA1.1 systems and on all PA2.0 systems. On HP 9000 systems, most underflow cases are supported by trapping into the kernel, where the IEEE-mandated conversion of the result into a denormalized value or zero is accomplished by software emulation. Flush-to-zero mode causes the hardware to substitute a zero for the result of an operation, with no fault occurring. This may be a significant performance optimization for applications that underflow frequently. Flush-to-zero mode also causes denormalized floating-point operands to be treated as if they were true zero operands.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**fenv.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

For more information, see the *HP-UX Floating-Point Guide*.

RETURN VALUE

The **fegetflushtozero()** function returns zero if the current underflow mode is IEEE-754-compliant. The function returns 1 if the current underflow mode is flush-to-zero.

On systems that do not support flush-to-zero mode, this function returns an undefined value.

ERRORS

No errors are defined.

EXAMPLE

Save the current underflow mode, set flush-to-zero mode, and restore the previous mode.

```
#include <fenv.h>
/*...*/
   int fm_saved;

   fm_saved = fegetflushtozero();
   fesetflushtozero(1);
   /*...*/
   fesetflushtozero(fm_saved);
```

AUTHOR

fegetflushtozero() was developed by HP and is not required by any current standard.

SEE ALSO

fesetflushtozero(3M), fenv(5).

NAME

fegetround() - get floating-point rounding direction mode

SYNOPSIS

```
#include <fenv.h>
int fegetround(void);
```

DESCRIPTION

The **fegetround()** function gets the current rounding direction.

The default rounding direction mode is round to nearest (**FE_TONEAREST**).

The ISO/ANSI C committee has approved the **fegetround()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**fenv.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

For more information, see the *HP-UX Floating-Point Guide*.

RETURN VALUE

The **fegetround()** function returns the value of the rounding direction macro representing the current rounding direction. The return value will match one of the macros **FE_TONEAREST**, **FE_UPWARD**, **FE_DOWNWARD**, and **FE_TOWARDZERO**.

ERRORS

No errors are defined.

EXAMPLE

Save, set, and restore the rounding direction.

```
#include <fenv.h>
/*...*/
{
   int save_round;

   save_round = fegetround();
   fesetround(FE_UPWARD);
   /*...*/
   fesetround(save_round);
   /*...*/
}
```

SEE ALSO

fesetround(3M), fenv(5).

NAME

fegettrapenable( ) - get exception trap enable bits

SYNOPSIS

```
#include <fenv.h>
int fegettrapenable(void);
```

DESCRIPTION

The **fegettrapenable( )** function determines which exception trap enable bits are currently set.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**fenv.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

For more information, see the *HP-UX Floating-Point Guide*.

f

RETURN VALUE

The **fegettrapenable( )** function returns the bitwise OR of the exception macros corresponding to the currently set exception trap enable bits. The macros are **FE_INEXACT**, **FE_DIVBYZERO**, **FE_UNDERFLOW**, **FE_OVERFLOW**, and **FE_INVALID**. **FE_ALL_EXCEPT** represents all the exceptions.

ERRORS

No errors are defined.

EXAMPLE

Retrieve the current trap settings and determine whether a trap for the divide by zero exception is enabled.

```
#include <fenv.h>
/*...*/
   if (fegettrapenable() & FE_DIVBYZERO)
      printf("divide by zero trap set\\n");
```

AUTHOR

fegettrapenable( ) was developed by HP and is not required by any current standard.

SEE ALSO

feclearexcept(3M), fegetexceptflag(3M), feraiseexcept(3M), fesetexceptflag(3M), fesettrapenable(3M), fenv(5).

NAME

feholdexcept() - save floating-point environment

SYNOPSIS

```
#include <fenv.h>
int feholdexcept(fenv_t *envp);
```

DESCRIPTION

The **feholdexcept()** function saves the current floating-point environment in the object pointed to by the argument *envp*, clears the exception flags, and disables all traps.

Use **feholdexcept()** with **feupdateenv()** to hide spurious exceptions. Use it with **fesetenv()** to hide all exceptions.

The ISO/ANSI C committee has approved the **feholdexcept()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**fenv.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

For more information, see the *HP-UX Floating-Point Guide*.

RETURN VALUE

The **feholdexcept()** function returns a nonzero value, indicating the successful disabling of any traps.

ERRORS

No errors are defined.

EXAMPLE

Store the current floating-point environment in **holdenv**, hide spurious underflow exceptions, and continue on any exceptions that occur until the call to **feupdateenv()** is encountered.

```
#include <fenv.h>
/*...*/
   fenv_t holdenv;

   feholdexcept(&holdenv);
   /* perform operations */
   if (/* test for spurious underflow */)
     feclearexcept(FE_UNDERFLOW);
   feupdateenv(&holdenv); /* raise accumulated exceptions */
```

SEE ALSO

fegetenv(3M), fesetenv(3M), feupdateenv(3M), fenv(5).

NAME

feraiseexcept() - raise floating-point exceptions

SYNOPSIS

```
#include <fenv.h>

void feraiseexcept(int excepts);
```

DESCRIPTION

The **feraiseexcept()** function raises the exceptions represented by its argument. The argument can be constructed as a bitwise OR of the exception macros: **FE_INEXACT**, **FE_DIVBYZERO**, **FE_UNDERFLOW**, **FE_OVERFLOW**, and **FE_INVALID**. **FE_ALL_EXCEPT** represents all the exceptions.

Any traps enabled for the specified exceptions will be taken.

The ISO/ANSI C committee has approved the **feraiseexcept()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**fenv.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

For more information, see the *HP-UX Floating-Point Guide*.

RETURN VALUE

None.

ERRORS

No errors are defined.

EXAMPLES

Raise the underflow and inexact exceptions:

```
#include <fenv.h>
/*...*/
   feraiseexcept(FE_UNDERFLOW | FE_INEXACT);
```

Raise all exceptions:

```
#include <fenv.h>
/*...*/
   feraiseexcept(FE_ALL_EXCEPT);
```

SEE ALSO

feclearexcept(3M), fegetexceptflag(3M), fegettrapenable(3M), fesetexceptflag(3M), fesettrapenable(3M), fetestexcept(3M), fenv(5).

NAME

ferror(), feof(), clearerr() - stream status inquiries

SYNOPSIS

```
#include <stdio.h>
int ferror(FILE *stream);
int feof(FILE *stream);
void clearerr(FILE *stream);
```

Obsolescent Interfaces

```
int ferror_unlocked(FILE *stream);
int feof_unlocked(FILE *stream);
void clearerr_unlocked(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. Unless cleared by **clearerr()**, or unless the specific *stdio* routine so indicates, the error indication lasts until the stream is closed.

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 on the named *stream* to zero.

Obsolescent Interfaces

ferror_unlocked(), **feof_unlocked()**, and **clearerr_unlocked()** stream status inquiries.

APPLICATION USAGE

feof(), **ferror()** and **clearerr()** are thread-safe. These interfaces are not async-cancel-safe.

WARNINGS

All these routines are implemented both as library functions and as macros. The macro versions, which are used by default, are defined in <**stdio.h**>. To obtain the library function, either use a **#undef** to remove the macro definition or, if compiling in ANSI-C mode, enclose the function name in parentheses or use the function address. The following example illustrates each of these methods :

```
#include <stdio.h>
#undef ferror
     ...
main()
{
     int (*find_error()) ();
          ...
     return_val=ferror(fd);
          ...
     return_val=(feof)(fd1);
          ...
     find_error = feof;
};
```

Reentrant Interfaces

If **_REENTRANT** is defined before including <**stdio.h**>, the locked versions of the library functions for **ferror()**, **feof()**, and **clearerr()** are used by default.

ferror_unlocked(), feof_unlocked() and **clearerr_unlocked()** are obsolescent interfaces supported only for compatibility with existing DCE applications. New multithreaded applications should use **ferror()**, **feof()** and **clearerr()**.

SEE ALSO

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

STANDARDS CONFORMANCE

ferror(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

clearerr(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

feof(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

f

NAME

fesetenv() - set floating-point environment

SYNOPSIS

```
#include <fenv.h>

void fesetenv(const fenv_t *envp);
```

DESCRIPTION

The **fesetenv()** function establishes the floating-point environment represented by the object pointed to by *envp*. The argument *envp* must point to an object set by a call to **fegetenv()** or **feholdexcept()**, or equal the macro **FE_DFL_ENV**.

Note that **fesetenv()** merely installs the state of the exception flags represented through its argument, and does not raise these exceptions.

The ISO/ANSI C committee has approved the **fesetenv()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**fenv.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

For more information, see the *HP-UX Floating-Point Guide*.

RETURN VALUE

None.

ERRORS

No errors are defined.

EXAMPLES

Store the current floating-point environment, continue on exceptions, then restore the previous environment without raising the accumulated exceptions.

```
#include <fenv.h>
/*...*/
   fenv_t holdenv;

   feholdexcept(&holdenv);
     /* perform operations */
   fesetenv(&holdenv);
```

Restore the default environment:

```
#include <fenv.h>
/*...*/
   fesetenv(FE_DFL_ENV);
```

SEE ALSO

fegetenv(3M), feholdexcept(3M), feupdateenv(3M), fenv(5).

NAME

fesetexceptflag( ) - set floating-point exception flags

SYNOPSIS

```
#include <fenv.h>

void fesetexceptflag(const fexcept_t *flagp, int excepts);
```

DESCRIPTION

The **fesetexceptflag()** function sets the status for the exception flags indicated by the argument *excepts* according to the representation in the object pointed to by *flagp*. The value of **flagp* must have been set by a previous call to **fegetexceptflag()**; otherwise, the effect on the indicated exception flags is undefined. This function does not raise exceptions, but only sets the state of the flags. The *excepts* argument can be constructed as a bitwise OR of the exception macros: **FE_INEXACT**, **FE_DIVBYZERO**, **FE_UNDERFLOW**, **FE_OVERFLOW**, and **FE_INVALID**. **FE_ALL_EXCEPT** represents all the exceptions.

f

The ISO/ANSI C committee has approved the **fesetexceptflag()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**fenv.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

For more information, see the *HP-UX Floating-Point Guide*.

RETURN VALUE

None.

ERRORS

No errors are defined.

EXAMPLE

Use **fegetexceptflag()** to save the current state of two exception flags. Later, use **fesetexceptflag()** to restore the saved state.

```
#include <fenv.h>
/*...*/
   fexcept_t saved_flags;
   /*...*/
   fegetexceptflag(&saved_flags, FE_DIVBYZERO | FE_INEXACT);
   /*...*/
   fesetexceptflag(&saved_flags, FE_DIVBYZERO | FE_INEXACT);
```

SEE ALSO

feclearexcept(3M), fegetexceptflag(3M), fegettrapenable(3M), feraiseexcept(3M), fesettrapenable(3M), fetestexcept(3M), fenv(5).

NAME

fesetflushtozero( ) - set floating-point underflow mode

SYNOPSIS

```
#include <fenv.h>
void fesetflushtozero(int);
```

DESCRIPTION

The **fesetflushtozero()** function sets the current underflow mode. If the argument is 1, the underflow mode is set to flush-to-zero mode. If the argument is zero, the underflow mode is set to IEEE-754-compliant (gradual) underflow mode. For arguments other than 1 or zero, the effect is undefined.

The default underflow mode is IEEE-754-compliant.

Flush-to-zero mode, also known as fast underflow mode, is supported on most PA1.1 systems and on all PA2.0 systems. On HP 9000 systems, most underflow cases are supported by trapping into the kernel, where the IEEE-mandated conversion of the result into a denormalized value or zero is accomplished by software emulation. Flush-to-zero mode causes the hardware to substitute a zero for the result of an operation, with no fault occurring. This may be a significant performance optimization for applications that underflow frequently. Flush-to-zero mode also causes denormalized floating-point operands to be treated as if they were true zero operands.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**fenv.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

For more information, see the *HP-UX Floating-Point Guide*.

RETURN VALUE

None.

ERRORS

No errors are defined.

EXAMPLE

Save the current underflow mode, set flush-to-zero mode, and restore the previous mode.

```
#include <fenv.h>
/*...*/
   int fm_saved;

   fm_saved = fegetflushtozero();
   fesetflushtozero(1);
   /*...*/
   fesetflushtozero(fm_saved);
```

AUTHOR

fesetflushtozero() was developed by HP and is not required by any current standard.

SEE ALSO

fegetflushtozero(3M), fenv(5).

NAME

fesetround() - set floating-point rounding direction mode

SYNOPSIS

```
#include <fenv.h>
int fesetround(int round);
```

DESCRIPTION

The **fesetround()** function establishes the rounding direction represented by its argument *round*. The *round* argument must equal one of the macros **FE_TONEAREST**, **FE_UPWARD**, **FE_DOWNWARD**, and **FE_TOWARDZERO**. If the argument does not match a rounding direction macro, the rounding direction is not changed.

The default rounding direction mode is round to nearest (**FE_TONEAREST**).

The ISO/ANSI C committee has approved the **fesetround()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**fenv.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

For more information, see the *HP-UX Floating-Point Guide*.

RETURN VALUE

The **fesetround()** function returns a nonzero value if and only if the argument matches a rounding direction macro.

ERRORS

No errors are defined.

EXAMPLE

Save, set, and restore the rounding direction.

```
#include <fenv.h>
/*...*/
{
   int save_round;

   save_round = fegetround();
   fesetround(FE_UPWARD);
   /*...*/
   fesetround(save_round);
   /*...*/
}
```

SEE ALSO

fegetround(3M), fenv(5).

NAME

fesettrapenable() - set exception trap enable bits

SYNOPSIS

```
#include <fenv.h>
void fesettrapenable(int excepts);
```

DESCRIPTION

The **fesettrapenable()** function sets the exception trap enable bits indicated by the argument *excepts*. The argument can be constructed as a bitwise OR of the exception macros: **FE_INEXACT**, **FE_DIVBYZERO**, **FE_UNDERFLOW**, **FE_OVERFLOW**, and **FE_INVALID**. **FE_ALL_EXCEPT** represents all the exceptions.

The function also clears the trap enable bits for any exceptions not indicated by the argument *excepts*.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**fenv.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

For more information, see the *HP-UX Floating-Point Guide*.

RETURN VALUE

None.

ERRORS

No errors are defined.

EXAMPLE

Set a trap

```
#include <fenv.h>
/*...*/
   fesettrapenable(FE_OVERFLOW | FE_DIVBYZERO);
```

AUTHOR

fesettrapenable() was developed by HP and is not required by any current standard.

SEE ALSO

feclearexcept(3M), fegetexceptflag(3M), fegettrapenable(3M), feraiseexcept(3M), fesetexceptflag(3M), fetestexcept(3M), fenv(5).

NAME

fetestexcept() - test floating-point exceptions

SYNOPSIS

```
#include <fenv.h>
int fetestexcept(int excepts);
```

DESCRIPTION

The **fetestexcept()** function determines which of a specified subset of the exception flags are currently set. The *excepts* argument specifies the exception flags to be queried. The argument can be constructed as a bitwise OR of the exception macros: **FE_INEXACT**, **FE_DIVBYZERO**, **FE_UNDERFLOW**, **FE_OVERFLOW**, and **FE_INVALID**. **FE_ALL_EXCEPT** represents all the exceptions.

The ISO/ANSI C committee has approved the **fetestexcept()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**fenv.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

For more information, see the *HP-UX Floating-Point Guide*.

RETURN VALUE

The **fetestexcept()** function returns the bitwise OR of the exception macros corresponding to the currently set exceptions included in *excepts*.

ERRORS

No errors are defined.

EXAMPLE

Call **f()** if invalid is set, then **g()** if overflow is set:

```
#include <fenv.h>
/*...*/
   int set_excepts;
   /* operations that may raise exceptions */
   set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
   if (set_excepts & FE_INVALID) f();
   if (set_excepts & FE_OVERFLOW) g();
```

SEE ALSO

feclearexcept(3M), fegetexceptflag(3M), fegettrapenable(3M), feraiseexcept(3M), fesetexceptflag(3M), fesettrapenable(3M), fenv(5).

NAME

feupdateenv( ) - update floating-point environment

SYNOPSIS

```
#include <fenv.h>

void feupdateenv(const fenv_t *envp);
```

DESCRIPTION

The **feupdateenv()** function saves the current exceptions in its automatic storage, installs the floating-point environment represented through *envp*, and then raises the saved exceptions. The argument *envp* must point to an object set by a call to **fegetenv()** or **feholdexcept()**, or equal the macro **FE_DFL_ENV**.

The ISO/ANSI C committee has approved the **feupdateenv()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**fenv.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

For more information, see the *HP-UX Floating-Point Guide*.

RETURN VALUE

None.

ERRORS

No errors are defined.

EXAMPLE

Store the current floating-point environment in **holdenv**, hide spurious underflow exceptions, and continue on any exceptions that occur until the call to **feupdateenv()** is encountered.

```
#include <fenv.h>
/*...*/
   fenv_t holdenv;

   feholdexcept(&holdenv);
   /* perform operations */
   if (/* test for spurious underflow */)
     feclearexcept(FE_UNDERFLOW);
   feupdateenv(&holdenv); /* raise accumulated exceptions */
```

SEE ALSO

fegetenv(3M), feholdexcept(3M), fesetenv(3M), fenv(5).

NAME

fgetpos(), fsetpos() - save and restore a file position indicator for a stream

SYNOPSIS

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, const fpos_t *pos);
```

DESCRIPTION

fgetpos() Store the current value of the file position indicator for the stream pointed to by *stream* in the object pointed to by *pos*. The value stored contains information usable by **fsetpos()** for repositioning the stream to its position at the time of the call to **fgetpos()**.

fsetpos() Set the file position indicator for the stream pointed to by *stream* according to the value of the object pointed to by *pos*, which must be a value set by an earlier call to **fgetpos()** on the same stream.

A successful call to **fsetpos()** clears the end-of-file indicator for the stream and undoes any effects of *ungetc*(3S) on the same stream. After a **fsetpos()** call, the next operation on a update stream can be either input or output.

APPLICATION USAGE

fgetpos() and **fsetpos()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing **fgetpos()** and **fsetpos()**.

RETURN VALUE

If successful, these functions return zero; otherwise non-zero.

ERRORS

If **fgetpos()** fails, **errno** is set to one of the following values.

[EINVAL] The current value of the file position cannot be represented correctly in an object of size **fpos_t** in this environment.

Additional **errno** values may be set by the underlying **ftell()** function (see *fseek*(3S)).

WARNINGS

Failure can occur if these functions are used on a file that has not been opened via **fopen()**. In particular, they must not be used on a terminal or on a file opened via *popen*(3S).

fsetpos() has no effect on streams that are open for append (see *fopen*(3S)).

SEE ALSO

fgetpos64(3S), fseek(3S), fopen(3S), popen(3S), ungetc(3S).

STANDARDS CONFORMANCE

fgetpos(): AES, SVID3, XPG4, ANSI C

fsetpos(): AES, SVID3, XPG4, ANSI C

NAME

fgetpos64(), fopen64(), freopen64(), fseeko64(), fsetpos64(), fstatvfsdev64(), ftello64(), ftw64(), nftw64(), statvfsdev64(), tmpfile64() - non-POSIX standard API interfaces to support large files.

SYNOPSIS

```
#include <stdio.h>
int fgetpos64(FILE *stream, fpos64_t *pos);
#include <stdio.h>
FILE *fopen64(const char *pathname, const char *type);
#include <stdio.h>
FILE *freopen64(const char *pathname, const char *type, FILE *stream);
#include <stdio.h>
int fseeko64(FILE *stream, off64_t *offset, int whence);
#include <stdio.h>
int fsetpos64(FILE *stream, const fpos64_t *pos);
#include <sys/statvfs.h>
int fstatvfsdev64(int filedes, struct statvfs64 *buf);
#include <stdio.h>
off64_t ftello64(FILE *stream);
#include <ftw.h>
int ftw64(const char *path,
         int (*fn)(const char *obj_path,
                   const struct stat64 *obj_stat,
                   int obj_flags),
         int depth);
#include <ftw.h>
int nftw64(const char *path,
          int (*fn)(const char *obj_path,
                    const struct stat64 *obj_stat,
                    int obj_flags,
                    struct FTW obj_FTW),
          int depth,
          int flags);
#include <sys/statvfs.h>
int statvfsdev64(const char *path, struct statvfs64 *buf);
#include <stdio.h>
FILE *tmpfile64(void);
```

DESCRIPTION

New API's to support large files. These API interfaces are not a part of the POSIX standard and may be removed in the future.

fgetpos64() The **fgetpos64()** function is identical to **fgetpos()** except that **fgetpos64()** returns the position in a **fpos64_t** instead of a **fpos_t**. All other functional behaviors, returns, and errors are identical.

fopen64() The **fopen64()** function is identical to **fopen()** in 64-bit compile environment. The **fopen64()** function returns a pointer to a FILE which can be used to grow the file past 2 GB if desired. All other functional behaviors, returns, and errors are identical to **fopen()**.

freopen64() The **freopen64()** function is identical to **freopen()** in 64-bit compile environment. The **freopen64()** function returns a pointer to a FILE which can

be used to grow the file past 2 GB if desired. All other functional behaviors, returns, and errors are identical to **freopen()**.

fseeko64() The **fseeko64()** function is identical to **fseeko()** except that **fseeko64()** accepts an **off64_t** for the size parameter instead of **off_t**. All other functional behaviors, returns, and errors are identical.

fsetpos64() The **fsetpos64()** function is identical to **fsetpos()** except that **fsetpos64()** accepts an **fpos64_t** for the pos parameter instead of **fpos_t**. All other functional behaviors, returns, and errors are identical.

fstatvfsdev64() The **fstatvfsdev64()** function is identical to **fstatvfsdev()** except that **fstatvfsdev64()** accepts a **struct statvfs64** for the second parameter instead of **struct statvfs**. All other functional behaviors, returns, and errors are identical.

ftello64() The **ftello64()** function is identical to **ftello()** except that **ftello64()** returns the file position in an **off64_t** instead of an **off_t**. All other functional behaviors, returns, and errors are identical.

ftw64() The **ftw64()** function is identical to **ftw()** except that it that it utilizes a **struct stat64** as the second parameter to the function whose pointer is passed to **ftw64()**. All other functional behaviors, returns, and errors are identical.

nftw64() The **nftw64()** function is identical to **nftw()** except that it that it utilizes a **struct stat64** as the second parameter to the function whose pointer is passed to **nftw64()**. All other functional behaviors, returns, and errors are identical.

statvfsdev64() The **statvfsdev64()** function is identical to **statvfsdev()** except that **statvfsdev64()** accepts a **struct statvfs64** for the second parameter instead of **struct statvfs**. All other functional behaviors, returns, and errors are identical.

tmpfile64() The **tmpfile64()** function is identical to **tmpfile()** in the 64-bit compile environment. Both interfaces create a temporary file that is capable of growing beyond 2GB's if desired. All other functional behaviors, returns, and errors are identical.

APPLICATION USAGE

fgetpos64(), **fopen64()**, **freopen64()**, **fseeko64()**, **fsetpos64()**, **fstatvfsdev64()**, **ftello64()**, **ftw64()**, **nftw64()**, **statvfsdev64()**, and **tmpfile64()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing any of these interfaces.

NAME

fgetws() - get a wide character string from a stream file

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>
wchar_t *fgetws(wchar_t *ws, int n, FILE *stream);
```

Obsolescent Interface

```
wchar_t *fgetws_unlocked(wchar_t *ws, int n, FILE *stream);
```

Remarks:

fgetws() is compliant with the XPG4 Worldwide Portability Interface wide-character I/O functions. It parallels the 8-bit character I/O function defined in *gets*(3S).

DESCRIPTION

fgetws() Reads characters from the *stream*, converts them into corresponding wide characters, and places them into the array pointed to by *ws*, until *n* – 1 characters are read, a new-line character is read and transferred to *ws*, or an end-of-file condition is encountered. The wide string is then terminated with a null wide character.

The definition for this functions and the type **wchar_t** are provided in the <**wchar.h**> header.

Obsolescent Interface

fgetws_unlocked() get a wide character string from a stream file.

APPLICATION USAGE

fgetws() is a thread-safe interface. It is not async-cancel-safe. A cancellation point may occur when a thread is executing **fgetws()**.

EXTERNAL INFLUENCES

Locale

The **LC_CTYPE** category determines how wide character conversions are done.

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

Upon successful completion, **fgetws()** and **fgetws_unlocked()** return *ws*. If the stream is at end-of-file, the end-of-file indicator for the stream is set and a null pointer is returned. If a read error occurs, the error indicator for the stream is set, **errno** is set to indicate the error, and a null pointer is returned.

ferror() and **feof()** can be used to distinguish between an error condition and an end-of-file condition.

ERRORS

fgetws() and **fgetws_unlocked()** fail if data needs to be read into the *stream*'s buffer, and:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying *stream* and the process would be delayed in the read operation.

[EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for reading.

[EINTR] The read operation was terminated due to the receipt of a signal, and either no data was transferred or the implementation does not report partial transfer for this file.

[EIO] The process is a member of a background process and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the **SIGTTIN** signal or the process group of the process is orphaned.

[EILSEQ] The data obtained from the input stream do not form a valid wide character string.

Additional **errno** values can be set by the underlying **read()** function (see *read*(2)).

WARNINGS

fgetws_unlocked() is an obsolescent interface supported only for compatibility with existing DCE applications. New multithreaded applications should use **fgetws()**.

AUTHOR

fgetws() was developed by OSF and HP.

SEE ALSO

ferror(3S), flockfile(3S), fopen(3S), fread(3S), getwc(3C), putws(3C), scanf(3S).

STANDARDS COMPLIANCE

fgetws(): XPG4

f

NAME
fileno() - map stream pointer to file descriptor

SYNOPSIS
```
#include <stdio.h>

int fileno(FILE *stream);
```

Obsolescent Interfaces
```
int fileno_unlocked(FILE *stream);
```

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

The following symbolic values in <**unistd.h**> define the file descriptors associated with **stdin**, **stdout**, and **stderr** when a program is started :

STDIN_FILENO	Value of zero for standard input, **stdin**.
STDOUT_FILENO	Value of 1 for standard output, **stdout**.
STDERR_FILENO	Value of 2 for standard error, **stderr**.

Obsolescent Interfaces
fileno_unlocked() map stream pointer to file descriptor

APPLICATION USAGE
fileno() is thread-safe. It is not async-cancel-safe.

fileno_unlocked() is an obsolescent interface supported only for compatibility with existing DCE applications. New multithreaded applications should use **fileno()**.

RETURN VALUE
Upon error, **fileno()** and **fileno_unlocked()** return -1.

SEE ALSO
open(2), flockfile(3S), fopen(3S).

STANDARDS CONFORMANCE
fileno(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME
filter — disable use of certain terminal capabilities

SYNOPSIS
```
#include <curses.h>
void filter(void);
```

DESCRIPTION
The **filter()** function changes the algorithm for initialising terminal capabilities that assume that the terminal has more than one line. A subsequent call to **initscr()** or **newterm()** performs the following additional actions:

- Disable use of **clear**, **cud**, **cud1**, **cup**, **cuu1** and **vpa**.
- Set the value of the **home** string to the value of the **cr** string
- Set **lines** equal to 1.

Any call to **filter()** must precede the call to **initscr()** or **newterm()**.

RETURN VALUE
The **filter()** function does not return a value.

ERRORS
No errors are defined.

SEE ALSO
Defined Capabilities in terminfo(4), initscr(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 4.

NAME
flash — flash the screen

SYNOPSIS
```
#include <curses.h>
int flash(void);
```

DESCRIPTION
The **flash()** function alerts the user. It flashes the screen, or if that is not possible, it sounds the audible alarm on the terminal. If neither signal is possible, nothing happens.

RETURN VALUE
The **flash()** function always returns OK.

ERRORS
No errors are defined.

APPLICATION USAGE
Nearly all terminals have an audible alarm, but only some can flash the screen.

SEE ALSO
beep(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 4.

In previous issues, this function was included in the entry for **beep()**. It is moved to its own entry in X/Open Curses, Issue 4, the argument list is explicitly declared as **void**, and the *RETURN VALUE* section is changed to indicate that the function always returns OK.

NAME

flockfile(), ftrylockfile(), funlockfile() - explicit locking of streams within a multithread application

SYNOPSIS

```
#include <stdio.h>
void flockfile(FILE *stream);
int ftrylockfile(FILE *stream);
void funlockfile(FILE *stream);
```

DESCRIPTION

The **flockfile()**, **ftrylockfile()**, and **funlockfile()** functions provide for explicit application-level locking of streams. These functions can be used by a thread to delineate a sequence of I/O statements that are to be executed as a unit.

The **flockfile()** function is used by a thread to acquire ownership of a **(FILE *)** object.

The **ftrylockfile()** function is used by a thread to acquire ownership of a **(FILE *)** object if the object is available; **ftrylockfile()** is a non-blocking version of **flockfile()**.

The **funlockfile()** function is used to relinquish the ownership granted to the thread. The behavior is undefined if a thread other than the current owner calls the **funlockfile()** function.

Logically, there is a count associated with each stream. This count is implicitly initialized to zero when the stream is created. The stream is unlocked when the count is zero. When the count is positive, a single thread owns the stream. When the **flockfile()** function is called, if the count is zero or if the count is positive and the caller owns the stream, the count is incremented. Otherwise, the calling thread is suspended, waiting for the count to return to zero. Each call to **funlockfile()** decrements the count. This allows matching calls to **flockfile()** (or successful calls to **ftrylockfile()**) and **funlockfile()** to be nested.

All POSIX.1 and C standard functions that reference **(FILE *)** objects behave as if they use **flockfile()** and **funlockfile()** internally to obtain ownership of these **(FILE *)** objects.

RETURN VALUE

None for **flockfile()** and **funlockfile()**. The function **ftrylockfile()** returns zero for success and nonzero to indicate that the lock cannot be acquired.

NAME
floor() - floor function

SYNOPSIS
```
#include <math.h>
double floor(double x);
```

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

To use this function, make sure your program includes <**math.h**>, and link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE
If *x* is ±INFINITY or ±zero, **floor()** returns *x*.

If *x* is NaN, **floor()** returns NaN.

If the correct value would overflow, **floor()** returns −**HUGE_VAL** and sets **errno** to [ERANGE]. (This description complies with XPG4.2; however, the **floor()** function never overflows on HP-UX systems.)

ERRORS
If **floor()** fails, **errno** is set to the following value.

[ERANGE] The correct value would overflow.

SEE ALSO
ceil(3M), fabs(3M), fmod(3M), rint(3M), math(5).

STANDARDS CONFORMANCE
floor(): SVID3, XPG4.2, ANSI C

NAME
flushinp — discard input

SYNOPSIS
```
#include <curses.h>
int flushinp(void);
```

DESCRIPTION
The **flushinp()** function discards (flushes) any characters in the input buffer associated with the current screen.

RETURN VALUE
The **flushinp()** function always returns OK.

ERRORS
No errors are defined.

SEE ALSO
<curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The entry is rewritten for clarity. The argument list for the **flushinp()** function is explicitly declared as **void**.

NAME
fmax() - maximum value function

SYNOPSIS
```
#include <math.h>
double fmax(double x, double y);
```

DESCRIPTION
The **fmax()** function determines the maximum numeric value of its arguments.

The ISO/ANSI C committee has approved the **fmax()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE
The **fmax()** function returns the maximum numeric value of its arguments.

If one argument is a NaN and the other is numeric, **fmax()** returns the numeric argument.

If both arguments are NaNs, **fmax()** returns one of the arguments.

ERRORS
No errors are defined.

SEE ALSO
fdim(3M), fmin(3M), math(5).

NAME
fmin() - minimum value function

SYNOPSIS
```
#include <math.h>
double fmin(double x, double y);
```

DESCRIPTION
The **fmin()** function determines the minimum numeric value of its arguments.

The ISO/ANSI C committee has approved the **fmin()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

f

RETURN VALUE
The **fmin()** function returns the minimum numeric value of its arguments.

If one argument is a NaN and the other is numeric, **fmin()** returns the numeric argument.

If both arguments are NaNs, **fmin()** returns one of the arguments.

ERRORS
No errors are defined.

SEE ALSO
fdim(3M), fmax(3M), math(5).

NAME

fmod(), fmodf() - remainder functions

SYNOPSIS

```
#include <math.h>
double fmod(double x, double y);
float fmodf(float x, float y);
```

DESCRIPTION

The **fmod()** function returns the floating-point remainder (f) of the division of x by y, where f has the same sign as x, such that $x = iy + f$ for some integer i, and $|f| < |y|$.

fmodf() is a **float** version of **fmod()**; it takes **float** arguments and returns a **float** result. To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options.

fmodf() is not specified by any standard, but it is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, make sure your program includes <**math.h**>, and link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

If y is ±INFINITY and x is not ±INFINITY, **fmod()** returns x.

If x is ±zero and y is a nonzero number, **fmod()** returns x.

If x or y is NaN, **fmod()** returns NaN.

If the correct value after rounding would be smaller in magnitude than **MINDOUBLE**, **fmod()** returns zero.

If y is zero, **fmod()** returns NaN and sets **errno** to [EDOM].

If x is ±INFINITY, **fmod()** returns NaN and sets **errno** to [EDOM].

ERRORS

If **fmod()** fails, **errno** is set to one of the following values.

[EDOM]	y is zero.
[EDOM]	x is ±INFINITY.

SEE ALSO

ceil(3M), fabs(3M), floor(3M), remainder(3M), rint(3M), math(5), values(5).

STANDARDS CONFORMANCE

fmod(): SVID3, XPG4.2, ANSI C

NAME

fmtmsg() - displays formatted message on standard error and console

SYNOPSIS

```
#include <fmtmsg.h>

int fmtmsg(
     long class,
     const char *label,
     int severity,
     const char *text,
     const char *action,
     const char *tag
);
```

DESCRIPTION

The **fmtmsg()** routine is intended as a language-independent error message service. Messages are displayed on the system console, standard error, or both, depending on the setting of the *class* parameter. The format of messages is under the control of the user, who may specify how much of the message is to be displayed on standard error. However, the format of the message displayed to the system console is not under user control.

All messages specify a *class*, which indicates the source and status of the error condition, and where the message should be directed. The *class* may be functionally excluded by specifying **MM_NULLMC**. If this is done, no message is produced. The class types are:

Display	The display class has allowable values of **MM_PRINT**, which directs the message to standard error, **MM_CONSOLE**, which directs the message to the system console, or **MM_PRINT | MM_CONSOLE**, which directs the message to both.
Error Type	The error type class may be one of **MM_HARD**, indicating a hardware error, **MM_SOFT**, indicating a software failure, or **MM_FIRM**, which indicates a firmware error.
Error Source	The error source class may be one of **MM_APPL**, indicating an application, **MM_UTIL**, indicating an OS utility, or **MM_OPSYS**, indicating a system error.
Recovery Status	The recovery status class may be one of **MM_RECOVER**, indicating recovery from the error is possible, or **RMM_NRECOV**, stating no recovery is possible.

The remaining parameters are functionally optional, in that the message produced may exclude any or all of them. Each is discussed in parameter order.

The *label* component states where the error originated. It has the form *major:minor*, where *major* is a 10 character field specifying the major system producing the message and *minor* is a 14 character field specifying the subsystem in error. For example,

```
mail:send
```

This component can be excluded by setting it to **MM_NULLLBL** or **NULL**.

The *severity* component describes the degree of importance of the failure. The predefined severity levels are:

MM_NOSEV	No severity is specified and no print string is generated.
MM_INFO	The message is informational only and the **INFO** string is generated.
MM_WARNING	The situation might be in error, and should be checked. The **WARNING** string is generated.
MM_ERROR	An error has been detected. The **ERROR** string is generated.
MM_HALT	An unrecoverable error has been detected. The **HALT** string is generated.

The *text* component describes the nature of the failure in the most specific terms. It is expected to provide a sufficiently detailed account of the error to remove all doubt about its cause. It may be excluded by setting *text* to **MM_NULLTXT** or **MM_NULL**.

The *action* component describes the options available for recovery from the error. As output, it is prefaced with the string **TO FIX:**. It may be disabled by setting **action** to **MM_NULLACT** or **NULL**.

The *tag* component is intended to direct the user to the appropriate documentation to correct or avoid the error. It may be disabled by setting *tag* to **MM_NULLTXT** or **RMM_NULL**.

APPLICATION USAGE

fmtmsg() is thread-safe. It is not async-cancel-safe. A cancellation point may occur when a thread is executing **fmtmsg()**.

EXTERNAL INFLUENCES

The user may control the appearance of the message produced on standard error through the use of two environment variables: **MSGVERB** and **SEV_LEVEL**. These have no effect on the console message.

The **MSGVERB** environment variable describes the components the user is interested in seeing. The value of **MSGVERB** is a list of one or more components, separated by colons if more than one is specified. All components with non-NULL values listed in **MSGVERB** are displayed when **fmtmsg()** is called; all other components are excluded. Only the component names *label*, *severity*, *text*, *action*, and *tag* are valid list elements. Any others (or an empty list) are considered an error in **MSGVERB** format. If **MSGVERB** is erroneous or unset, **fmtmsg()** produces the full message.

The **SEV_LEVEL** environment variable gives users control over the text string displayed for the severity component if the severity is other than those described above. The predefined severity levels (**MM_NOSEV** through **MM_HALT**) cannot be overridden.

The value of **SEV_LEVEL** is a list of one or more level specifiers separated by colons. A level specifier is a three item comma list of the form *identifier*, *level*, *message*. The *identifier* is not used by **fmtmsg()**, and is only included for compatibility. It is not optional, however, as **fmtmsg()** expects it when examining level specifiers for the other two parts. The *level* is a number greater than four indicating the level defined. The *message* is a string to be displayed in the **severity** field, in the same manner as **HALT** and **INFO** are displayed for the **MM_HALT** and **MM_INFO** severities. Level specifiers having more or fewer than three items are invalid, as are null level specifiers. Invalid or null valued **SEV_LEVEL** lists have no impact on the behavior of **fmtmsg()**.

RETURN VALUE

The **fmtmsg()** routine returns one of the following values on exit:

MM_OK	Success.
MM_NOTOK	Failure.
MM_NOMSG	Failure to standard error, success to the system console.
MM_NOCON	Success to standard error, failure to the system console.

AUTHOR

fmtmsg() was developed by OSF and HP.

SEE ALSO

printf(3S).

STANDARDS COMPLIANCE

fmtmsg: SVID3

NAME

fnmatch() - match filename patterns

SYNOPSIS

```
#include <fnmatch.h>

int fnmatch(const char *pattern, const char *string, int flags);
```

DESCRIPTION

fnmatch() performs pattern matching as described in *regexp*(5) under *PATTERN MATCHING NOTATION*. By default, the rule qualifications for filename expansion do not apply; i.e., periods (dots) and slashes are matched as ordinary characters. This default behavior can be modified by using the flags described below.

The *flag* argument modifies the interpretation of *pattern* and *string*. If **FNM_PATHNAME**, which is defined in <**fnmatch.h**>, is set in *flag*, a slash character in *string* must be explicitly matched by a slash in *pattern;* it cannot be matched by either the asterisk or question mark special characters or by a bracket expression.

If **FNM_PERIOD** is set in *flag*, a leading period (**.**) must be explicitly matched. It will not be matched by a bracket expression, question mark or asterisk. By default, a period is leading if it is the first character in *string*. If **FNM_PATHNAME** is set in *flag*, a period is leading if it is the first character in *string* or immediately follows a slash.

If **FNM_NOESCAPE** is not set in *flag*, a backslash character (****) in *pattern* followed by any other character matches that second character in *string*. In particular, **** matches a backslash in *string*. If **FNM_NOESCAPE** is set, a backslash character is treated as an ordinary character.

If *flag* is zero, the slash character and the period are treated as regular characters. If *flag* has any other value, the result is undefined.

APPLICATION USAGE

fnmatch() is thread-safe and async-cancel-safe.

RETURN VALUE

If *string* matches the pattern specified by *pattern*, **fnmatch()** returns zero. Otherwise, **fnmatch()** returns non-zero.

EXAMPLE

The following excerpt uses **fnmatch()** to check each file in a directory against the pattern ***.c**:

```
pattern = "*.c";

while(dp = readdir(dirp)){
    if((fnmatch(pattern, dp->d_name,0)) == 0){
        /* do processing for match */
              ...
    }
}
```

AUTHOR

fnmatch() was developed by OSF and HP.

SEE ALSO

sh(1), glob(3C).

STANDARDS CONFORMANCE

fnmatch(): XPG4, POSIX.2

NAME

fopen(), freopen(), fdopen() - open or re-open a stream file; convert file to stream

SYNOPSIS

```
#include <stdio.h>
FILE *fopen(const char *pathname, const char *type);
FILE *freopen(const char *pathname, const char *type, FILE *stream);
FILE *fdopen(int fildes, const char *type);
```

DESCRIPTION

fopen() Opens the file named by *pathname* and associates a *stream* with it. **fopen()** returns a pointer to the **FILE** structure associated with the *stream*.

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* and makes an implicit call to **clearerr()** (see *ferror*(3S)).

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

fdopen() associates a stream with a file descriptor. File descriptors are obtained from **open()**, **dup()**, **creat()**, or **pipe()** (see *open*(2), *dup*(2), *creat*(2), and *pipe*(2)), which open files but do not return pointers to a **FILE** structure stream. Streams are necessary input for many of the Section (3S) library routines. The *type* of stream must agree with the mode of the open file. The meanings of *type* used in the **fdopen()** call are exactly as specified above, except that **w**, **w+**, **wb**, and **wb+** do not cause truncation of the file.

pathname Points to a character string containing the name of the file to be opened.

type Character string having one of the values listed below. The **b** in the following values has no effect. It exists to distinguish binary files from text files; however, there is no distinction between these types of files on UNIX systems (it is required for ISO C standard conformance).

r or **rb**	open file for reading
w or **wb**	truncate to zero length or create file for writing
a or **ab**	append; open file for writing at end of file, or create file or writing
r+, **rb+**, or **r+b**	open file for update (reading and writing)
w+, **wb+**, or **w+b**	truncate file to zero length or create file for update
a+, **ab+**, or **a+b**	append; open or create file for update at end-of-file

When a file is opened for update, both input and output can be done on the resulting *stream*. However, output cannot be directly followed by input without an intervening call to **fflush()** or to a file positioning function (**fseek()**, **fsetpos()**, or **rewind()**), and input cannot be directly followed by output without an intervening call to a file positioning function unless the input operation encounters end-of-file.

When a file is opened for append (i.e., when *type* is **a**, **a+**, **ab+**, or **a+b**), it is impossible to overwrite information already in the file. All output is written at the end of the file, regardless of intervening calls to **fseek()**. If two separate processes open the same file for append, each process can write freely to the file without fear of destroying output being written by the other. Output from the two processes will be intermixed in the file in the order in which it is written.

APPLICATION USAGE

fopen(), **fdopen()** and **freopen()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing these interfaces.

RETURN VALUE

Upon successful completion, **fopen()**, **fdopen()** and **freopen()** return a **FILE** * pointer to the stream. Otherwise, a null pointer is returned and **errno** is set to indicate the error.

ERRORS

fopen(), **fdopen()**, and **freopen()** fail if:

[EINVAL] The *type* argument is not a valid mode.

[ENOMEM] There is insufficient space to allocate a buffer.

fopen() and **freopen()** fail if:

[EACCES] Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by *type* are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.

[EINTR] A signal was caught during **fopen()** or **freopen()**. function.

[EISDIR] The named file is a directory and *type* requires write access.

[EMFILE] The calling process has attempted to exceed its open file limit.

[ENAMETOOLONG] The length of the *pathname* string exceeds **PATH_MAX** or a pathname component is longer than **NAME_MAX** while POSIX_NO_TRUNC is in effect.

[ENFILE] The system file table is full.

[ENOENT] The named file does not exist or the *pathname* argument points to an empty string.

[ENOSPC] The directory or file system that would contain the new file cannot be expanded, the file does not exist, and it was to be created.

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

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

[EOVERFLOW] The named file is a regular file and the size of the file cannot be represented correctly in an object of size **off_t** in this environment.

[EROFS] The named file resides on a read-only file system and *type* requires write access.

Additional **errno** values can be set by the underlying **open()** call made from the **fopen()** and **freopen()** functions (see *open*(2)).

NOTES

HP-UX binary file *types* are equivalent to their non-binary counterparts. For example, types **r** and **rb** are equivalent.

SEE ALSO

creat(2), dup(2), open(2), pipe(2), fclose(3S), fgetpos64(3S), fseek(3S), popen(3S), setbuf(3S).

STANDARDS CONFORMANCE

fopen(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

fdopen(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

freopen(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

fpclassify() - floating-point operand classification macro

SYNOPSIS

```
#include <math.h>
int fpclassify(floating-type x);
```

DESCRIPTION

The **fpclassify()** macro classifies its argument value as NaN, infinite, normalized, denormalized, or zero.

The **fpclassify()** macro can be used with either **double** or **float** arguments, and classifies the argument based on its type.

The ISO/ANSI C committee has approved the **fpclassify()** macro for inclusion in the forthcoming C9X draft standard. The **fpclassify()** macro implements the **class()** function recommended by the IEEE-754 standard for floating-point arithmetic.

To use the **fpclassify()** macro, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

The **fpclassify()** macro, used in conjunction with the **signbit()** macro, replaces the **fpclassify()** and **fpclassifyf()** functions, which are obsolete and are no longer supported.

RETURN VALUE

The **fpclassify()** macro returns the value of the number classification macro appropriate to the type and value of its argument.

The value returned is one of the following macros, which are defined in <**math.h**>:

FP_NORMAL	Normalized
FP_ZERO	Zero
FP_INFINITE	Infinity
FP_SUBNORMAL	Denormalized
FP_NAN	NaN

Every possible argument value falls into one of these categories, so these functions never result in an error.

ERRORS

No errors are defined.

EXAMPLE

Take certain actions if **x** is either a denormalized **float** value or zero:

```
#include <math.h>
/*...*/
   int class;
   float x;
   /*...*/
   class = fpclassify(x);
   if ( (class == FP_SUBNORMAL) || (class == FP_ZERO) ) {
      /*...*/
   }
```

SEE ALSO

isfinite(3M), isinf(3M), isnan(3M), isnormal(3M), signbit(3M), math(5).

NAME

fread(), fwrite() - buffered binary input/output to a stream file

SYNOPSIS

```
#include <stdio.h>

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

size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream);
```

Obsolescent Interfaces

```
size_t fread_unlocked(
    void *ptr,  size_t size,  size_t nitems,  FILE *stream);

size_t fwrite_unlocked(
    const void *ptr,  size_t size,  size_t nitems,  FILE *stream);
```

DESCRIPTION

fread() copies, into an array pointed to by *ptr*, up to *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*.

Obsolescent Interfaces

fread_unlocked() and **fwrite_unlocked()** buffered binary input/output to a stream file.

APPLICATION USAGE

fread() and **fwrite()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing **fread()** or **fwrite()**.

RETURN VALUE

fread(), **fread_unlocked()**, **fwrite()**, and **fwrite_unlocked()** return the number of items read or written. If *size* or *nitems* is 0, no characters are read or written and 0 is returned.

The value returned will be less than *nitems* only if a read error or end-of-file is encountered. The **ferror()** or **feof()** functions must be used to distinguish between an error condition and an end-of-file condition.

ERRORS

Refer to *getc*(3S) for a description of errors.

WARNINGS

fread_unlocked() and **fwrite_unlocked()** are obsolescent interfaces supported only for compatibility with existing DCE applications. New multithreaded applications should use **fread()** and **fwrite()**.

SEE ALSO

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

STANDARDS CONFORMANCE

fread(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

fwrite(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

frexp() - extract mantissa and exponent from double-precision number

SYNOPSIS

```
#include <math.h>
double frexp(double num, int *exp);
```

DESCRIPTION

The **frexp()** function breaks a floating-point number into a normalized fraction and an integral power of 2. It stores the integer exponent in the **int** object pointed to by *exp*.

To use this function, make sure your program includes <**math.h**>, and link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

The **frexp()** function returns the value *x*, such that *x* is a **double** with magnitude in the interval [0.5, 1] or zero, and *num* equals *x* times 2 raised to the power **exp*.

If *num* is zero, both parts of the result are zero.

If *num* is NaN, **frexp()** returns NaN, and the value of **exp* is unspecified.

If *num* is ±INFINITY, **frexp()** returns *num*, and the value of **exp* is unspecified.

ERRORS

No errors are defined.

SEE ALSO

ldexp(3M), modf(3M), scalb(3M), scalbn(3M), math(5).

STANDARDS CONFORMANCE

frexp(): SVID3, XPG4.2, ANSI C

NAME

fseek(), fseeko(), rewind(), ftell(), ftello() - reposition a file pointer in a stream

SYNOPSIS

```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int whence);
int fseeko(FILE *stream, off_t offset, int whence);
void rewind(FILE *stream);
long int ftell(FILE *stream);
off_t ftello(FILE *stream);
```

Obsolescent Interfaces

```
int fseek_unlocked(FILE *stream, long int offset, int whence);
void rewind_unlocked(FILE *stream);
long int ftell_unlocked(FILE *stream);
```

DESCRIPTION

fseek() sets the file-position indicator for *stream*. The new position, measured in bytes from the beginning of the file, is obtained by adding *offset* to the position specified by *whence*. The specified position is the beginning of the file for **SEEK_SET**, the current position for **SEEK_CUR**, or end-of-file for **SEEK_END**.

fseeko() is a non-POSIX standard API provided by the _LARGEFILE_SOURCE compile option. It is identical to the **fseek()** except that the *offset* parameter is an **off_t** instead of a **long int**. All other functional behaviors, returns, and errors are identical to the POSIX **fseek()**.

If the most recent operation, other than **ftell()**, on the *stream* is **fflush()**, the file offset in the underlying open file description is adjusted to reflect the location specified by the **fseek()**.

rewind(*stream***)** is equivalent to **fseek (stream, 0L, SEEK_SET)**, 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 can be either input or output. **fseek()** clears the EOF indicator for the *stream*. **rewind()** does an implicit **clearerr()** call (see *ferror*(3S)).

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

ftello() is a non-POSIX standard API provided by the _LARGEFILE_SOURCE compile option. It is identical to the **ftell()** except that it returns an **off_t** instead of a **long int**. All other behaviors, returns, and errors are identical to the POSIX **ftell()**.

Obsolescent Interfaces

fseek_unlocked(), **rewind_unlocked()**, and **ftell_unlocked()** reposition a file pointer in a stream.

APPLICATION USAGE

fseek(), **fseeko()**, **ftell()**, **ftello()** and **rewind()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing these interfaces.

RETURN VALUE

fseek() and **fseek_unlocked()** return zero if they succeed. Otherwise they return –1 and set **errno** to indicate the error.

ftell() and **ftell_unlocked()** return the current value of the file position indicator for the stream measured in bytes from the beginning of the file. Otherwise, **ftell()** and **ftell_unlocked()** return –1 and set **errno** to indicate the error.

rewind() and **rewind_unlocked()** do not return any value. Therefore, any application that needs to detect errors should clear **errno** before calling **rewind()** or **rewind_unlocked()**. Then, upon completion, if **errno** is non-zero, it should assume an error has occurred.

ERRORS

fseek(), **fseeko()**, **fseek_unlocked()**, **ftell()**, **ftello()**, **ftell_unlocked()**, **rewind()** and **rewind_unlocked()** fail if the *stream* is unbuffered or the buffered data needs to be flushed, or if any of the following conditions are encountered:

[EAGAIN] The O_NONBLOCK flag is set for the file descriptor and the process would be delayed in the write operation.

[EBADF] The underlying file is not open for writing.

[EFBIG] An attempt was made to write a file that exceeds the process's file size limit or the maximum file size. See *ulimit*(2).

[EINVAL] The file *offset* cannot be represented correctly in an object of type **long** or size **off_t** in this environment.

[EINTR] A signal was caught during the write operation.

[EIO] The process is in a background process group and is attempting to write to its controlling terminal, **TOSTOP** is set, the process is neither ignoring nor blocking the **SIGTTOU** signal, and the process group of the process is orphaned.

[ENOSPC] There was no free space remaining on the device containing the file.

[EPIPE] An attempt was made to write to a pipe that is not open for reading by any process. A **SIGPIPE** signal is also sent to the process.

[ESPIPE] A seek operation was attempted and the file descriptor underlying *stream* is associated with a pipe.

fseek() and **fseek_unlocked()** also fail if:

[EINVAL] The *whence* argument is invalid, or the file-position indicator would be set to a negative value.

Additional **errno** values may be set by the underlying **write()** and **lseek()** functions (see *write*(2) and *lseek*(2)).

WARNINGS

On HP-UX systems, the offset returned by **ftell()** or **ftell_unlocked()** is measured in bytes and it is permissible to seek to positions relative to that offset. However, when porting to non-HP-UX systems, **fseek()** should be used directly without relying on any offset obtained from **ftell()** because arithmetic cannot meaningfully be performed on such an offset if it is not measured in bytes on a particular operating system.

fseek(), **fseek_unlocked()**, **rewind()**, and **rewind_unlocked()** have no effect on streams that have been opened in append mode (see *fopen*(3S)).

fseek_unlocked(), **ftell_unlocked()** and **rewind_unlocked()** are obsolescent interfaces supported for compatibility with existing DCE applications. New multithreaded applications should use **fseek()**, **ftell()** and **rewind()**.

SEE ALSO

lseek(2), write(2), ferror(3S), flockfile(3S), fopen(3S), fgetpos(3S), fgetpos64(3S), ungetc(3S).

STANDARDS CONFORMANCE

fseek(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

ftell(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

rewind(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

ftok() - create interprocess communication identifier

SYNOPSIS

```
#include <sys/ipc.h>
key_t ftok(const char *path, int id);
```

DESCRIPTION

All interprocess communication facilities require the user to supply a key to be used by the **msgget()**, **semget()**, and **shmget()** system calls to obtain interprocess communication identifiers (see *msgget*(2), *semget*(2), and *shmget*(2)).

ftok() returns a key based on *path* and *id* that is usable in subsequent **msgget()**, **semget()**, and **shmget()** system calls.

The parameters for the **ftok()** function are as follows:

path	must be the path name of an existing file that is accessible to the process.
id	is a character that uniquely identifies a project. Note that **ftok()** returns the same key for linked files when called with the same *id* and that it returns different keys when called with the same file name but different *id*s.

Application Usage

ftok() is thread-safe and async-cancel-safe.

RETURN VALUE

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

EXAMPLES

The following call to **ftok**() returns a key associated with the file *myfile* and id **A**:

```
key_t mykey;i
      mykey = ftok ( myfile", 'A');"
```

WARNINGS

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, **ftok()** is likely to return a different key than it did the original time it was called.

SEE ALSO

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

NAME

ftw(), nftw(), nftw2() - walk a file tree executing a function

SYNOPSIS

```
#include <ftw.h>

int ftw (const char *path,
         int (*fn)(const char *obj_path,
                   const struct stat *obj_stat,
                   int obj_flags),
         int depth);

int nftw2(const char *path,
          int (*fn)(const char *obj_path,
                    const struct stat *obj_stat,
                    int obj_flags,
                    struct FTW obj_FTW),
          int depth,
          int flags);

int nftw (const char *path,
          int (*fn)(const char *obj_path,
                    const struct stat *obj_stat,
                    int obj_flags,
                    struct FTW obj_FTW),
          int depth,
          int flags);
```

UNIX95

```
int nftw (const char *path,
          int (*fn)(const char *obj_path,
                    const struct stat *obj_stat,
                    int obj_flags,
                    struct *FTW obj_FTW),
          int depth,
          int flags);
```

DESCRIPTION

The **ftw()** function 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 containing information about the object (see **lstat()** in *stat*(2)), and an integer. The possible values of the integer, defined in the **<ftw.h>** header file, are:

FTW_F The object is a file.

FTW_D The object is a directory.

FTW_SL The object is a symbolic link.

FTW_DNR The object is a directory without read permission. *fn* will not be called for any of its descendants.

FTW_NS **lstat()** failed on the object. The contents of the **stat** structure are undefined. If the **lstat()** failure is because the object is in a directory without search permission, *fn* is called and the walk continues. If **lstat()** fails for any other reason, **ftw()** does not call *fn*, it sets **errno**, and returns **-1**.

Tree traversal continues until the tree is exhausted, an invocation of *fn* returns a nonzero value, or an 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 the same value as returned by *fn*. If **ftw()** detects an error, it returns **-1** and sets the error type in **errno** (see *errno*(2)).

ftw() visits a directory before visiting any of its descendants.

ftw() and **nftw()** use one file descriptor for each level in the tree. The *depth* argument limits the number of file descriptors that can be used. If *depth* is 0 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. For best performance, *depth* should be at least as large as the number of levels in the tree.

nftw() is similar to **ftw()** except that it takes the additional argument *flags*. The *flags* field is the inclusive OR of the following values, as defined in the **<ftw.h>** header file:

FTW_PHYS **nftw()** does a physical walk. It does not follow symbolic links. **nftw()** follows hard links but does not walk down any path that crosses itself. If **FTW_PHYS** is not specified, **nftw()** follows symbolic and hard links but does not walk a path that crosses itself.

FTW_MOUNT The walk does not cross a mount point. This means the walk does not visit any files that reside on a device other than the one where the walk started. It does not cross NFS mount points.

FTW_DEPTH **nftw()** performs a depth-first search. This means that a directory's contents are visited before the directory itself is visited.

FTW_CHDIR The walk does a **chdir()** (see *chdir*(2)) to each directory before reading it.

FTW_SERR The walk normally terminates and returns **-1** if **lstat()** fails for any reason. If **FTW_SERR** is specified and an **lstat()** failure is encountered, *fn* is called, and the walk continues.

nftw() calls *fn* with four arguments for each file and directory visited. The first argument is the path name of the file or directory, the second is a pointer to a **stat** structure (see *lstat*(2)) containing information about the object, and the third is an integer giving additional information as follows:

FTW_F The object is a file.

FTW_D The object is a directory.

FTW_DP The object is a directory and subdirectories have been visited. This can be passed to *fn* only if **FTW_DEPTH** is specified.

FTW_SL The object is a symbolic link. This can be passed to *fn* only if **FTW_PHYS** is specified.

FTW_SLN The object is a symbolic link that does not point to an existing object. This can be passed to *fn* only if **FTW_PHYS** is not specified.

FTW_DNR The object is a directory that cannot be read. *fn* is not called for any of its descendants.

FTW_NS **lstat()** failed on the object. The contents of the **stat** structure passed to *fn* are undefined. If the **lstat()** failure occurred because the object is in a directory without search permission, **errno** is set, and **nftw()** returns **-1** after calling *fn*. Note that this behavior differs from **ftw()**. If **lstat()** fails for any other reason, **nftw()** does not call *fn*, it sets **errno**, and returns **-1**. This behavior can be altered by specifying the **FTW_SERR** *flag*.

The fourth argument is different for the default environment and the **UNIX95** environment. For the default environment, the fourth argument is a structure **FTW**. For the **UNIX95** environment, the fourth argument is a pointer to a structure **FTW** (ie: ***FTW**). **FTW** contains the following members:

```
int base;
int level;
```

The value of *base* is the offset from the first character in the path name to where the base name of the object starts; this path name is passed as the first argument to *fn*. The value of *level* indicates depth relative to the start of the walk, where the start level has a value of zero.

nftw2() is equivalent to **nftw()**. This function is provided for HP-UX compatibility in future releases.

APPLICATION USAGE

ftw(), **nftw()** and **nftw2()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing **ftw()**, **nftw()** or **nftw2()**.

To use the **UNIX95** prototype, the **UNIX95** environment must be defined. This is done by defining the **UNIX95** environment variable, passing the **_XOPEN_UNIX_EXTENDED** flag as a compiler option, and adding **/usr/xpg4/bin** to your path. This can be done as follows:

1. **export UNIX95=**
2. **PATH=/usr/xpg4/bin:$PATH**
3. **cc foo.c -D_XOPEN_SOURCE_EXTENDED**

ERRORS

If **ftw()** or **nftw()** fails, it sets **errno** (see *errno*(2)) to one of the following values:

[EACCES]	If a component of the *path* prefix denies search permission, or if read permission is denied for *path*, *fn* returns **-1** and does not reset **errno**.
[EINVAL]	The value of the *depth* argument is invalid.
[ENAMETOOLONG]	The length of the specified path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.
[ENOENT]	*path* points to the name of a file that does not exist, or points to an empty string.
[ENOTDIR]	A component of *path* is not a directory.
[EOVERFLOW]	One of the values in **struct stat** (st_size or st_blocks) is too large to store into the structure to be passed to the function pointed to by *fn*.

In addition, if the function pointed to by *fn* encounters system errors, **errno** may be set accordingly.

WARNINGS

Because these functions are recursive, it is possible for them to terminate with a memory fault when applied to very deep file structures.

ftw() and **nftw()** use **malloc()** to allocate dynamic storage during their operation (see *malloc*(3C)) If they are forcibly terminated (such as if **longjmp()** is executed by *fn* or an interrupt routine), the calling function will not have a chance to free that storage, causing it to remain allocated until the process terminates. 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.

The syntax for **nftw()** may be changed in a future release. (See **nftw2()** in the DESCRIPTION section).

AUTHOR

ftw(), **nftw()**, and **nftw2()** were developed by AT&T and HP.

SEE ALSO

stat(2), fgetpos64(3C), malloc(3C).

STANDARDS CONFORMANCE

ftw(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

NAME

get_expiration_time() - add a specific time interval to the current absolute system time

SYNOPSIS

```
#include <time.h>

int get_expiration_time(
    struct timespec *delta,
    struct timespec *abstime

);
```

DESCRIPTION

The **get_expiration_time()** function adds a specific time interval to the current absolute system time and returns the new absolute time. This new absolute time is used as the expiration time in a call to *pthread_cond_timedwait*(3T).

The *delta* argument represents the number of seconds and nanoseconds to add to the current system time. On return from this function, the *abstime* argument contains the absolute system time that will be used in a call to *pthread_cond_timedwait*(3T).

Parameters

delta Number of seconds and nanoseconds to add to the current system time.

abstime Output parameter for the absolute system time after adding *delta* to the current absolute system time.

RETURN VALUE

Upon successful completion, **get_expiration_time()** returns zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

If any of the following occur, the **get_expiration_time()** function returns the corresponding error number:

[EINVAL] The value specified by *delta* or *abstime* is invalid.

AUTHOR

get_expiration_time() was developed by X/Open.

SEE ALSO

pthread_cond_timedwait(3T).

STANDARDS CONFORMANCE

get_expiration_time() : X/Open.

(ENHANCED CURSES)

NAME

get_wch, mvget_wch, mvwget_wch, wget_wch — get a wide character from a terminal

SYNOPSIS

```
#include <curses.h>
int get_wch(wint_t *ch);
int mvget_wch(int y, int x, wint_t *ch);
int mvwget_wch(WINDOW *win, int y, int x, wint_t *ch);
int wget_wch(WINDOW *win, wint_t *ch);
```

DESCRIPTION

These functions read a character from the terminal associated with the current or specified window. If **keypad()** is enabled, these functions respond to the pressing of a function key by setting the object pointed to by *ch* to the corresponding KEY_ value defined in **<curses.h>** and returning KEY_CODE_YES.

Processing of terminal input is subject to the general rules described in *Input Processing* in *curses_intro*(3X).

If echoing is enabled, then the character is echoed as though it were provided as an input argument to **ins_wch()**, except for the following characters:

<backspace>, <left-arrow> and the current erase character:	The input is interpreted as specified in **specialchars** and then the character at the resulting cursor position is deleted as though **delch()** were called, except that if the cursor was originally in the first column of the line, then the user is alerted as though **beep()** were called.
Function keys	The user is alerted as though **beep()** were called. Information concerning the function keys is not returned to the caller.

If the current or specified window is not a pad, and it has been moved or modified since the last refresh operation, then it will be refreshed before another character is read.

RETURN VALUE

When these functions successfully report the pressing of a function key, they return KEY_CODE_YES. When they successfully report a wide character, they return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

Applications should not define the escape key by itself as a single-character function.

When using these functions, **nocbreak()** mode and **echo()** mode should not be used at the same time. Depending on the state of the terminal when each character is typed, the application may produce undesirable results.

SEE ALSO

Input Processing in curses_intro(3X), beep(3X), cbreak(3X), ins_wch(3X), keypad(3X), move(3X), <curses.h>, <wchar.h> (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification).

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

(ENHANCED CURSES)

NAME

getbegyx, getmaxyx, getparyx — get additional cursor and window coordinates

SYNOPSIS

```
#include <curses.h>
void getbegyx(WINDOW *win, int y, int x);
void getmaxyx(WINDOW *win, int y, int x);
void getparyx(WINDOW *win, int y, int x);
```

DESCRIPTION

The **getbegyx()** macro stores the absolute screen coordinates of the specified window's origin in *y* and *x*.

The **getmaxyx()** macro stores the number of rows of the specified window in *y* and stores the window's number of columns in *x*.

The **getparyx()** macro, if the specified window is a subwindow, stores in *y* and *x* the coordinates of the window's origin relative to its parent window. Otherwise, –1 is stored in *y* and *x*.

RETURN VALUE

No return values are defined.

ERRORS

No errors are defined.

APPLICATION USAGE

These interfaces are macros and '**&**' cannot be used before the *y* and *x* arguments.

SEE ALSO

getyx(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

getbootpent(), putbootpent(), setbootpent(), endbootpent(), parse_bp_htype(), parse_bp_haddr(), parse_bp_iaddr() - get or put bootptab entry

SYNOPSIS

```
#include <bootpent.h>

int getbootpent (struct bootpent **bootpent);

int setbootpent (const char *path);

int endbootpent (void);

int putbootpent (
     struct bootpent *bootpent,
     int numfields,
     FILE * bootpfile
);

int parse_bp_htype (const char *source);

int parse_bp_haddr (
     char **source,
     int htype,
     unsigned char *result,
     unsigned int *bytes
);

int parse_bp_iaddr (
     char **source,
     unsigned long *result
);
```

Remarks

These functions reside in libdc.a, and are linked using the **-ldc** option to the **ld** or **cc** command.

DESCRIPTION

These functions help a program read or modify a bootptab (**bootpd** control) file one entry at a time. **getbootpent()** locates an entry in the **/etc/bootptab** file, or an alternate file specified to **setbootpent()**, and returns a pointer to an array of objects of type **struct bootpent** that breaks the entry into separate data fields with preceding, or embedded, comment (text) lines.

The **bootpent** structure is defined in <**bootpent.h**> and includes the following members:

```
int   bp_type;   /* BP_DATA, BP_COMMENT, BP_BLANK  */
char  *bp_text;  /* one field or one comment line  */
```

The file also defines the following data type and constants:

```
typedef struct bootpent *bpp_t;

#define BP_NULLP  ((bpp_t) 0)
#define BP_SIZE   (sizeof (struct bootpent))

#define MAXHADDRLEN          6

#define HTYPE_UNKNOWN        0   /* 0 bytes  */
#define HTYPE_ETHERNET       1   /* 6 bytes  */
#define HTYPE_EXP_ETHERNET   2   /* 1 byte   */
#define HTYPE_AX25           3   /* 0 bytes  */
#define HTYPE_PRONET         4   /* 1 byte   */
#define HTYPE_CHAOS          5   /* 0 bytes  */
#define HTYPE_IEEE802        6   /* 6 bytes  */
#define HTYPE_ARCNET         7   /* 0 bytes  */

#define MAXHTYPES            7
```

The fields are described in the *Field Definitions* section below. The purpose of each function is as follows.

getbootpent() When first called, **getbootpent()** returns a pointer to, and the number of elements in, an array of **bootpent** structures. The array holds the first entry

in the **/etc/bootptab** file (or from an alternate file specified by a call to **setbootpent()**), including leading, or embedded, comment lines. Each subsequent call returns a pointer to the next entry in the file so that successive calls can be used to search the entire file.

If no file is currently in memory, **getbootpent()** reads the **/etc/bootptab** file prior to doing its work.

The returned array exists in static space (malloc'd memory) overwritten by the next call (so previously returned pointers become invalid). However, each array element's **bp_text** pointer points to text in an in-memory copy of the file. This text is not altered by the next call (nor by changes to the file itself). Hence, it is possible to copy an entry's array in order to save it, as illustrated in EXAMPLES below. The data remains valid until the next call of **setbootpent()** or **endbootpent()**.

If there are comments after the last entry, they are returned as a separate entry with no data parts.

setbootpent() Opens the specified file for reading by **getbootpent()**, reads a copy into memory, and closes the file (which as a side-effect releases any locks on the file; see *lockf*(2)). If the given *path* is a null pointer or a null string, **setbootpent()** opens and reads **/etc/bootptab**.

If the last file opened by **setbootpent()** (or implicitly by **getbootpent()**) was **/etc/bootptab**, a subsequent call to **setbootpent()** for the same file rewinds the file to the beginning, making visible any recent changes to the file, without first requiring a call to **endbootpent()**.

endbootpent() Frees the in-memory copy of the last file opened by **setbootpent()**, or **getbootpent()**.

putbootpent() Writes (to the current location in the stream specified by *bootpfile*) the ASCII equivalent of the specified array of **bootpent** structures containing one file entry, and its leading, or embedded, comments (a total of *numfields* array elements). Entries are written in canonical form, meaning the entry name and each data field are on separate lines, data fields are preceded by one tab each, and each line except the last ends with ":\". If *numfields* is less than or equal to zero, nothing is written.

parse_bp_htype() Converts a host address type from string to numeric format (**HTYPE_***) in the same manner as **bootpd**.

parse_bp_haddr() Converts a host (hardware, link level) address from string to binary format in the same manner as **bootpd** given a host address type (**HTYPE_***). The calling program's *result*, which must be an array containing at least MAXHADDRLEN elements, is modified to hold the host address binary value, and **bytes** is modified to indicate the length in bytes of the resulting address. This can be used to compare two host addresses, independent of string representations. *source* is modified to point to the first char after the parsed address.

parse_bp_iaddr() Converts an internet address from string to binary format in the same manner as **bootpd**. This can be used to compare two internet addresses, independent of string representations. The calling program's *result* is modified to hold the internet address binary value. *source* is modified to point to the first char after the parsed address.

Field Definitions

If **bootpent.bp_type** is BP_DATA, the associated text is one field from the current entry, either the name field or one of the tag fields. Null tag fields (two colons in a row) are ignored, not returned.

If **bootpent.bp_type** is BP_COMMENT or BP_BLANK, the associated text is one comment line or blank line from the file, either preceding the current entry or embedded in it following a data line that was continued with a backslash. The text is exactly as it appears in the file, including any whitespace appearing on a blank line, and there is no trailing newline.

The returned array elements are in the same order as data fields and comment lines appear in the file.

Entry field strings are of the form:

tag[**@**][**="***value***"**]

with surrounding whitespace, if any, removed (see *bootpd*(1M) for the full description). Double quotes, and backslashes, can appear anywhere in the field strings.

Template entries (those referred to by other entries using **tc** fields) are not special. They can be managed like other entries. It is the calling program's responsibility to correctly manage the order of fields, **tc** fields, and "@" fields that override earlier field values.

RETURN VALUE

getbootpent() returns the number of valid array elements (one or more) upon successful completion. At the end of the input file it returns zero. If it cannot open or close the file it returns -1. If it encounters a memory allocation or map error, or a read error, it returns -2.

setbootpent() returns zero if successful opening and reading the specified or default file. If it cannot open or close the file it returns -1. If it encounters a memory allocation or map error or a read error it returns -2.

endbootpent() returns zero if successful freeing the memory for the current open file. If there is no current file it returns -1. If it cannot free the memory for the current file it returns -2.

putbootpent() returns zero if successful writing an entry to the specified file, with the **ferror()** indication clear (see *ferror*(3S)). Otherwise it returns non-zero with **ferror()** set.

In all cases above, if a failure is due to a failed system call, the **errno** value from the system is valid on return from the called function.

parse_bp_htype() returns HTYPE_UNKNOWN if the hardware type string is unrecognized.

parse_bp_haddr() returns zero if successful, otherwise non-zero in case of parsing error, invalid *htype*, or a host address type for which the address length is unknown; *source* is modified to point to the first illegal char (a NUL if the string is too short). The caller's *bytes* value is unmodified, but *result* might be changed.

parse_bp_iaddr() returns zero if successful, otherwise non-zero, and *source* is modified to point to the first illegal char (a NUL if the string is null).

EXAMPLES

The following code fragment copies all data and comments from **/etc/bootptab** to a temporary copy of the file. It converts data entries to canonical form as a side effect, and prints to standard output the first field of each entry copied (should be the field name, assuming the entry doesn't start with a continuation line).

```
#include <bootpent.h>

FILE *newfilep;   /* to write temp file */
bpp_t bp;         /* read from file */
int field;        /* current field number */
int fields;       /* total in array for one entry */

if ((newfilep = fopen ("/tmp/bootptab", "w")) == (FILE *) NULL)
{
    (handle error)
}

while ((fields = getbootpent (&bp)) > 0)
{
    for (field = 0; field < fields; ++field)
    {
        if ((bp[field].bp_type) == BP_DATA)
        {
            (void) puts (bp[field].bp_text);
            break;
        }
    }
```

```
        if (putbootpent (bp, fields, newfilep))
        {
            (handle error)
        }
    }

    if (fields < 0)    /* error reading file */
    {
        (handle error)
    }

    if (endbootpent())
    {
        (handle error)
    }

    if (fclose (newfilep))
    {
        (handle error)
    }
```

The following code fragment saves a copy of a bootptab entry returned by **getbootpent()**.

```
#include <malloc.h>
#include <string.h>
#include <bootpent.h>

bpp_t bpnew;
unsigned size;

size = fields *BP_SIZE;

if ((bpnew = (bpp_t) malloc (size)) == BP_NULLP)
{
    (handle error)
}

(void) memcpy ((void *)bpnew, (void *)bp, size);
```

WARNINGS

These functions are unsafe in multi-thread applications.

Calling **setbootpent()** releases any locks on the file it opens.

AUTHOR

These functions were developed by HP.

FILES

/etc/bootptab control file for **bootpd**

SEE ALSO

bootpd(1M), errno(2), lockf(2), ferror(3S), fopen(3S), malloc(3C).

NAME

getc(), getc_unlocked(), getchar(), getchar_unlocked(), fgetc(), getw() - get character or word from a stream file

SYNOPSIS

```
#include <stdio.h>
int getc(FILE *stream);
int getc_unlocked(FILE *stream);
int getchar(void);
int getchar_unlocked(void);
int fgetc(FILE *stream);
int getw(FILE *stream);
```

Obsolescent Interface

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


DESCRIPTION

getc() Returns the next character (i.e., byte) from the named input *stream*, as an unsigned character converted to 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 defined both as macros and as functions.

fgetc() Same as **getc()**, but is a function rather than a macro. **fgetc()** is slower 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., **int** in C) 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.

getc_unlocked() and **getchar_unlocked()** are identical to **getc()** and **getchar()** respectively except they do not perform any internal locking of the *stream* for multithreaded applications.

Obsolescent Interface

getw_unlocked() gets character or word from a stream file.

APPLICATION USAGE

getc_unlocked() and **getchar_unlocked()** are not thread-safe. These interfaces should be used by multithread applications which have already used **flockfile()** to acquire a mutual exclusion lock for the *stream* (see *flockfile*(3S)).

getc(), **getchar()**, **fgetc()** and **getw()** are thread-safe. The interfaces **getc_unlocked()**, **getchar_unlocked()**, **getc()**, **getchar()**, **fgetc()** and **getw()** are not async-cancel-safe. A cancellation point may occur when a thread is executing any of these interfaces.

RETURN VALUE

Upon successful completion, **getc()**, **getc_unlocked()**, **getchar()**, **getchar_unlocked()**, and **fgetc()** return the next byte from the input stream pointed to by *stream* (**stdin** for **getchar()** and **getchar_unlocked()**). If the stream is at end-of-file, the end-of-file indicator for the stream is set and EOF is returned. If a read error occurs, the error indicator for the stream is set, **errno** is set to indicate the error, and EOF is returned.

Upon successful completion, **getw()** and **getw_unlocked()** return the next word from the input stream pointed to by *stream*. If the stream is at end-of-file, the end-of-file indicator for the stream is set and **getw()** and **getw_unlocked()** return EOF. If a read error occurs, the error indicator for the stream is set, and **getw()** and **getw_unlocked()** return EOF and set **errno** to indicate the error.

ferror() and **feof()** can be used to distinguish between an error condition and an end-of-file condition.

ERRORS

getc(), **getc_unlocked()**, **getchar()**, **getchar_unlocked()**, **getw()**, **getw_unlocked()**, and **fgetc()** fail if data needs to be read into the *stream*'s buffer, and:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying *stream* and the process would be delayed in the read operation.

[EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for reading.

[EINTR] The read operation was terminated due to the receipt of a signal, and either no data was transferred or the implementation does not report partial transfer for this file.

[EIO] A physical I/O error has occurred, or the process is a member of a background process and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the **SIGTTIN** signal or the process group of the process is orphaned.

Additional **errno** values may be set by the underlying **read()** function (see *read*(2)).

WARNINGS

getc() and **getchar()** are implemented both as library functions and macros. The macro versions, which are used by default, are defined in <**stdio.h**>. To obtain the library function either use a **#undef** to remove the macro definition or, if compiling in ANSI-C mode, enclose the function name in parenthesis or use the function address. The following example illustrates each of these methods :

```
#include <stdio.h>
#undef getc
    ...
main()
{
    int (*get_char()) ();
    ...
    return_val=getc(c,fd);
    ...
    return_val=(getc)(c,fd1);
    ...
    get_char = getchar;
};
```

If the integer value returned by **getc()**, **getc_unlocked()**, **getchar()**, **getchar_unlocked()**, or **fgetc()** is stored into a character variable 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.

The macro version of **getc()** incorrectly treats a *stream* argument with side effects. In particular, **getc(*f++)** does not work sensibly. The function version of **getc()** or **fgetc()** should be used instead.

Because of possible differences in word length and byte ordering, files written using **putw()** are machine-dependent, and may be unreadable by **getw()** on a different processor.

Reentrant Interfaces

If **_REENTRANT** is defined before including <**stdio.h**>, the locked versions of the library functions for **getc()** and **getchar()** are used by default.

getw_unlocked() is an obsolescent interface supported only for compatibility with existing DCE applications. New multithreaded applications should use **getw()**.

SEE ALSO

fclose(3S), ferror(3S), flockfile(3S), fopen(3S), fread(3S), gets(3S), putc(3S), read(2), scanf(3S).

STANDARDS CONFORMANCE

getc(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

getc_unlocked(): POSIX.1C

fgetc(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, XPG4.2, FIPS 151-2, POSIX.1, ANSI C

getchar(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
getchar_unlocked(): POSIX.1C
getw(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

(ENHANCED CURSES)

NAME

getcchar — get a wide character string and rendition from a **cchar_t**

SYNOPSIS

```
#include <curses.h>

int getcchar(const cchar_t *wcval, wchar_t *wch, attr_t *attrs,
             short *color_pair, void *opts);
```

DESCRIPTION

When *wch* is not a null pointer, the **getcchar()** function extracts information from a **cchar_t** defined by *wcval*, stores the character attributes in the object pointed to by *attrs*, stores the colour pair in the object pointed to by *color_pair*, and stores the wide character string referenced by *wcval* into the array pointed to by *wch*.

When *wch* is a null pointer, **getcchar()** obtains the number of wide characters in the object pointed to by *wcval* and does not change the objects pointed to by *attrs* or *color_pair*.

The *opts* argument is reserved for definition in a future edition of this document. Currently, the application must provide a null pointer as *opts*.

RETURN VALUE

When *wch* is a null pointer, **getcchar()** returns the number of wide characters referenced by *wcval*, including the null terminator.

When *wch* is not a null pointer, **getcchar()** returns OK upon successful completion, and ERR otherwise.

ERRORS

No errors are defined.

APPLICATION USAGE

The *wcval* argument may be a value generated by a call to **setcchar()** or by a function that has a **cchar_t** output argument. If *wcval* is constructed by any other means, the effect is unspecified.

SEE ALSO

attroff(3X), can_change_color(3X), setcchar(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

getch, wgetch, mvgetch, mvwgetch — get a single-byte character from the terminal

SYNOPSIS

```
#include <curses.h>
int getch(void);
int mvgetch(int y, int x);
int mvwgetch(WINDOW *win, int y, int x);
int wgetch(WINDOW *win);
```

DESCRIPTION

These functions read a single-byte character from the terminal associated with the current or specified window. The results are unspecified if the input is not a single-byte character. If **keypad()** is enabled, these functions respond to the pressing of a function key by returning the corresponding KEY_ value defined in **<curses.h>**.

Processing of terminal input is subject to the general rules described in *Input Processing* in *curses_intro*(3X).

If echoing is enabled, then the character is echoed as though it were provided as an input argument to **insch()**, except for the following characters:

<backspace>, <left-arrow> and the current erase character:	The input is interpreted as specified in **specialchars** and then the character at the resulting cursor position is deleted as though **delch()** were called, except that if the cursor was originally in the first column of the line, then the user is alerted as though **beep()** were called.
Function keys	The user is alerted as though **beep()** were called. Information concerning the function keys is not returned to the caller.

If the current or specified window is not a pad, and it has been moved or modified since the last refresh operation, then it will be refreshed before another character is read.

RETURN VALUE

Upon successful completion, these functions return the single-byte character, KEY_ value, or ERR.

If in nodelay mode and no data is available, ERR is returned.

ERRORS

No errors are defined.

APPLICATION USAGE

Applications should not define the escape key by itself as a single-character function.

When using these functions, nocbreak mode (**nocbreak()**) and echo mode (**echo()**) should not be used at the same time. Depending on the state of the terminal when each character is typed, the program may produce undesirable results.

SEE ALSO

Input Processing in curses_intro(3X), cbreak(3X), doupdate(3X), insch(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4

The entry is rewritten for clarity. The argument list for the **getch()** function is explicitly declared as **void**.

NAME
getclock - get current value of system-wide clock

SYNOPSIS
```
#include <sys/timers.h>
int getclock(int clock_type, struct timespec *tp);
```

DESCRIPTION
The **getclock()** function gets the current value **tp** of the specified system-wide clock, **clock_type**.

getclock() supports a **clock_type** of **TIMEOFDAY**, defined in <**sys/timers.h**>, which represents the time-of-day clock for the system. For this clock, the values returned by **getclock()** represent the amount of time since the Epoch.

APPLICATION USAGE
getclock() is thread-safe and async-cancel-safe.

RETURN VALUE
Upon successful completion, **getclock()** returns a value of zero; otherwise it returns a value of –1 and sets **errno** to indicate the error.

ERRORS
getclock() fails if any of the following conditions are encountered:

[EINVAL] **clock_type** does not specify a known system-wide clock.

[EIO] An error occurred while accessing the clock device.

FILES
/usr/include/sys/timers.h

SEE ALSO
clock_gettime(2), gettimer(3C), setclock(3C).

STANDARDS CONFORMANCE
getclock(): AES

NAME

getcwd() - get pathname of current working directory

SYNOPSIS

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

DESCRIPTION

The **getcwd()** function places the absolute pathname of the current working directory in the array pointed to by *buf*, and returns *buf*. The value of *size* must be at least one greater than the length of the pathname to be returned.

If *buf* is a NULL pointer, **getcwd()** obtains *size* bytes of space using **malloc()** (see *malloc*(3C)). In this case, the pointer returned by **getcwd()** can be used as the argument in a subsequent call to **free()** (see *malloc*(3C)). Invoking **getcwd()** with *buf* as a null pointer is not recommended because this functionality may be removed from the HP-UX operating system in a future release.

APPLICATION USAGE

getcwd() is thread-safe. It is not async-cancel-safe. A cancellation point may occur when a thread is executing **getcwd()**.

RETURN VALUE

Upon successful completion, **getcwd()** returns a pointer to the current directory pathname. Otherwise, it returns NULL with **errno** set if *size* is not large enough, or if an error occurs in a lower-level function.

ERRORS

getcwd() fails if any of the following conditions are encountered:

[EINVAL]	The *size* argument is zero.
[ERANGE]	The *size* argument is greater than zero, but is smaller than the length of the pathname.
[ENAMETOOLONG]	The length of the specified pathname exceeds **PATH_MAX+1** bytes, or the length of a component of the pathname exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.

getcwd() may fail if any of the following conditions are encountered:

[EACCES]	Read or search permission is denied for a component of pathname.
[EFAULT]	*buf* points outside the allocated address space of the process. **getcwd()** may not always detect this error.
[ENOMEM]	**malloc()** failed to provide *size* bytes of memory.

EXAMPLES

```
#include <unistd.h>
#include <limits.h>

char *cwd;
char buf[PATH_MAX+1];
        ...
if ((cwd = getcwd(buf, PATH_MAX+1)) == NULL) {
    perror("pwd");
    exit(1);
}
puts(cwd);
```

AUTHOR

getcwd() was developed by AT&T.

SEE ALSO

pwd(1), malloc(3C).

STANDARDS CONFORMANCE

getcwd(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

g

NAME

getdate() - convert user format date and time

SYNOPSIS

```
#include <time.h>

struct tm *getdate(const char *string);
```

Obsolescent Interface

```
int getdate_r(const char *string, struct tm *result, int *errnum);
```

DESCRIPTION

The **getdate()** function converts user definable date and/or time specifications pointed to by *string* into a **struct tm**. The structure declaration is in the **<time.h>** header file (see *ctime*(3C)).

User-supplied templates are used to parse and interpret the input string. The templates are text files created by the user and identified via the environment variable **DATEMSK**. **DATEMSK** should be set to indicate the full path name of the template file. The first line in the template that matches the input specification is used for interpretation and conversion into the internal time format. Upon successful completion, **getdate()** returns a pointer to a **struct tm**; otherwise, it returns NULL and the symbol **getdate_err** is set to indicate the error.

The following field descriptors are supported:

%%	same as **%**
%a	abbreviated weekday name
%A	full weekday name
%b	abbreviated month name
%B	full month name
%c	locale's appropriate date and time representation
%d	day of the month (01 through 31; the leading 0 is optional)
%e	same as **%d**
%D	date as **%m/%d/%y**
%h	abbreviated month name
%H	hour (00 through 23)
%I	hour (01 through 12)
%m	month number (01 through 12)
%M	minute (00 through 59)
%n	same as **\n**
%p	locale's equivalent of either AM or PM
%r	time as **%I:%M:%S %p**
%R	time as **%H:%M**
%S	seconds (00 through 61)
%t	insert a tab
%T	time as **%H:%M:%S**
%w	weekday number (Sunday = 0 through Saturday = 6)
%x	locale's appropriate date representation
%X	locale's appropriate time representation
%y	year without century (00 through 99). For inputs 69-99, the 20th century (1900s) is assumed, and for inputs 00-68, the 21st century (2000s) is assumed.
%Y	year as *ccyy* (e.g., 1986)
%Z	time zone name or no characters if no time zone exists. If the time zone supplied by **%Z** is not the same as the time zone **getdate()** expects, an invalid specification error is returned. **getdate()** calculates the expected time zone from the **TZ** environment variable.

Month and weekday names may consist of any combination of uppercase and lowercase letters. The user can request that the input date or time specification be in a specific language by setting the **LC_TIME** category (see *setlocale*(3C)).

For descriptors that allow leading zeros, leading zeros are optional. However, the number of digits used for those descriptors must not exceed two, including leading zeros. Extra whitespace in either the template file or in *string* is ignored.

The field descriptors **%c**, **%x**, and **%X** are not supported if they include unsupported field descriptors.

The following example shows the possible contents of a template:

```
%m
%A %B %d, %Y, %H:%M:%S
%A
%B
%m/%d/%y %I %p
%d,%m,%Y %H:%M
at %A the %dst of %B in %Y
run job at %I %p, %B %dnd
%A den %d. %B %Y %H.%M Uhr
```

The following are examples of valid input specifications for the above template:

```
getdate("10/1/87 4 PM");
getdate("Friday");
getdate("Friday September 18, 1987, 10:30:30");
getdate("24,9,1986 10:30");
getdate("at monday the 1st of december in 1986");
getdate("run job at 3 PM, december 2nd");
```

If the **LC_TIME** category is set to a German locale that includes **freitag** as a weekday name and **oktober** as a month name, the following would be valid:

```
getdate("freitag den 10. oktober 1986 10.30 Uhr");
```

This example shows how local date and time specification can be defined in the template:

Invocation	Line in Template
`getdate("11/27/86")`	`%m/%d/%y`
`getdate("27.11.86")`	`%d.%m.%y`
`getdate("86-11-27")`	`%y-%m-%d`
`getdate("Friday 12:00:00")`	`%A %H:%M:%S`

The following rules apply when converting the input specification into the internal format:

- If only the weekday is given, today is assumed if the given day is equal to the current day, and next week if it is less.
- If only the month is given, the current month is assumed if the given month is equal to the current month, and next year if it is less and no year is given (the first day of the month is assumed if no day is given).
- If no hour, minute and second are given, the current hour, minute and second are assumed.
- If no date is given, today is assumed if the given hour is greater than the current hour and tomorrow is assumed if it is less.

The following examples help to illustrate the above rules assuming that the current date is **Mon Sep 22 12:19:47 EDT 1986**, and the **LC_TIME** category is set to the default **C** locale.

Input	Line in Template	Date
`Mon`	`%a`	Mon Sep 22 12:19:47 EDT 1986
`Sun`	`%a`	Sun Sep 28 12:19:47 EDT 1986
`Fri`	`%a`	Fri Sep 26 12:19:47 EDT 1986
`September`	`%B`	Mon Sep 1 12:19:47 EDT 1986
`January`	`%B`	Thu Jan 1 12:19:47 EST 1987
`December`	`%B`	Mon Dec 1 12:19:47 EST 1986
`Sep Mon`	`%b %a`	Mon Sep 1 12:19:47 EDT 1986
`Jan Fri`	`%b %a`	Fri Jan 2 12:19:47 EST 1987
`Dec Mon`	`%b %a`	Mon Dec 1 12:19:47 EST 1986
`Jan Wed 1989`	`%b %a %Y`	Wed Jan 4 12:19:47 EST 1989
`Fri 9`	`%a %H`	Fri Sep 26 09:00:00 EDT 1986
`Feb 10:30`	`%b %H:%S`	Sun Feb 1 10:30:00 EST 1987
`10:30`	`%H:%M`	Tue Sep 23 10:30:00 EDT 1986
`13:30`	`%H:%M`	Mon Sep 22 13:30:00 EDT 1986

Obsolescent Interface
getdate_r() converts user format date and time.

ERRORS
Upon failure, **getdate()** returns NULL and the symbol **getdate_err** is set to indicate the error.

The following is a complete list of the **getdate_err** settings and their interpretation:

1 the **DATEMSK** environment variable is null or undefined,

2 the template file cannot be opened for reading,

3 failed to get file status information,

4 the template file is not a regular file,

5 an error is encountered while reading the template file,

6 memory allocation failed (not enough memory available),

7 there is no line in the template that matches the input,

8 invalid input specification. For example, February 31; or the time specified cannot be represented in the **time_t** data type in 32-bit HP-UX (which represents Tuesday January 19 03:14:07 UTC, 2038) or exceeds the maximum date supported in 64-bit HP-UX (which is Friday December 31 23:59:59 UTC, 9999).

APPLICATION USAGE
getdate() is thread-safe. It is not async-cancel-safe.

WARNINGS
The return value for **getdate()** points to data whose content is overwritten by each call by the same thread.

getdate_r() is an obsolescent interface supported only for compatibility with existing DCE applications. New multi-threaded applications should use **getdate()**.

SEE ALSO
ctime(3C), ctype(3C), setlocale(3C), strftime(3C).

NAME

getdiskbyname() - get disk description by its name

SYNOPSIS

```
#include <disktab.h>

struct disktab *getdiskbyname(const char *name);
```

Obsolescent Interface

```
int getdiskbyname_r(
     const char *name,
     struct disktab *result,
     char *buffer,
     int buflen);
```

DESCRIPTION

getdiskbyname() takes a disk name (such as hp7959B) and returns a pointer to a structure that describes its geometry information and the standard disk partition tables. All information is obtained from the disktab database file (see *disktab*(4)).

The contents of the structure **disktab** include the following members. Note that there is not necessarily any correlation between the placement in this list and the order in the structure.

```
char    *d_name;            /* drive name */
char    *d_type;            /* drive type */
int     d_secsize;          /* sector size in bytes */
int     d_ntracks;          /* # tracks/cylinder */
int     d_nsectors;         /* # sectors/track */
int     d_ncylinders;       /* # cylinders */
int     d_rpm;              /* revolutions/minute */
struct  partition {
        int     p_size;     /* #sectors in partition */
        short   p_bsize;    /* block size in bytes */
        short   p_fsize;    /* frag size in bytes */
} d_partitions[NSECTIONS];
```

The constant **NSECTIONS** is defined in <**disktab.h**>.

Obsolescent Interface

getdiskbyname_r() gets disk description by its name.

APPLICATION USAGE

getdiskbyname() is thread-safe. It is not async-cancel-safe.

DIAGNOSTICS

A NULL pointer is returned in case of an error, or if *name* is not found in the disktab database file.

WARNINGS

The return value for **getdiskbyname()** points to data whose content is overwritten by each call. **getdiskbyname_r()** is an obsolescent interface supported only for compatibility with existing DCE applications. New multithreaded applications should use **getdiskbyname()**.

AUTHOR

getdiskbyname() was developed by HP and the University of California, Berkeley.

SEE ALSO

disktab(4).

NAME

getdvagent, getdvagnam, setdvagent, enddvagent, putdvagnam, copydvagent - manipulate device assignment database entry for a trusted system

SYNOPSIS

```
#include <sys/types.h>
#include <hpsecurity.h>
#include <prot.h>

struct dev_asg *getdvagent( );

struct dev_asg *getdvagnam(const char *name);

void setdvagent( );

void enddvagent( );

int putdvagnam(const char *name, struct dev_asg *dv);

struct dev_asg *copydvagent(struct dev_asg *dv);
```

DESCRIPTION

getdvagent, **getdvagnam**, and **copydvagent** each return a pointer to an object with the following structure containing the broken-out fields of an entry in the Device Assignment database. Each database entry is returned as a **dev_asg** structure, declared in the **<prot.h>** header file:

```
struct dev_field {
     char      *fd_name;        /* external name */
     char      **fd_devs;       /* device list */
     mask_t    fd_type[1];      /* tape, printer, terminal */
     char      **fd_users;      /* authorized user list */
};

/* Device Assignment Database entry */

#define AUTH_DEV_TYPE           "device type"
#define AUTH_DEV_PRINTER        0
#define AUTH_DEV_TERMINAL       1
#define AUTH_DEV_TAPE           2
#define AUTH_DEV_REMOTE         3
#define AUTH_MAX_DEV_TYPE       3
#define AUTH_DEV_TYPE_SIZE      (WORD_OF_BIT (AUTH_MAX_DEV_TYPE) + 1)

/* this structure tells which of the corresponding fields
 * in dev_field are valid (filled).
 */
struct dev_flag {
     unsigned short
          fg_name  : 1,
          fg_devs  : 1,
          fg_type  : 1,
          fg_users : 1,
     ;
};

struct dev_asg {
     struct dev_field ufld;
     struct dev_flag  uflg;
     struct dev_field sfld;
     struct dev_flag  sflg;
};
```

The Device Assignment database stores device characteristics that are related to user authorizations and synonyms. On systems supporting network connections, the Device Assignment database stores information about hosts initiating connections.

Each entry contains a *name*, which is a cross reference to the terminal control database, and a list of *devices*, each of which is a pathname corresponding to that device. This list allows the device assignment software of the trusted system to invalidate all references to a device when re-assigning it. The list is a

table of character string pointers, whose last entry is a **NULL** pointer.

fd_users is a pointer to a null-terminated table of character string pointers referring to user allowed access.

For trusted system versions supporting network connections, the device name can be a 12 character host name, where the first 8 characters are the ASCII hex address of the device, and the last 4 characters are ASCII zeroes. For example, a host with Internet address 129.75.0.3 has device name 814b00030000. The trailing four zeroes are for compatibility with ports on terminal concentrators. The SAM API's supports conversion of host name to device name. Thus, sensitivity level ranges and user authorization lists can be enforced on hosts as well as on directly connected terminals.

When **getdvagent** is first called, it returns a pointer to the first device assignment entry. Thereafter, it returns a pointer to the next entry, so successive calls can be used to search the database. **getdvagnam** searches from the beginning of the database until an entry with a device name matching *name* is found, and returns a pointer to that entry. If an end of file or an error is encountered on reading, these functions return a **NULL** pointer. **copydvagent** copies a device assignment structure and the fields to which it refers to a newly-allocated data area. Since **getdvagent**, **getdvagnam**, and **putdvagent** re-use a static structure when accessing the database, the values of any entry must be saved if these routines are used again. The *dev_asg* structure returned by **copydvagent** can be freed using *free* (see *malloc*(3C) or *malloc*(3X)).

A call to **setdvagent** has the effect of setting the device assignment database back to the first entry to allow repeated searches of the database. **enddvagent** frees all memory and closes all files used to support these routines.

putdvagnam rewrites or adds an entry to the database. If there is an entry whose *fd_name* field matches the *name* argument, that entry is replaced with the contents of the *dv* structure. Otherwise, that entry is added to the database.

APPLICATION USAGE

In a multithreaded application, these routines are safe to be called only from one dedicated thread. These routines are not POSIX.1c async-cancel safe nor async-signal safe.

RETURN VALUE

getdvagent and **getdvagnam** return a pointer to a static structure on success, or a **NULL** pointer on failure. This static structure is overwritten by **getdvagent**, **getdvagnam**, and **putdvagnam**.

putdvagnam returns 1 on success, or 0 on failure.

copydvagent returns a pointer to the newly-allocated structure on success, or a **NULL** pointer if there was a memory allocation error.

WARNINGS

The structure returned by this routine contains *pointers* to character strings and lists rather than being self-contained. **copydvagent** must be used instead of structure assignments to save a returned structure.

The value returned by **getdvagent** and **getdvagnam** refers to a structure that is overwritten by calls to these routines. To retrieve an entry, modify it, and replace it in the database, copy the entry using **copydvagent** and supply the modified buffer to **putdvagent**.

NOTES

Programs using this routine must be compiled with **-lsec**.

FILES

/tcb/files/devassign Device assignment database

SEE ALSO

authcap(4).

NAME

getenv() - return value for environment name

SYNOPSIS

```
#include <stdlib.h>
char *getenv(const 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. *name* can be either the desired name, null-terminated, or of the form *name*=*value*, in which case **getenv()** uses the portion to the left of the **=** as the search key.

APPLICATION USAGE

getenv() is thread-safe. It is not async-cancel-safe.

WARNINGS

getenv() returns a pointer to static data which can be overwritten by subsequent calls.

EXTERNAL INFLUENCES

Locale

The **LC_CTYPE** category determines the interpretation of characters in *name* as single- and/or multi-byte characters.

International Code Set Support

Single- and multi-byte character code sets are supported.

SEE ALSO

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

STANDARDS CONFORMANCE

getenv(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, POSIX.2, ANSI C

NAME

getfsent(), getfsspec(), getfsfile(), getfstype(), setfsent(), endfsent() - get file system descriptor file entry

SYNOPSIS

```
#include <checklist.h>
struct checklist *getfsent(void);
struct checklist *getfsspec(const char *spec);
struct checklist *getfsfile(const char *file);
struct checklist *getfstype(const char *type);
int setfsent(void);
int endfsent(void);
```

Remarks:

These routines are included only for compatibility with 4.2 BSD. For maximum portability and improved functionality, new applications should use the *getmntent*(3X) library routines.

DESCRIPTION

getfsent(), **getfsspec()**, **getfsfile()**, and **getfstype()** each returns a pointer to an object with the following structure containing the broken-out fields of a line in the **/etc/fstab** file. The structure is declared in the <**checklist.h**> header file:

```
struct checklist {
    char   *fs_spec;     /* special file name */
    char   *fs_bspec;    /* block special file name */
    char   *fs_dir;      /* file sys directory name */
    char   *fs_type;     /* type: ro, rw, sw, xx */
    int    fs_passno;    /* fsck pass number */
    int    fs_freq;      /* backup frequency */
};
```

The fields have meanings described in *fstab*(4). If the block special file name, the file system directory name, and the type are not all defined on the associated line in **/etc/fstab**, these routines return pointers to NULL in the **fs_bspec**, **fs_dir**, and **fs_type** fields. If the pass number or the backup frequency field are not present on the line, these routines return –1 in the corresponding structure member. **fs_freq** is reserved for future use.

getfsent() Reads the next line of the file, opening the file if necessary.

setfsent() Opens and rewinds the file.

endfsent() Closes the file.

getfsspec() Sequentially searches from beginning of file until a matching special file name is found, or until EOF is encountered.

getfsfile() Sequentially searches from the beginning of the file until a matching file system file name is found, or until EOF is encountered. **getfstype()** Sequentially searches from the beginning of the file until a matching file system type field is found, or until EOF is encountered.

DIAGNOSTICS

A null pointer is returned on EOF, invalid entry, or error.

WARNINGS

Since all information is contained in a static area, it must be copied to be saved.

AUTHOR

getfsent() was developed by HP and the University of California, Berkeley.

FILES

/etc/fstab

SEE ALSO
fstab(4).

NAME

getgrent(), getgrgid(), getgrgid_r(), getgrnam(), getgrnam_r(), setgrent(), endgrent(), fgetgrent() - get group file entry

SYNOPSIS

```
#include <grp.h>

struct group *getgrent(void);

struct group *getgrgid(gid_t gid);

int  getgrgid_r(gid_t gid, struct group *grp, char *buffer,
          size_t buflen, struct group ** result);

struct group *getgrnam(const char *name);

int  getgrnam_r(const char *name, struct group *grp, char *buffer,
          size_t buflen, struct group ** result);

void setgrent(void);

void endgrent(void);

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

Obsolescent Interfaces

```
#include <grp.h>

int  getgrent_r(struct group *result, char *buffer, int buflen,
          FILE **grfp);

void setgrent_r(FILE **grfp);

void endgrent_r(FILE **grfp);

int  fgetgrent_r(FILE *stream, struct group *result, char *buffer,
           int buflen);
```

DESCRIPTION

getgrent(), **getgrgid()**, and **getgrnam()** are used to obtain group entries, and return a pointer to an object of **group** structure. An entry may come from any of the sources for **group** specified in the **/etc/nsswitch.conf** file. See *nsswitch.conf*(4).

The **group** structure is defined in <**grp.h**> and includes the following members:

```
char  *gr_name;     /* the name of the group */
char  *gr_passwd;   /* the encrypted group password */
gid_t gr_gid;       /* the numerical group ID */
char  **gr_mem;     /* null-terminated array of pointers
                       to member names */
```

getgrent() When first called, **getgrent()** returns a pointer to the first **group** structure in the group database; thereafter, it returns a pointer to the next **group** structure in the database. In this way, successive calls can be used to search the entire database;

setgrent() Has the effect of rewinding the group database to allow repeated searches;

endgrent() Can be called to indicate that group database processing is complete;

getgrgid() Searches from the beginning of the group database until a numeric 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 group database until a group name matching *name* is found, and returns a pointer to the particular structure in which it was found;

fgetgrent() Returns a pointer to the next **group** structure in the standard I/O stream *stream*, which should be open for reading, and its contents should match the format of **/etc/group**.

Obsolescent Interfaces

getgrent_r(), **setgrent_r()**, **endgrent_r()**, **fgetgrent_r()** get group file entry.

Reentrant Interfaces

getgrgid_r() and **getgrnam_r()** both update the **struct** group pointed to by **grp** and store a pointer to that structure at the location pointed to by **result**. The structure shall contain an entry from the group database with a matching **gid** or **name**. Storage referenced by the group structure pointed to by **grp** shall be allocated from the memory provided with the **buffer** parameter, which is **buflen** in size. The maximum size needed for this buffer can be determined with the **_SC_GETGR_R_SIZE_MAX sysconf()** parameter. A NULL pointer is returned at the location pointed to by **result** on error or if the requested entry is not found.

A buffer length of 1024 is recommended.

RETURN VALUE

getgrent(), **getgrgid()**, **getgrnam()**, and **fgetgrent()** return a NULL pointer if an end-of-file or error is encountered on reading. Otherwise, the return value points to an internal static area containing a valid **group** structure.

getgrgid_r() and **getgrnam_r()** return zero upon success. Otherwise, an error number is returned to indicate the error.

ERRORS

getgrent(), **getgrgid()**, **getgrnam()** and **fgetgrent()** fail if any of the following are true:

[EIO] An I/O error has occurred.

[EMFILE] *OPEN_MAX* file descriptors are currently open in the calling process.

[ENFILE] The maximum allowable number of files is currently open in the system.

APPLICATION USAGE

getgrgid() and **getgrnam()** are not thread-safe. **setgrent()**, **getgrent()**, **endgrent()**, **getgrgid_r()** and **getgrgnam_r()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur in a thread executing **setgrent()**, **getgrent()**, **endgrent()**, **getgrgid()**, **getgrnam()**, **getgrgid_r()** or **getgrgnam_r()**.

WARNINGS

The value returned by **getgrent()**, **getgrgid()**, **getgrnam()**, and **fgetgrent()** points to an area that is overwritten by each call to any of the functions. It must be copied if it is to be saved.

Users of **getgrgid_r()** and **getgrnam_r()** should note that these interfaces now conform with POSIX.1c. **getgrent_r()**, **setgrent_r()**, **endgrent_r()** and **fgetgrent_r()** are obsolescent interfaces. These interfaces and the old prototypes of **getgrgid_r()** and **getgrnam_r()** are supported for compatibility with existing DCE applications only.

The interfaces **setgrent()**, **getgrent()**, **endgrent()**, **getgrgid()**, **getgrnam()**, **getgrgid_r()** and **getgrgnam_r()** use the Dynamic Name Service Switch. (See *nsswitch.conf*(4).) An application that uses these interfaces cannot be fully archive bound.

DEPENDENCIES

NFS

Files

/var/yp/*domainname***/group.byname**
/var/yp/*domainname***/group.bygid**

See Also

niscat(1), ypcat(1).

FILES

/etc/group

SEE ALSO

niscat(1), ypcat(1), getgroups(2), getpwent(3C), stdio(3S), group(4).

STANDARDS CONFORMANCE

getgrent(): SVID2, SVID3, XPG2

endgrent(): SVID2, SVID3, XPG2
fgetgrent(): SVID2, SVID3, XPG2
getgrgid(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1
getgrnam(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1
setgrent(): SVID2, SVID3, XPG2
getgrnam_r(), getgrgid_r(): POSIX.1c

NAME

gethostent(), gethostbyaddr(), gethostbyname(), sethostent(), endhostent() - get network host entry

SYNOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

struct hostent *gethostent(void);

struct hostent *gethostbyname(const char *name);

struct hostent *gethostbyaddr(const char *addr,
                              int len,
                              int type);

_XOPEN_SOURCE_EXTENDED only
struct hostent *gethostbyaddr(const void *addr,
                              size_t len,
                              int type);

int sethostent(int stayopen);

int endhostent(void);

_XOPEN_SOURCE_EXTENDED only
void sethostent(int stayopen);
void endhostent(void);
```

MULTITHREAD USAGE

Thread Safe:	Yes
Cancel Safe:	Yes
Async-cancel Safe:	No
Async-signal Safe:	No

These functions can be called safely in a Multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

DESCRIPTION

The **gethostent()**, **gethostbyname()**, and **gethostbyaddr()** functions each return a pointer to a structure of type **hostent**, defined as follows in **<netdb.h>**:

```
struct hostent {
    char    *h_name;
    char    **h_aliases;
    int     h_addrtype;
    int     h_length;
    char    **h_addr_list;
};
#define h_addr  h_addr_list[0]
```

The members of this structure are:

h_name	The official name of the host.
h_aliases	A null-terminated array of alternate names for the host.
h_addrtype	The type of address being returned; always **AF_INET**.
h_length	The length, in bytes, of the address.
h_addr_list	A null-terminated array of network addresses for the host.
h_addr	The first address in **h_addr_list**; this is for compatibility with previous HP-UX implementations where a **struct hostent** contains only one network address per host.

Name Service Switch-Based Operation

These host entry library routines internally call the name service switch to access the "hosts" database lookup policy configured in the **/etc/nsswitch.conf** file (see *nsswitch.conf*(4)). The lookup policy defines the order and the criteria of the supported name services used to resolve host names and Internet addresses. The operations of these name services: Domain Name Server, NIS, NIS+, and nonserver mode (e.g., files) are listed below.

Domain Name Server Operation

If the local system is configured to use the **named** name server (see *named*(1M) and *resolver*(4)) for name or address resolution, then the function:

gethostent() Always returns a NULL pointer.

sethostent() Requests the use of a connected stream socket for queries to the name server if the *stayopen* flag is non-zero. The connection is retained after each call to **gethostbyname()** or **gethostbyaddr()**.

endhostent() Closes the stream socket connection.

gethostbyname()
gethostbyaddr() Each retrieves host information from the name server. Names are matched without respect to uppercase or lowercase. For example, **berkeley.edu**, **Berkeley.EDU**, and **BERKELEY.EDU** all match the entry for **berkeley.edu**.

NIS Server Operation

If **ypserv**, the server for the Network Information Service (see *ypserv*(1M)), is used for name or address resolution, then the function:

gethostent() Returns the next entry in the NIS database.

sethostent() Initializes an internal key for the NIS database. If the *stayopen* flag is non-zero, the internal key is not cleared after calls to **endhostent()**.

endhostent() Clears the internal NIS database key.

gethostbyname()
gethostbyaddr() Each retrieves host information from the NIS database. Names are matched without respect to uppercase or lowercase. For example, **berkeley.edu**, **Berkeley.EDU**, and **BERKELEY.EDU** all match the entry for **berkeley.edu**.

NIS Plus Server Operation

If **rpc.nisd**, the server for the Network Information Service Plus (see *nis+*(1)), is used for name or address resolution, then the function:

gethostent() Returns the next entry in the NIS+ database.

sethostent() Initializes an internal key for the NIS+ database. If the *stayopen* flag is non-zero, the internal key is not cleared after calls to **endhostent()**.

endhostent() Clears the internal NIS+ database key.

gethostbyname()
gethostbyaddr() Each retrieves host information from the NIS+ database. Names are matched without respect to uppercase or lowercase. For example, **berkeley.edu**, **Berkeley.EDU**, and **BERKELEY.EDU** all match the entry for **berkeley.edu**.

Nonserver Operation

If the **/etc/hosts** file is used for name or address resolution, then the function:

gethostent() Reads the next line of **/etc/hosts**, opening the file if necessary.

sethostent() Opens and rewinds the file. If the *stayopen* flag is non-zero, the host data base is not closed after each call to **gethostent()** (either directly or indirectly through one of the other **gethost** calls).

endhostent() Closes the file.

gethostbyname() Sequentially searches from the beginning of the file until a host name (among either the official names or the aliases) matching its *name* parameter is found, or until EOF is encountered. Names are matched without respect to uppercase or lowercase, as described above in the name server case.

gethostbyaddr() Sequentially searches from the beginning of the file until an Internet address matching its *addr* parameter is found, or until EOF is encountered.

In a multithreaded application, **gethostent()**, **gethostbyaddr()**, and **gethostbyname()** use thread-specific storage that is re-used in each call. The return value, **struct hostent**, should be unique for each thread and should be saved, if desired, before the thread makes the next **gethost*()** call.

For enumeration in multithreaded applications, the position within the enumeration is a process-wide property shared by all threads. **sethostent()** may be used in a multithreaded application, but resets the enumeration position for all threads. If multiple threads interleave calls to **gethostent()**, the threads will enumerate disjoint subsets of the host database.

Arguments

Currently, only the Internet address format is understood. In calls to **gethostbyaddr()**, the parameter *addr* must be a pointer to an *in_addr* structure, an Internet address in network order (see *byteorder*(3N)) and the header file **<netinet/in.h>**). The parameter *len* must be the number of bytes in an Internet address; that is, **sizeof (struct in_addr)**. The parameter *type* must be the constant **AF_INET**.

RETURN VALUE

If successful, **gethostbyname()**, **gethostbyaddr()**, and **gethostent()** return a pointer to the requested **hostent** structure.

gethostbyname() and **gethostbyaddr()** return NULL if their *host* or *addr* parameters, respectively, cannot be found in the database. If **/etc/hosts** is being used, they also return NULL if they are unable to open **/etc/hosts**.

gethostbyaddr() also returns NULL if either its *addr* or *len* parameter is invalid.

gethostent() always returns NULL if the name server is being used.

ERRORS

If the name server is being used and **gethostbyname()** or **gethostbyaddr()** returns a NULL pointer, the external integer **h_errno** contains one of the following values:

HOST_NOT_FOUND No such host is known.

TRY_AGAIN This is usually a temporary error. The local server did not receive a response from an authoritative server. A retry at some later time may succeed.

NO_RECOVERY This is a non-recoverable error.

NO_ADDRESS The requested name is valid but does not have an IP address; this is not a temporary error. This means another type of request to the name server will result in an answer.

If the name server is not being used, the value of **h_errno** may not be meaningful.

EXAMPLES

The following code excerpt counts the number of host entries:

```
int count = 0;

(void) sethostent(0);
while (gethostent() != NULL)
     count++;
(void) endhostent();
```

The following sample program prints the canonical name, aliases, and "." separated Internet IP addresses for a given "." separated IP address.

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

```
#inlcude <netdb.h>
#include <netinet/in.h>

main(int argc, const  char **argv)
{
     u_int addr;
     struct hostent *hp;
     char **p;

     if (argc != 2) {
          (void) printf("usage: %s IP-address\n",argv[0]);
          exit (1);
     }
     if ((int) (addr = inet_addr (argv[1])) == -1) {
          (void) printf("IP-address must be of the form a.b.c.d\n");
          exit (2);
     }

     hp=gethostbyaddr((char *) &addr, sizeof (addr), AF_INET);
     if (hp == NULL) {
          (void) printf("host information for %s no found \n", argv[1]);
          exit (3);
     }

     for (p = hp->h_addr_list; *p!=0;p++){
          struct in_addr in;
          char **q;

          (void)memcpy(&in.s_addr, *p, sizeof(in.s_addr);
          (void)printf("%s\t%s",inet_ntoa(in), hp->h_name);
          for (q=hp->h_aliases;*q != 0; q++)
               (void) printf("%s", *q);
          (void)putchar('\n');
     }
     exit (0);
}
```

WARNINGS

Programs that use the interfaces described in this manpage cannot be linked statically because the implementations of these functions employ dynamic loading and linking of shared objects at run time.

h_errno is referenced as an **extern int** for single thread applications and is defined as function call macro for multithreaded applications in file **/usr/include/netdb.h**. Applications that reference **h_errno** need to include **/usr/include/netdb.h**.

OBSOLESCENT INTERFACES

```
int gethostent_r(struct hostent *result,
                 struct hostent_data *buffer);

int gethostbyname_r(const char *name,
                    struct hostent *result,
                    struct hostent_data *buffer);

int gethostbyaddr_r(const char *addr,
                    int len,
                    int type,
                    struct hostent *result,
                    struct hostent_data *buffer);

int sethostent_r(int stayopen, struct hostent_data *buffer);

int endhostent_r(struct hostent_data *buffer);
```

The above reentrant interfaces have been moved from **libc** to **libd4r**. They are included to support existing applications and may be removed in the future release. New multithreaded applications should use the regular APIs (those without the **-r** suffix).

The reentrant interfaces function the same as the regular interfaces without the **-r** suffix. However, **gethostent_r()**, **gethostbyname_r()**, and **gethostbyaddr_r()** expect to be passed the address of a **struct hostent** and will store the address of the result at the supplied parameter. The passed in address of **struct hostent_data** in the reentrant interfaces cannot be a NULL pointer.

The reentrant routines return **-1** if the operation is unsuccessful, or, in the case of **gethostent_r()**, if the end of the hosts list has been reached. **0** is returned otherwise.

AUTHOR

gethostent() was developed by Sun Microsystems Inc.

FILES

/etc/hosts

SEE ALSO

named(1M), ypserv(1M), resolver(3N), ypclnt(3C), hosts(4), nsswitch.conf(4), ypfiles(4), nis+(1).

STANDARDS CONFORMANCE

gethostent(): XPG4

NAME

getlogin(), getlogin_r() - get name of user logged in on this terminal

SYNOPSIS

```
#include <unistd.h>
char *getlogin(void);
int getlogin_r(char *buf, size_t buflen);
```

DESCRIPTION

The **getlogin()** function retrieves the name of the user currently logged in on a terminal associated with the calling process, as found in **/etc/utmp**.

At least one of the standard input, standard output, or standard error must be a terminal. For the first of these found that is a terminal, a user must have logged in on that terminal, and that terminal must be the controlling terminal of the session leader process of the calling process's session.

The **getlogin()** function can be used in conjunction with **getpwnam()** to locate the correct password file entry when the same user ID is shared by several login names.

The recommended procedure to obtain the user name associated with the real user ID of the calling process is to call **getlogin()**, and if that fails, to call **getpwuid(getuid())**.

To get the user name associated with the effective user ID, call **getpwuid(geteuid())**.

getlogin_r() performs the same operations as **getlogin()**, but returns the login name in the buffer to which *buf* points, whose size in bytes should be passed in *buflen*. *buf* should have space for the name and the terminating null character. The maximum size of the login name is **LOGIN_NAME_MAX**.

APPLICATION USAGE

The return value from **getlogin()** points to static data whose content is overwritten by each call. **getlogin_r()** is thread-safe. **getlogin_r()** is not async-cancel-safe. A cancellation point may occur when a thread is executing **getlogin()** or **getlogin_r()**.

RETURN VALUE

Upon successfully finding and validating the login name of the user logged in on the terminal, **getlogin()** returns a pointer to the name. Otherwise, it returns a null pointer, and sets **errno** to indicate the error.

Upon successfully finding, validating, and copying to the buffer the login name of the user logged in on the terminal, **getlogin_r()** returns 0 upon success and returns an error number upon failure.

ERRORS

getlogin() and **getlogin_r()** fail if any of the following is true:

[EACCES] Access permission to read the **/etc/utmp** file, or to get the status of the terminal device file, was denied.

[EMFILE] Too many file descriptors are in use by this process.

[ENFILE] Too many file descriptors are in use on the system.

[ENOENT] The **/etc/utmp** file or the terminal device file cannot be found.

[ENOTTY] None of the standard input, standard output, or standard error is a terminal, or for the first of these that is a terminal, no current login is registered on that terminal, or the session leader process of the calling process has no controlling terminal.

[EPERM] One of the standard input, standard output, or standard error is a terminal, and a current login was found on that terminal, but that terminal is not the same as the controlling terminal of the session of the calling process.

[ESRCH] The session leader process of the calling process is no longer running.

The error condition associated with [EPERM] prevents processes that have access to some other user's terminal from believing that they are related to that other user's login session.

getlogin_r() also fails if the following is true:

[ERANGE] The length of the name to be returned, including the terminating null byte, exceeds *buflen*.

FILES

/etc/utmp Database that maps user logins to terminals.

WARNINGS

Users of **getlogin_r()** should note **getlogin_r()** now conforms with the POSIX.1c Threads standard. The old prototype of **getlogin_r()** is supported for compatibility with existing DCE applications only.

SEE ALSO

getuid(2), getgrent(3C), getpwent(3C), utmp(4).

STANDARDS CONFORMANCE

getlogin(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

getlogin_r(): POSIX.1c

NAME

getmntent(), getmntent_r(), setmntent(), addmntent(), endmntent(), hasmntopt() - get file system descriptor file entry

SYNOPSIS

```
#include <mntent.h>

FILE *setmntent(const char *path, char *type);

struct mntent *getmntent(FILE *stream);

int getmntent_r(
     FILE *stream,
     struct mntent *result,
     char *buffer,
     int buflen);

int addmntent(FILE *stream, struct mntent *mnt);

char *hasmntopt(struct mntent *mnt, const char *opt);

int endmntent(FILE *stream);
```

DESCRIPTION

These routines replace the obsolete **getfsent()** routines (see *getfsent*(3X)) for accessing the file system description file **/etc/fstab**. They are also used to access the mounted file system description file **/etc/mnttab**.

setmntent() Opens a file system description file and returns a file pointer which can then be used with **getmntent()**, **addmntent()**, or **endmntent()**. The *type* argument is the same as in *fopen*(3C).

getmntent() Reads the next line from *stream* and returns a pointer to an object with the following structure containing the broken-out fields of a line in the file-system description file, <**mntent.h**>. The fields have meanings described in *fstab*(4).

```
struct mntent {
   char    *mnt_fsname; /* file system name */
   char    *mnt_dir;    /* file system path prefix */
   char    *mnt_type;   /* hfs, nfs, swap, or xx */
   char    *mnt_opts;   /* ro, suid, etc. */
   int     mnt_freq;    /* dump frequency, in days */
   int     mnt_passno;  /* pass number on parallel fsck */
   long    mnt_time;    /* When file system was mounted; */
                        /* see mnttab(4). */
                        /* (0 for NFS) */
};
```

getmntent_r() Uses three extra parameters to provide results equivalent to those produced by **getmntent()**. The extra parameters are:
1. The address of a **struct mntent** where the result will be stored.
2. A buffer to store character strings to which fields in the **struct mntent** will point.
3. The length of the user-supplied buffer. A buffer length of 1025 is recommended.

addmntent() Adds the **mntent** structure *mnt* to the end of the open file *stream*. Note that *stream* must be opened for writing.

hasmntopt() Scans the **mnt_opts** field of the **mntent** structure *mnt* for a substring that matches *opt*. It returns the address of the substring if a match is found; **0** otherwise.

endmntent() Closes the file.

The following definitions are provided in <**mntent.h**>:

```
#define MNT_CHECKLIST  "/etc/fstab"
#define MNT_MNTTAB     "/etc/mnttab"

#define MNTMAXSTR       128        /* Max size string in mntent */
```

```
#define MNTTYPE_HFS      "hfs"      /* HFS file system */
#define MNTTYPE_CDFS     "cdfs"     /* CD-ROM file system */
#define MNTTYPE_NFS      "nfs"      /* Network file system */
#define MNTTYPE_SWAP     "swap"     /* Swap device */
#define MNTTYPE_SWAPFS   "swapfs"   /* File system swap */
#define MNTTYPE_IGNORE   "ignore"   /* Ignore this entry */

#define MNTOPT_DEFAULTS "defaults" /* Use all default options */
#define MNTOPT_RO        "ro"       /* Read only */
#define MNTOPT_RW        "rw"       /* Read/write */
#define MNTOPT_SUID      "suid"     /* Set uid allowed */
#define MNTOPT_NOSUID    "nosuid"   /* No set uid allowed */
#define MNTOPT_QUOTA     "quota"    /* Enable disk quotas */
#define MNTOPT_NOQUOTA   "noquota"  /* Disable disk quotas */
```

The following definition is provided for device swap in <**mntent.h**>:

```
#define MNTOPT_END    "end"     /* swap after end of file system,
                                   Series 300/400/700 only */
```

The following definitions are provided for file system swap in <**mntent.h**>:

```
#define MNTOPT_MIN    "min"     /* minimum file system swap */
#define MNTOPT_LIM    "lim"     /* maximum file system swap */
#define MNTOPT_RES    "res"     /* reserve space for file system */
#define MNTOPT_PRI    "pri"     /* file system swap priority */
```

NETWORKING FEATURES

NFS

The following definitions are provided in <**mntent.h**>:

```
#define MNTOPT_BG        "bg"       /* Retry mount in background */
#define MNTOPT_FG        "fg"       /* Retry mount in foreground */
#define MNTOPT_RETRY     "retry"    /* Number of retries allowed */
#define MNTOPT_RSIZE     "rsize"    /* Read buffer size in bytes */
#define MNTOPT_WSIZE     "wsize"    /* Write buffer size in bytes*/
#define MNTOPT_TIMEO     "timeo"    /* Timeout in 1/10 seconds */
#define MNTOPT_RETRANS   "retrans"  /* Number of retransmissions */
#define MNTOPT_PORT      "port"     /* Server's IP NFS port */
#define MNTOPT_SOFT      "soft"     /* Soft mount */
#define MNTOPT_HARD      "hard"     /* Hard mount */
#define MNTOPT_INTR      "intr"     /* Interruptable hard mounts */
#define MNTOPT_NOINTR    "nointr"   /* Uninterruptable hard mounts*/
#define MNTOPT_DEVS      "devs"     /* Device file access allowed */
#define MNTOPT_NODEVS    "nodevs"   /* No device file access allowed */
```

RETURN VALUE

setmntent() Returns a null pointer on error. **setmntent()** attempts to establish either a shared read lock or an exclusive write lock on the file it is opening. If it cannot get the lock, it returns a null pointer and sets errno to either EACESS or EAGAIN. The application program can retry when encountering such error.

getmntent() Returns a null pointer on error or EOF. Otherwise, **getmntent()** returns a pointer to a mntent structure. Some of the fields comprising a mntent structure are optional in **/etc/fstab** and **/etc/mnttab**. In the supplied structure, such missing character pointer fields are set to NULL and missing integer fields are set to –1 for **mnt_freq** and **mnt_passno.** If the integer field for **mnt_time** is missing, it is set to 0.

getmntent_r() Returns a **-1** on error or EOF, or if the supplied buffer is of insufficient length. If the operation is successful, **0** is returned.

addmntent() Returns **1** on error.

endmntent() Returns **1**.

WARNINGS

The return value for **getmntent()** points to static information that is overwritten in each call. Thus, **getmntent()** is unsafe for multi-thread applications. **getmntent_r()** is MT-Safe and should be used instead.

AUTHOR

addmntent(), **endmntent()**, **getmntent()**, **hasmntopt()**, and **setmntent()** were developed by The University of California, Berkeley, Sun Microsystems, Inc., and HP.

FILES

/etc/fstab
/etc/mnttab

SEE ALSO

fstab(4), getfsent(3X), mnttab(4).

g

NAME

getn_wstr, get_wstr, mvgetn_wstr, mvget_wstr, mvwgetn_wstr, mvwget_wstr, wgetn_wstr, wget_wstr — get an array of wide characters and function key codes from a terminal

SYNOPSIS

```
#include <curses.h>
int getn_wstr(wchar_t *wstr, int n);
int get_wstr(wchar_t *wstr);
int mvgetn_wstr(int y, int x, wchar_t *wstr, int n);
int mvget_wstr(int y, int x, wchar_t *wstr);
int mvwgetn_wstr(WINDOW *win, int y, int x, wchar_t *wstr, int n);
int mvwget_wstr(WINDOW *win, int y, int x, wchar_t *wstr);
int wgetn_wstr(WINDOW *win, wchar_t *wstr, int n);
int wget_wstr(WINDOW *win, wchar_t *wstr);
```

DESCRIPTION

The effect of **get_wstr()** is as though a series of calls to **get_wch()** were made, until a newline character, end-of-line character, or end-of-file character is processed. An end-of-file character is represented by WEOF, as defined in **<wchar.h>**. A newline or end-of-line is represented as its **wchar_t** value. In all instances, the end of the string is terminated by a null **wchar_t**. The resulting values are placed in the area pointed to by *wstr*.

The user's erase and kill characters are interpreted and affect the sequence of characters returned.

The effect of **wget_wstr()** is as though a series of calls to **wget_wch()** were made.

The effect of **mvget_wstr()** is as though a call to **move()** and then a series of calls to **get_wch()** were made. The effect of **mvwget_wstr()** is as though a call to **wmove()** and then a series of calls to **wget_wch()** were made. The effect of **mvget_nwstr()** is as though a call to **move()** and then a series of calls to **get_wch()** were made. The effect of **mvwget_nwstr()** is as though a call to **wmove()** and then a series of calls to **wget_wch()** were made.

The **getn_wstr()**, **mvgetn_wstr()**, **mvwgetn_wstr()** and **wgetn_wstr()** functions read at most *n* characters, letting the application prevent overflow of the input buffer.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

Reading a line that overflows the array pointed to by *wstr* with **get_wstr()**, **mvget_wstr()**, **mvwget_wstr()** or **wget_wstr()** causes undefined results. The use of **getn_wstr()**, **mvgetn_wstr()**, **mvwgetn_wstr()** or **wgetn_wstr()**, respectively, is recommended.

These functions cannot return KEY_ values as there is no way to distinguish a KEY_ value from a valid **wchar_t** value.

SEE ALSO

get_wch(3X), getstr(3X), <curses.h>, <wchar.h> (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification), *X/Open System Interface Definitions, Issue 4, Version 2* specification, Chapter 9, *General Terminal Interface*.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

getnetconfig(), setnetconfig(), endnetconfig(), getnetconfigent(), freenetconfigent(), nc_perror(), nc_sperror() - get network configuration database entry

SYNOPSIS

```
#include <netconfig.h>
struct netconfig *getnetconfig(void *handlep );
void *setnetconfig(void);
int endnetconfig(void *handlep );
struct netconfig *getnetconfigent(const char *netid );
void freenetconfigent(struct netconfig *netconfigp );
void nc_perror(const char *msg );
char *nc_sperror(void);
```

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

DESCRIPTION

The library routines described on this page are part of the Network Selection component. They provide the application access to the system network configuration database, **/etc/netconfig**. In addition to the routines for accessing the **netconfig** database, Network Selection includes the environment variable *NETPATH* (see *environ*(5)) and the *NETPATH* access routines described in *getnetpath*(3N).

getnetconfig() returns a pointer to the current entry in the **netconfig** database, formatted as a **struct netconfig**. Successive calls will return successive **netconfig** entries in the **netconfig** database. **getnetconfig()** can be used to search the entire **netconfig** file. **getnetconfig()** returns NULL at the end of the file. *handlep* is the handle obtained through **setnetconfig()**.

A call to **setnetconfig()** has the effect of "binding" to or "rewinding" the **netconfig** database. **setnetconfig()** must be called before the first call to **getnetconfig()** and may be called at any other time. **setnetconfig()** need *not* be called before a call to **getnetconfigent()**.

setnetconfig() returns a unique handle to be used by **getnetconfig()**.

endnetconfig() should be called when processing is complete to release resources for reuse. *handlep* is the handle obtained through **setnetconfig()**. Programmers should be aware, however, that the last call to **endnetconfig()** frees all memory allocated by **getnetconfig()** for the **struct netconfig** data structure. **endnetconfig()** may not be called before **setnetconfig()**.

getnetconfigent() returns a pointer to the struct netconfig structure corresponding to *netid*. It returns NULL if *netid* is invalid (that is, does not name an entry in the *netconfig* database).

freenetconfigent() frees the **netconfig** structure pointed to by *netconfigp* (previously returned by **getnetconfigent()**).

nc_perror() prints a message to the standard error indicating why any of the above routines failed. The message is prepended with the string *msg* and a colon. A NEWLINE is appended at the end of the message.

nc_sperror() is similar to **nc_perror()** but instead of sending the message to the standard error, will return a pointer to a string that contains the error message.

nc_perror() and **nc_sperror()** can also be used with the *NETPATH* access routines defined in *getnetpath*(3N).

RETURN VALUES

setnetconfig() returns a unique handle to be used by **getnetconfig()**. In the case of an error, **setnetconfig()** returns NULL and **nc_perror()** or **nc_sperror()** can be used to print the reason for failure.

getnetconfig() returns a pointer to the current entry in the **netconfig()** database, formatted as a **struct netconfig**. **getnetconfig()** returns NULL at the end of the file, or upon failure.

endnetconfig() returns 40 on success and -1 on failure, for example, if **setnetconfig()** was not called previously.

On success, **getnetconfigent()** returns a pointer to the **struct netconfig** structure corresponding to *netid*; otherwise it returns NULL.

nc_sperror() returns a pointer to a buffer which contains the error message string. This buffer is overwritten on each call. In multithreaded applications, this buffer is implemented as thread-specific data.

SEE ALSO

getnetpath(3N), netconfig(4), environ(5).

NAME

getnetent(), getnetbyaddr(), getnetbyname(), setnetent(), endnetent()- get network entry

SYNOPSIS

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

struct netent *getnetent(void);

struct netent *getnetbyname(const char *name);

struct netent *getnetbyaddr(int net, int type);

_XOPEN_SOURCE_EXTENDED only
struct netent *getnetbyaddr(in_addr_t net, int type);

int setnetent(int stayopen);

int endnetent(void);

_XOPEN_SOURCE_EXTENDED only
void setnetent(int stayopen);
void endnetent(void);
```

MULTITHREAD USAGE

Thread Safe:	Yes
Cancel Safe:	Yes
Async-cancel Safe:	No
Async-signal Safe:	No

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

DESCRIPTION

getnetent(), **getnetbyname()**, and **getnetbyaddr()** each return a pointer to a structure of type *netent* containing the broken-out fields of a line in the network data base, **/etc/networks**.

The members of this structure are:

n_name The official name of the network.

n_aliases A null-terminated list of alternate names for the network.

n_addrtype The type of the network number returned; always **AF_INET**.

n_net The network number.

Functions behave as follows:

getnetent() Reads the next line of the file, opening the file if necessary.

setnetent() Opens and rewinds the file. If the *stayopen* flag is non-zero, the network data base is not closed after each call to **getnetent()** (either directly or indirectly through one of the other **getnet*** calls).

endnetent() Closes the file.

getnetbyname() Sequentially searches from the beginning of the file until a network name (among either the official names or the aliases) matching its parameter *name* is found, or until EOF is encountered.

getnetbyaddr() Sequentially searches from the beginning of the file until a network number matching its parameter *net* is found, or until EOF is encountered. The parameter *net* must be in network order. The parameter *type* must be the constant **AF_INET**. Network numbers are supplied in host order (see *byteorder*(3N)).

If the system is running Network Information Service (NFS), **getnetbyname()** and **getnetbyaddr()** obtain their network information from the NIS server (see *ypserv*(1M) and *ypfiles*(4)) or NIS+ server (see *nis+*(1)) ,respectively.

In a multithreaded application, **getnetent()**, **getentbyaddr()**, and **getentbyname()** use thread-specific storage that is re-used in each call. The return value, **struct netent**, should be unique for each thread and should be saved, if desired, before the thread makes the next **getnet*()** call.

For enumeration in multithreaded applications, the position within the enumeration is a process-wide property shared by all threads. **setnetent()** may be used in a multithreaded application, but resets the enumeration position for all threads. If multiple threads interleave calls to **getnetent()**, the threads will enumerate disjoint subsets of the network database.

Name Service Switch-Based Operation

The library routines **getnetbyname()**, **getnetbyaddr()**, and **getnetent()** internally call the name service switch to access the "networks" database lookup policy configured in the **/etc/nsswitch.conf** file (see *nsswitch.conf*(4)). The lookup policy defines the order and the criteria of the supported name services used to resolve network names and addresses.

RETURN VALUE

getnetent(), **getnetbyname()**, and **getnetbyaddr()** return a null pointer (0) on EOF or when they are unable to open **/etc/networks**. **getnetbyaddr()** also returns a null pointer if its *type* parameter is invalid.

EXAMPLE

The following code excerpt counts the number of network entries:

```
int count = 0;

(void) setnetent(0);
while (netbuf=getnetent() != NULL)
    count++;
(void) endnetent();
```

OBSOLESCENT INTERFACES

```
int getnetent_r(struct netent *result, struct netent_data *buffer);

int getnetbyname_r(
     const char *name,
     struct netent *result,
     struct netent_data *buffer);

int getnetbyaddr_r(
     int net,
     int type,
     struct netent *result,
     struct netent_data *buffer);

int setnetent_r(int stayopen, struct netent_data *buffer);

int endnetent_r(struct netent_data *buffer);
```

The above reentrant interfaces have been moved from **libc** to **libd4r**. They are included to support existing applications and may be removed in the future release. New multithreaded applications should not use these APIs.

The reentrant interfaces function the same as the regular interfaces (those without the **-r** suffix.) However, **getnetent_r()**, **getnetbyname_r()**, and **getnetbyaddr_r()** expect to be passed the address of a *struct netent* and will store the address of the result at the supplied parameter. An additional parameter, the address of *struct netent_data*, which is defined in the file **<netdb.h>**, cannot be a NULL pointer.

getnetent_r(), **getnetbyname_r()**, **getnetbyaddr_r()**, **setnetent_r()**, and **endnetent_t()** return a –1 if the operation is unsuccessful. A 0 is returned otherwise.

WARNINGS

Programs that use the interfaces described in this manpage cannot be linked statically because the implementations of these functions employ dynamic loading and linking of shared objects at run time.

AUTHOR

getnetent() was developed by Sun Microsystems Inc.

FILES
`/etc/networks`

SEE ALSO
ypserv(1M), networks(4), ypfiles(4), nsswitch.conf(4), nis+(1).

STANDARDS CONFORMANCE
`getnetent()`: XPG4

NAME

getnetgrent(), setnetgrent(), endnetgrent(), innetgr(), - get network group entry

SYNOPSIS

```
int innetgr(
     char *netgroup,
     char *machine,
     char *user,
     char *domain
);
int setnetgrent(char *netgroup);
int endnetgrent();
int getnetgrent(
     char **machinep,
     char **userp,
     char **domainp
);
```

DESCRIPTION

These functions are used to test membership in and enumerate members of "netgroup" network groups defined in a system database. Netgroups are sets of (machine,user,domain) triples (see *netgroup*(4)).

These functions consult the source specified for **netgroup** in the **/etc/nsswitch.conf** file (see *nsswitch.conf*(4)).

The function **innetgr()** returns **1** if there is a netgroup *netgroup* that contains the specified *machine*, *user*, *domain* triple as a member; otherwise it returns **0**. Any of the supplied pointers *machine*, *user*, and *domain* may be **NULL**, signifying a "wild card" that matches all values in that position of the triple.

The **innetgr()** function is safe for use in multithreaded applications.

The functions **setnetgrent()**, **getnetgrent()**, and **endnetgrent()** are used to enumerate the members of a given network group.

The function **setnetgrent()** establishes the network group specified in the parameter *netgroup* as the current group whose members are to be enumerated.

Successive calls to the function **getnetgrent()** will enumerate the members of the group established by calling **setnetgrent()**; each call returns 1 if it succeeds in obtaining another member of the network group, or 0 if there are no further members of the group.

When calling **getnetgrent()** addresses of the three character pointers are used as arguments; i.e.:

```
char *mp, *up, *dp;
getnetgrent(&mp, &up, &dp);
```

Upon successful return from **getnetgrent()**, the pointer *mp* points to a thread specific storage area containing the name of the machine part of the member triple, *up* points to a thread specific storage area containing the user name and *dp* points to a thread specific storage area containing the domain name. If the pointer returned for *mp*, *up*, or *dp* is **NULL**, it signifies that the element of the netgroup contains wild card specifier in that position of the triple.

The storage allocated by **setnetgrent()** is released when an **endnetgrent()** call is made, and should not be released by the caller.

The function **endnetgrent()** frees the space allocated by the previous **setnetgrent()** call. The equivalent of an **endnetgrent()** implicitly performed whenever a **setnetgrent()** call is made to a new network group.

Note that while **setnetgrent()** and **endnetgrent()** are safe for use in multi-threaded applications, the effect of each is process-wide. Calling **setnetgrent()** resets the enumeration position for all threads. If multiple threads interleave calls to **getnetgrent_r()** each will enumerate a disjoint subset of the netgroup. Thus the effective use of these functions in multi-threaded applications may require coordination by the caller.

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

WARNINGS

Programs that use the interfaces described in this manual page cannot be linked statically since the implementations of these functions employ dynamic loading and linking of shared objects at run time.

FILES

/etc/netgroup
/etc/nsswitch.conf

SEE ALSO

netgroup(4), nsswitch.conf(4).

NAME

getnetpath(), setnetpath(), endnetpath() - get /etc/netconfig entry corresponding to NETPATH component

SYNOPSIS

```
#include <netconfig.h>
struct netconfig *getnetpath(void *handlep );
void *setnetpath(void);
int endnetpath(void *handlep );
```

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that supports asynchronous cancellation or asynchronous signals.

DESCRIPTION

The routines described on this page are part of the Network Selection component. They provide the application access to the system network configuration database, **/etc/netconfig**, as it is "filtered" by the *NETPATH* environment variable (see *environ*(5)). See *getnetconfig*(3N) for other routines that also access the network configuration database directly. The *NETPATH* variable is a list of colon-separated network identifiers.

getnetpath() returns a pointer to the **netconfig** database entry corresponding to the first valid *NETPATH* component. The **netconfig** entry is formatted as a **struct** netconfig**.** On each subsequent call, **getnetpath()** returns a pointer to the **netconfig** entry that corresponds to the next valid **NETPATH** component. **getnetpath()** can thus be used to search the **netconfig** database for all networks included in the *NETPATH* variable. When *NETPATH* has been exhausted, **getnetpath()** returns NULL.

A call to **setnetpath()** "binds" to or "rewinds" *NETPATH* must be called before the first call to **getnetpath()** and may be called at any other time. It returns a handle that is used by **getnetpath()**.

getnetpath() silently ignores invalid *NETPATH* components. A *NETPATH* component is invalid if there is no corresponding entry in the **netconfig** database.

If the *NETPATH* variable is *unset*, **getnetpath()** behaves as if *NETPATH* were set to the sequence of "default" or "visible" networks in the **netconfig** database, in the order in which they are listed.

endnetpath() may be called to "unbind" from *NETPATH* when processing is complete, releasing resources for reuse. Programmers should be aware, however, that **endnetpath()** frees all memory allocated by **getnetpath()** for the **struct netconfig** data structure. **endnetpath()** returns 40 on success and -1 on failure, for example, if **setnetpath()** was not called previously.

RETURN VALUES

setnetpath() returns a handle that is used by **getnetpath()**. In case of an error, **setnetpath()** returns NULL. **nc_perror()** or **nc_sperror()** can be used to print out the reason for failure. See *getnetconfig*(3N).

When first called, **getnetpath()** returns a pointer to the **netconfig** database entry corresponding to the first valid *NETPATH* component. When *NETPATH* has been exhausted, **getnetpath()** returns NULL.

endnetpath() returns 40 on success and -1 on failure, for example, if **setnetpath()** was not called previously.

SEE ALSO
getnetconfig(3N), netconfig(4), environ(5).

(ENHANCED CURSES)

NAME

getnstr, mvgetnstr, mvwgetnstr, wgetnstr, — get a multi-byte character length limited string from the terminal

SYNOPSIS

```
#include <curses.h>
int getnstr(char *str, int n);
int mvgetnstr(int y, int x, char *str, int n);
int mvwgetnstr(WINDOW *win, int y, int x, char *str, int n);
int wgetnstr(WINDOW *win, char *str, int n);
```

DESCRIPTION

The effect of **getnstr()** and **wgetnstr()** is as though a series of calls to **getch()** were made, until a newline or carriage return is received. The resulting value is placed in the area pointed to by *str*. The **getnstr()** and **wgetnstr()** functions read at most *n* bytes, thus preventing a possible overflow of the input buffer. The user's erase and kill characters are interpreted, as well as any special keys (such as function keys, home key, clear key, and so on).

The **mvgetnstr()** function is identical to **getnstr()** except that it is as though it is a call to **move()** and then a series of calls to **getch()**. The **mvwgetnstr()** function is identical to **getnstr()** except it is as though a call to **wmove()** is made and then a series of calls to **wgetch()**.

The **getnstr()**, **wgetnstr()**, **mvgetnstr()** and **mvwgetnstr()** functions will only return the entire multi-byte sequence associated with a character. If the array is large enough to contain at least one character, the functions fill the array with complete characters. If the array is not large enough to contain any complete characters, the function fails.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

Traditional implementations often limited the number of bytes returned to 256.

SEE ALSO

Input Processing in curses_intro(3X), beep(3X), getch(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

getopt(), optarg, optind, opterr - get option letter from argument vector

SYNOPSIS

```
#include <unistd.h>
int getopt(int argc, char * const argv[], const char *optstring);
extern char *optarg;
extern int optind, opterr, optopt;
```

DESCRIPTION

getopt() returns the next option letter in *argv* (starting from *argv*[1]) that matches a letter in *optstring*. *argc* and *argv* are the argument count and argument array as passed to **main()**. *optstring* is a string of recognized option characters; if a character is followed by a colon, the option takes an argument which may or may not be separated from it by white space.

optind is the index of the next element of the *argv*[] vector to be processed. It is initialized to 1 by the system, and **getopt()** updates it when it finishes with each element of *argv*[].

getopt() returns the next option character from *argv* that matches a character in *optstring*, if there is one that matches. If the option takes an argument, **getopt()** sets the variable **optarg** to point to the option-argument as follows:

- If the option was the last character in the string pointed to by an element of *argv*, then **optarg** contains the next element of *argv*, and **optind** is incremented by 2. If the resulting value of **optind** is greater than or equal to *argc*, this indicates a missing option argument, and **getopt()** returns an error indication.
- Otherwise, **optarg** points to the string following the option character in that element of *argv*, and **optind** is incremented by 1.

If, when **getopt()** is called, *argv*[*optind*] is NULL, or the string pointed to by *argv*[*optind*] either does not begin with the character **-** or consists only of the character **-**, **getopt()** returns –1 without changing *optind*. If *argv*[*optind*] points to the string **--**, **getopt()** returns –1 after incrementing *optind*.

If **getopt()** encounters an option character that is not contained in *optstring*, it returns the question-mark (**?**) character. If it detects a missing option argument, it returns the colon character (**:**) if the first character of *optstring* was a colon, or a question-mark character otherwise. In either case, **getopt()** sets the variable *optopt* to the option character that caused the error. If the application has not set the variable **opterr** to zero and the first character of *optstring* is not a colon, **getopt()** also prints a diagnostic message to standard error.

The special option **--** can be used to delimit the end of the options; –1 is returned, and **--** is skipped.

RETURN VALUE

getopt() returns the next option character specified on the command line. A colon (**:**) is returned if **getopt()** detects a missing argument and the first character of *optstring* was a colon (**:**).

A question-mark (**?**) is returned if **getopt()** encounters an option character not in *optstring* or detects a missing argument and the first character of *optstring* was not a colon (**:**).

Otherwise, **getopt()** returns –1 when all command line options have been parsed.

EXTERNAL INFLUENCES

Locale

The **LC_CTYPE** category determines the interpretation of option letters as single and/or multi-byte characters.

International Code Set Support

Single- and multibyte character code sets are supported.

ERRORS

getopt() fails under the following conditions:

[EILSEQ] An invalid multibyte character sequence was encountered during option processing.

EXAMPLES

The following code fragment shows to process 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:

```
#include <stdio.h>
#include <unistd.h>
main (int argc, char *argv[])
{
    int c;
    int bflg, aflg, errflg;
    extern char *optarg;
    extern int optind, optopt;
    .
    .
    .
    while ((c = getopt(argc, argv, ":abf:o:")) != -1)
        switch (c) {
        case 'a':
            if (bflg)
                errflg++;
            else
                aflg++;
            break;
        case 'b':
            if (aflg)
                errflg++;
            else {
                bflg++;
                bproc( );
            }
            break;
        case 'f':
            ifile = optarg;
            break;
        case 'o':
            ofile = optarg;
            break;
        case ':':       /* -f or -o without arguments */
            fprintf(stderr, "Option -%c requires an argument\n",
                optopt);
            errflg++;
            break;
        case '?':
            fprintf(stderr, "Unrecognized option: - %c\n",
                optopt);
            errflg++;
        }
    if (errflg) {
        fprintf(stderr, "usage: . . . ");
        exit (2);
    }
    for ( ; optind < argc; optind++) {
        if (access(argv[optind], 4)) {
    .
    .
    .
}
```

WARNINGS

Options can be any ASCII characters except colon (**:**), question mark (**?**), or null (**\0**). It is impossible to distinguish between a **?** used as a legal option, and the character that **getopt()** returns when it encounters an invalid option character in the input.

getopt() is unsafe in multi-thread applications.

SEE ALSO

getopt(1).

STANDARDS CONFORMANCE

getopt(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, POSIX.2

optarg: AES, SVID2, SVID3, XPG2, XPG3, XPG4, POSIX.2

opterr: AES, SVID2, SVID3, XPG2, XPG3, XPG4, POSIX.2

optind: AES, SVID2, SVID3, XPG2, XPG3, XPG4, POSIX.2

optopt: AES, SVID3, XPG4, POSIX.2

NAME

getpass() - read a password

SYNOPSIS

```
#include <unistd.h>
char *getpass(const 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 terminates input and sends an interrupt signal to the calling program before returning.

APPLICATION USAGE

getpass() is not thread-safe. It is not async-cancel-safe. A cancellation point may occur when a thread is executing **getpass().**

WARNINGS

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

FILES

/dev/tty

SEE ALSO

crypt(3C).

STANDARDS CONFORMANCE

getpass(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

NAME

getprdfent, getprdfnam, setprdfent, endprdfent, putprdfnam - manipulate system default database entry for a trusted system

SYNOPSIS

```
#include <sys/types.h>
#include <hpsecurity.h>
#include <prot.h>

struct pr_default *getprdfent(void);

struct pr_default *getprdfnam(const char *name);

void setprdfent(void);

void endprdfent(void);

int putprdfnam(const char *name, struct pr_default *pr);
```

DESCRIPTION

getprdfent and **getprdfnam** each returns a pointer to an object with the following structure containing the broken-out fields of a line in the system default database. Each line in the database contains a *pr_default* structure, declared in the **<prot.h>** header file:

```
struct  system_default_fields  {
     time_t  fd_inactivity_timeout ;
     char    fd_boot_authenticate ;

} ;

struct  system_default_flags {
     unsigned short
             fg_inactivity_timeout:1,
             fg_boot_authenticate:1,

} ;
struct  pr_default  {
     char                          dd_name[20] ;
     char                          dg_name ;
     struct pr_field               prd ;
     struct pr_flag                prg ;
     struct t_field                tcd ;
     struct t_flag                 tcg ;
     struct dev_field              devd ;
     struct dev_flag               devg ;
     struct system_default_fields  sfld ;
     struct system_default_flags   sflg ;
} ;
```

Currently there is only one entry in the system default database referenced by name **default**.

The System Default database contains default values for all parameters in the Protected Password, Terminal Control, and Device Assignment databases, as well as configurable system-wide parameters. The fields from the other databases are described in the corresponding manual entries. *fd_inactivity_timeout* is the number of seconds until a session is terminated on trusted systems.

fd_boot_authenticate is a Boolean flag that indicates whether an authorized user must authenticate before the system begins operation.

getprdfent returns a pointer to the first *pr_default* structure in the database when first called. Thereafter, it returns a pointer to the next *pr_default* structure in the database so that successive calls can be used to search the database (only one entry is supported).

getprdfnam searches from the beginning of the file until a default entry 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. Currently, all programs access the default database by calling **getprdfnam** (the entry name is **default**).

A call to **setprdfent** has the effect of rewinding the default control file to allow repeated searches. **endprdfent** can be called to close the database when processing is complete.

putprdfnam puts a new or replaced default control entry *pr* with key *name* into the database. If the *prg.fd_name* field is 0, the requested entry is deleted from the system default database. **putprdfnam** locks the database for all update operations and performs an **endprdfent** after the update or failed attempt.

APPLICATION USAGE

In a multithreaded application, these routines are safe to be called only from one dedicated thread. These routines are not POSIX.1c async-cancel safe nor async-signal safe.

RETURN VALUE

getprdfent and **getprdfnam** return **NULL** pointers on **EOF** or error. **putprdfnam** returns 0 if it cannot add or update the entry.

WARNINGS

Do not delete the system default entry.

NOTES

The value returned by **getprdfent** and **getprdfnam** refers to a structure that is overwritten by calls to these routines. To retrieve an entry, modify it, and replace it in the database, copy the entry using structure assignment and supply the modified buffer to **putprdfnam**.

Programs using these routines must be compiled with **-lsec**.

FILES

/tcb/files/auth/system/default System Defaults database

SEE ALSO

authcap(4), default(4), getprpwent(3), getprtcent(3), getdvagent(3).

NAME

getprotoent(), getprotobynumber(), getprotobyname(), setprotoent(), endprotoent() - get protocol entry

SYNOPSIS

```
#include <netdb.h>
struct protoent *getprotoent(void);
struct protoent *getprotobyname(const char *name);
struct protoent *getprotobynumber(int proto);
int setprotoent(int stayopen);
int endprotoent(void);
_XOPEN_SOURCE_EXTENDED only
void setprotoent(int stayopen);
void endprotoent(void);
```

MULTITHREAD USAGE

Thread Safe:	Yes
Cancel Safe:	Yes
Async-cancel Safe:	No
Async-signal Safe:	No

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

DESCRIPTION

The **getprotoent()**, **getprotobyname()**, and **getprotobynumber()** functions each return a pointer to a structure of type *protoent* containing the broken-out fields of a line in the network protocol data base, **/etc/protocols**.

The members of this structure are:

p_name The official name of the protocol.

p_aliases A null-terminated list of alternate names for the protocol.

p_proto The protocol number.

Functions behave as follows:

getprotoent() Reads the next line of the file, opening the file if necessary.

setprotoent() Opens and rewinds the file. If the *stayopen* flag is non-zero, the protocol data base is not closed after each call to **getprotoent()** (either directly or indirectly through one of the other **getproto*** calls).

endprotoent() Closes the file.

getprotobyname(), **getprotobynumber()** Each sequentially searches from the beginning of the file until a matching protocol name (among either the official names or the aliases) or protocol number is found, or until EOF is encountered.

In a multithreaded application, **getprotoent()**, **getprotobyaddr()**, and **getprotobyname()** use thread-specific storage that is re-used in each call. The return value **struct protoent** should be unique for each thread and should be saved, if desired, before the thread makes the next **getproto*()** call.

For enumeration in multithreaded applications, the position within the enumeration is a process-wide property shared by all threads. **setprotoent()** may be used in a multithreaded application, but resets the enumeration position for all threads. **getprotoent()** enumerates protocol entries: successive calls to **getprotoent()** will return either successive protocol entries or NULL. If multiple threads interleave calls to **getprotoent()**, the threads will enumerate disjoint subsets of the protocol database.

If the system is running the Network Information Service (NIS) services or Network Information Service Plus (NIS+) services, **getprotobyname()** and **getprotobynumber()** get the protocol information

from the NIS server (see *ypserv*(1M) and *ypfiles*(4)) or NIS+ server (see *nis+*(1)), respectively.

Name Service Switch-Based Operation

The library routines **getprotobyname()**, **getprotobynumber()**, **getprotoent()**, and their reentrant counterparts, internally call the name service switch to access the "protocols" database lookup policy configured in the **/etc/nsswitch.conf** file (see *nsswitch.conf*(4)). The lookup policy defines the order and the criteria of the supported name services used to resolve protocol names and numbers.

RETURN VALUE

getprotoent(), **getprotobyname()**, and **getprotobynumber()** return a null pointer (0) on EOF or when they are unable to open **/etc/protocols**.

OBSOLESCENT INTERFACES

```
int getprotoent_r(struct protoent *result,
                  struct protoent_data *buffer);

int getprotobyname_r(const char *name,
                     struct protoent *result,
                     struct protoent_data *buffer);

int getprotobynumber_r(int proto,
                       struct protoent *result,
                       struct protoent_data *buffer);

int setprotoent_r(int stayopen, struct protoent_data *buffer);

int endprotoent_r(struct protoent_data *buffer);
```

The above reentrant interfaces have been moved from **libc** to **libd4r**. They are included to support existing applications and may be removed in a future release. New multithreaded applications should use the regular APIs without **_r**.

The reentrant interfaces performs the same operation as their regular counterpart (those without the **_r** suffix.) However, **getprotoent_r()**, **getprotobyname_r()**, and **getprotobyport_r()** expect to be passed the address of a **struct protoent** parameter and will store the address of the result at this supplied parameter. An additional parameter, an address to *struct protoent_data*, which is defined in the file **<netdb.h>** cannot be a NULL pointer.

The reentrant routines return **-1** if the operation is unsuccessful, or, in the case of **getprotoent_r()**, if the end of the services list has been reached. **0** is returned otherwise.

EXAMPLES

The following code excerpt counts the number of protocols entries:

```
int count = 0;

(void) setprotoent(0);
while (getprotoent() != NULL)
     count++;
(void) endprotoent();
```

WARNINGS

Programs that use the interfaces described in this manpage cannot be linked statically because the implementations of these functions employ dynamic loading and linking of shared objects at run time.

AUTHOR

getprotoent() was developed by Sun Microsystems Inc.

FILES

/etc/protocols

SEE ALSO

ypserv(1M), protocols(4), nis+(1), nsswitch.conf(4), ypfiles(4).

STANDARDS CONFORMANCE

getprotoent(): XPG4

NAME

getprpwent, getprpwuid, getprpwnam, getprpwaid, setprpwent, endprpwent, putprpwnam - manipulate protected password database entries (for trusted systems only).

SYNOPSIS

```
#include <sys/types.h>
#include <hpsecurity.h>
#include <prot.h>

struct pr_passwd *getprpwent(void);

struct pr_passwd *getprpwuid(uid_t uid);

struct pr_passwd *getprpwnam(const char *name);

struct pr_passwd *getprpwaid(aid_t aid)

void setprpwent(void);

void endprpwent(void);

int putprpwnam(const char *name, struct pr_passwd *pr);
```

DESCRIPTION

getprpwent, **getprpwuid**, **getprpwaid**, and **getprpwnam** each returns a pointer to a *pr_passwd* structure containing the broken-out fields of a line in the protected password database. Each line in the database contains a *pr_passwd* structure, declared in the <**prot.h**> header file:

```
struct pr_field  {
     /* Identity: */
     char    fd_name[9];      /* uses 8 character maximum(and NULL) from utmp */
     uid_t   fd_uid;          /* uid associated with name above */
     char    fd_encrypt[xxx]; /* encrypted password */
     char    fd_owner[9];     /* if a pseudo-user, the user accountable */
     char    fd_boot_auth;    /* boot authorization */
     mask_t  fd_auditcntl;    /* reserved */
     mask_t  audit_reserve1;  /* reserved */
     mask_t  fd_auditdisp;    /* reserved */
     mask_t  audit_reserve2;  /* reserved */
     aid_t   fd_pw_audid;     /* audit ID */
     int     fd_pw_audflg;    /* audit flag */

     /* Password maintenance parameters: */
     time_t  fd_min;          /* minimum time between password changes */
     int     fd_maxlen;       /* maximum length of password */
     time_t  fd_expire;       /* expiration time duration in secs */
     time_t  fd_lifetime;     /* account death duration in seconds */
     time_t  fd_schange;      /* last successful change in secs past 1/1/70 */
     time_t  fd_uchange;      /* last unsuccessful change */
     time_t  fd_acct_expire;  /* absolute account lifetime in seconds */
     time_t  fd_max_llogin;   /* max time allowed between logins */
     time_t  fd_pw_expire_warning; /* password expiration warning */
     uid_t   fd_pswduser;     /* who can change this user's password */
     char    fd_pick_pwd;     /* can user pick his own passwords? */
     char    fd_gen_pwd;      /* can user get passwords generated for him? */
     char    fd_restrict;     /* should generated passwords be restricted? */
     char    fd_nullpw;       /* is user allowed to have a NULL password? */
     uid_t   fd_pwchanger;    /* who last changed user's password */
     long    fd_pw_admin_num; /* password generation verifier */
     char    fd_gen_chars;    /* can have password of random ASCII? */
     char    fd_gen_letters;  /* can have password of random letters? */
     char    fd_tod[AUTH_TOD_SIZE];    /* times when user may login */

     /* Login parameters: */
     time_t  fd_slogin;       /* last successful login */
     time_t  fd_ulogin;       /* last unsuccessful login */
     char    fd_suctty[14];   /* tty of last successful login */
```

```
     int     fd_nlogins;      /* consecutive unsuccessful logins */
     char    fd_unsuctty[14]; /* tty of last unsuccessful login */
     int     fd_max_tries;    /* maximum unsuc login tries allowed */
     char    fd_lock;         /* Unconditionally lock account? */
};

struct pr_flag  {
     unsigned short
          /* Identity: */
          fg_name:1,              /* Is fd_name set? */
          fg_uid:1,               /* Is fd_uid set? */
          fg_encrypt:1,           /* Is fd_encrypt set? */
          fg_owner:1,             /* Is fd_owner set? */
          fg_boot_auth:1,         /* Is fd_boot_auth set? */
          fg_pw_audid:1,          /* Is fd_auditcntl set? */
          fg_pw_audflg:1,         /* Is fd_auditdisp set? */

          /* Password maintenance parameters: */
          fg_min:1,               /* Is fd_min set? */
          fg_maxlen:1,            /* Is fd_maxlen set? */
          fg_expire:1,            /* Is fd_expire set? */
          fg_lifetime:1,          /* Is fd_lifetime set? */
          fg_schange:1,           /* Is fd_schange set? */
          fg_uchange:1,           /* Is fd_fchange set? */
          fg_acct_expire:1,       /* Is fd_acct_expire set? */
          fg_max_llogin:1,        /* Is fd_max_llogin set? */
          fg_pw_expire_warning:1, /* Is fd_pw_expire_warning set? */
          fg_pswduser:1,          /* Is fd_pswduser set? */
          fg_pick_pwd:1,          /* Is fd_pick_pwd set? */
          fg_gen_pwd:1,           /* Is fd_gen_pwd set? */
          fg_restrict:1,          /* Is fd_restrict set? */
          fg_nullpw:1,            /* Is fd_nullpw set? */
          fg_pwchanger:1,         /* Is fd_pwchanger set? */
          fg_pw_admin_num:1,      /* Is fd_pw_admin_num set? */
          fg_gen_chars:1,         /* Is fd_gen_chars set? */
          fg_gen_letters:1,       /* Is fd_gen_letters set? */
          fg_tod:1,               /* Is fd_tod set? */

          /* Login parameters: */
          fg_slogin:1,            /* Is fd_slogin set? */
          fg_suctty: 1,           /* is fd_suctty set ? */
          fg_unsuctty: 1,         /* is fd_unsuctty set ? */
          fg_ulogin:1,            /* Is fd_ulogin set? */
          fg_nlogins:1,           /* Is fd_nlogins set? */
          fg_max_tries:1,         /* Is fd_max_tries set? */
          fg_lock:1;              /* Is fd_lock set? */

};

struct pr_passwd  {
     struct pr_field ufld;   /* user specific fields */
     struct pr_flag uflg;    /* user specific flags */
     struct pr_field sfld;   /* system wide fields */
     struct pr_flag sflg;    /* system wide flags */
};
```

The protected password database stores user authentication profiles. The *pr_passwd* structure in the user-specific entry refers to parameters specific to a user. The *pr_passwd* structure in the system default database sets parameters that are used when there is no user-specific override.

The user-specific entry is keyed on the *fd_name* field, which is a cross reference to the **/etc/passwd** or the Network Information Service Plus (NIS+) **passwd** table entry for the user. The *fd_uid* field must match the UID in that file or the NIS+ **passwd** table as well. The *fd_encrypt* field is the encrypted password. The password is encrypted in eight character segments, so the size of this field is a multiple of the number of characters in an encrypted segment (**AUTH_CIPHERTEXT_SIZE** macro).

fd_owner is the user name accountable for the account. The *fd_boot_auth* field is used when the system default file specifies boot authorization is required. *init*(1M) prompts for a user name and password. If the authentication succeeds, a value in this field allows the user to continue the system boot process.

fd_min is the time, in seconds, that must elapse before the user can change passwords. *fd_maxlen* is the maximum password length (in characters) for the user. *fd_expire* is the time, in seconds, until the user's password expires. *fd_lifetime* is the number of seconds that must elapse before the password dies. The account is considered locked if the password is dead.

fd_schange and *fd_uchange* record the last successful and unsuccessful password change times.

The *fd_acct_expire* field specifies the absolute period of time in seconds that the account can be used. An absolute expiration date may be specified, which is then converted into seconds stored in this field. This is different from *fd_expire* in that *fd_acct_expire* specifies an absolute expiration date, while *fd_expire* is reset with each password change.

fd_max_llogin specifies the maximum time in seconds allowed since the last login before the account becomes locked. *fd_pw_expire_warning* is the time in seconds before the end of *fd_expire* that the system warns the user the password is about to expire. *fd_pswduser* stores the user ID of the user allowed to change passwords for the account. Typically, this is the account owner.

The next flag fields control password generation. *fd_pick_pwd*, if set, allows the user to pick his or her own password. *fd_nullpw*, if set, allows the account to be used without a password. *fd_gen_pwd* enables the use of the random pronounceable password generator for passwords for this account. *fd_gen_chars* and *fd_gen_letters* allow the password generator to generate passwords composed of random printable characters and random letters, neither of which is easy to remember. The password change software allows the user to pick from whichever options are available for his or her account. One of these three fields (*fd_gen_pwd*, *fd_gen_chars*, or *fd_gen_letters*) must be set.

fd_pwchanger is the user ID of the user who last changed the password on the user's account, if it was not the account owner. *fd_restrict*, if set, causes triviality checks to be made after the account password has been chosen to avoid palindromes, user name and machine name permutations, and words appearing in the dictionary.

The *fd_tod* specifier is a string, formatted like the UUCP Systems file, which specifies time intervals during which the user can log in.

The next fields are used to protect against login spoofing, listing the time and location of last login. *fd_slogin* and *fd_ulogin* are time stamps of the last successful and unsuccessful login attempts. *fd_suctty* and *fd_unsuctty* are the terminal device or (if supported) host names of the terminal or host from which the last login attempt occurred.

fd_nlogins is the number of unsuccessful login attempts since the last successful login. It is reset to zero after a successful login. *fd_max_tries* is the number of unsuccessful attempts until the account is considered locked.

fd_lock indicates whether the administrative lock on the account is set. The account is considered disabled (locked) if one or more of these activities has occurred:

1. if the password is dead,
2. if the maximum number of unsuccessful attempts has been exceeded,
3. if the administrative lock is set,
4. if the account expiration is reached, or
5. if the time since last login is exceeded.

When **getprpwent** is first called, it returns a pointer to the first user *pr_passwd* structure in the database; thereafter, it returns a pointer to the next *pr_passwd* structure in the database so that successive calls can be used to search the database. Note that entries without a corresponding entry in **/etc/passwd** are skipped. However, if NIS+ is configured, the entries are *not* skipped for users that have an entry in the NIS+ **passwd** table *and* the local protected database. A local protected database entry is created at login time for each NIS+ user that does not have an entry in the local protected database. The entries are scanned in the order they appear in **/etc/passwd** or in the NIS+ **passwd** table if NIS+ is configured and if the **nsswitch.conf** file refers to NIS+ first (for example, an entry in **nsswitch.conf** would contain **passwd: nisplus files**).

getprpwuid searches from the beginning of the database until a numerical user ID matching *uid* is found and returns a pointer to the particular structure in which it was found. **getprpwaid** functions like **getprpwuid** only it uses the audit ID instead of the UID.

getprpwnam searches from the beginning of the database 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 **setprpwent** has the effect of rewinding the protected password database to allow repeated searches. **endprpwent** can be called to close the protected password database when processing is complete.

putprpwnam puts a new or replaces a protected password entry *pr* with key *name* into the database. If the *uflg.fg_name* field is 0, the requested entry is deleted from the protected password database. **putprpwnam** locks the database for all update operations, and performs a **endprpwent** after the update or failed attempt. For NIS+, this function will add or remove protected password information from the **passwd** table and/or trusted table and/or the local protected database.

NOTES

The value returned by **getprpwent** and **getprpwnam** refers to a structure that is overwritten by calls to these routines. To retrieve an entry, modify it, and replace it in the database, copy the entry using structure assignment and supply the modified buffer to **putprpwnam**.

On systems supporting network connections, the *fd_suctty* and *fd_unsuctty* fields can be the ASCII representation of the network address of the host from which the last successful or unsuccessful remote login to the account occurred. Use **getdvagnam** (see *getdvagent*(3)) to investigate the type of device to determine whether a host or a terminal was used for the last successful or unsuccessful login.

Programs using these routines must be compiled with **-lsec**.

If you link your application with the archive version of **libsec** (**libsec.a**), these routines work independent of NIS+ or the Name Service Switch. The protected password database exists only in the local system; for example, **/tcb** only and not the NIS+ **passwd** or trusted table.

getprpwent assumes one name per UID and one UID per name. The sequential scan loops between the first two instances of a multiple UID.

getprpwent uses *getpwent*(3C) routines to sequentially scan databases. User program references to password entries obtained using *getpwent*(3C) routines will not be valid after using any routines described here (that is, the * **prp*** routines).

If NIS+ is configured in your system, the protected password information can be stored in three different repositories:

1. The NIS+ **passwd** Table for the local domain.
2. The NIS+ Trusted Table for the local domain.
3. The Local Protected Database file.

Additionally, all of these routines depend on the configuration of the Name Service Switch file, **/etc/nsswitch.conf**. These routines use the switch for the **passwd** database.

APPLICATION USAGE

In a multithreaded application, these routines are safe to be called only from one dedicated thread. These routines are not POSIX.1c async-cancel safe nor async-signal safe. In an NIS+ namespace, the user should be aware of the configuration of the **/etc/nsswitch.conf** file because protected password information is stored in three different repositories: **passwd** table, trusted table, and local database. The Name Service Switch database used for the protected database API is **passwd** (for example, an entry in **/etc/nsswitch.conf** would contain **passwd: nisplus files**).

RETURN VALUE

getprpwent, **getprpwuid**, **getprpwaid**, and **getprpwnam** return NULL pointers on EOF or error. **putprpwnam** returns 0 if it cannot add or update the entry.

FILES

/etc/passwd	System Password file
/tcb/files/auth/*/*	Protected Password database
/tcb/files/auth/system/default	System Defaults database

NIS+ TABLES

passwd, trusted

SEE ALSO
authcap(4), getpwent(3C), getprdfent(3), prpwd(4), ttsyncd(1M), nis+(1).

NAME

getprtcent, getprtcnam, setprtcent, endprtcent, putprtcnam - manipulate terminal control database entry for a trusted system

SYNOPSIS

```
#include <sys/types.h>
#include <hpsecurity.h>
#include <prot.h>

struct pr_term *getprtcent(void);

struct pr_term *getprtcnam(const char *name);

void setprtcent(void);

void endprtcent(void);

int putprtcnam(const char *name, struct pr_term *pr);
```

DESCRIPTION

getprtcent and **getprtcnam** each returns a pointer to an object with the following structure containing the broken-out fields of an entry in the terminal control database. Each entry in the database contains a *pr_term* structure, declared in the **<prot.h>** header file:

```
struct   t_field  {
   char    fd_devname[14];      /* Terminal (or host) name */
   uid_t   fd_uid;              /* uid of last successful login */
   time_t  fd_slogin;           /* time stamp of successful login */
   uid_t   fd_uuid;             /* uid of last unsuccessful login */
   time_t  fd_ulogin;           /* time stamp of unsuccessful login */
   int     fd_nlogins;          /* consecutive failed attempts */
   int     fd_max_tries;        /* maximum unsuc login tries allowed */
   time_t  fd_logdelay;         /* delay between login tries */
   char    fd_lock;             /* terminal locked? */
   int     fd_login_timeout;    /* login timeout in seconds */
};

struct  t_flag  {
   unsigned short
      fg_devname:1,             /* Is fd_devname set? */
      fg_uid:1,                 /* Is fd_uid set? */
      fg_slogin:1,              /* Is fd_stime set? */
      fg_uuid:1,                /* Is fd_uuid set? */
      fg_ulogin:1,              /* Is fd_ftime set? */
      fg_nlogins:1,             /* Is fd_nlogins set? */
      fg_max_tries:1,           /* Is fd_max_tries set? */
      fg_logdelay:1,            /* Is fd_logdelay set? */
      fg_lock:1,                /* Is fd_lock set? */
      fg_login_timeout:1        /* is fd_login_timeout valid? */
      ;
};

struct  pr_term  {
   struct t_field ufld;
   struct t_flag uflg;
   struct t_field sfld;
   struct t_flag sflg;
};
```

The system stores the user ID and time of the last successful login (*fd_uid* and *fd_slogin)* and unsuccessful login (*fd_uuid* and *fd_ulogin)* in the appropriate Terminal Control database entry. The system increments *fd_nlogins* with each unsuccessful login, and resets the field to 0 on a successful login. The *fd_max_tries* field is a limit on the number of unsuccessful logins until the account is locked. An administrative lock can also be applied, indicated by a non-zero *fd_lock* field. *fd_logdelay* stores the amount of time (in seconds) that the system waits between unsuccessful login attempts, and *fd_login_timeout* stores the number of seconds from the beginning of an authentication attempt until the login attempt is terminated.

Note that *ufld* and *uflg* refer to user-specific entries, and *sfld* and *sflg* refer to the system default values (see *authcap*(4)).

The value returned by **getprtcent** or **getprtcnam** refers to a structure that is overwritten by calls to these routines. To retrieve an entry, modify it, and replace it in the database, copy the entry using structure assignment and supply the modified buffer to **putprtcnam**.

getprtcent returns a pointer to the first terminal *pr_term* structure in the database when first called. Thereafter, it returns a pointer to the next *pr_term* structure in the database, so successive calls can be used to search the database. **getprtcnam** searches from the beginning of the database until a terminal 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 **setprtcent** has the effect of rewinding the Terminal Control database to allow repeated searches. **endprtcent** can be called to close the Terminal Control database when processing is complete.

putprtcnam puts a new or replaced terminal control entry *pr* with key *name* into the database. If the *fg_devname* field is 0, the requested entry is deleted from the Terminal Control database. **putprtcnam** locks the database for all update operations, and performs an **endprtcent** after the update or failed attempt.

g

APPLICATION USAGE

In a multithreaded application, these routines are safe to be called only from one dedicated thread. These routines are not POSIX.1c async-cancel safe nor async-signal safe.

RETURN VALUE

getprtcent and **getprtcnam** return **NULL** pointers on **EOF** or error. **putprtcnam** returns 0 if it cannot add or update the entry.

NOTES

The *fd_devname* field, on systems supporting connections, may refer to the ASCII representation of a host name. This can be determined by using **getdvagnam** (see *getdvagent*(3)) to interrogate the Device Assignment database as to the type of the device, passing in the *fd_devname* field of the Terminal Control structure as an argument. This allows lockout by machine, instead of the device (typically pseudo tty) on which the session originated.

Programs using these routines must be compiled with **-lsec**.

The *sfld* and *sflg* structures are filled from corresponding fields in the system default database. Thus, a program can easily extract the user-specific or system-wide parameters for each database field (see *getprpwent* and *getdvagent*).

FILES

/tcb/files/ttys	Terminal Control database
/tcb/files/auth/system/default	System Defaults database

SEE ALSO

getprdfent(3), authcap(4), ttys(4).

NAME

getpublickey(), getsecretkey(), publickey() - retrieve public or secret key

SYNOPSIS

```
#include <rpc/rpc.h>
#include <rpc/key_prot.h>

int getpublickey(
     const char  netname [MAXNETNAMELEN],
     char publickey[HEXKEYBYTES]);

int getsecretkey(
     const char  netname [MAXNETNAMELEN],
     char secretkey[HEXKEYBYTES],
     const char *passwd );
```

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

DESCRIPTION

getpublickey() and **getsecretkey()** get public and secret keys for *netname*. The key may come from one of the following sources: the **/etc/publickey** file (see *publickey*(4)) or the NIS map "publickey.byname" or the NIS table "cred.org_dir". The sources and their lookup order are specified in the **/etc/nsswitch.conf** file (see *nsswitch.conf*(4)).

getsecretkey() has an extra argument, *passwd*, used to decrypt the encrypted secret key stored in the database.

RETURN VALUES

Both routines return 1 if they are successful in finding the key, 0 (zero) otherwise. The keys are returned as NULL-terminated, hexadecimal strings. If the password supplied to **getsecretkey()** fails to decrypt the secret key, the routine will return 1 but the *secretkey* [0] will be set to NULL.

WARNINGS

If **getpublickey()** gets the public key from any source other than NIS+, all authenticated NIS+ operations may fail. To ensure that this does not happen, edit the **nsswitch.conf()** file to make sure that the public key is obtained from NIS+.

SEE ALSO

secure_rpc(3N), nsswitch.conf(4), publickey(4).

NAME

getpw() - get name from UID

SYNOPSIS

```
#include <pwd.h>
int getpw(uid_t 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. The line is null-terminated.

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

APPLICATION USAGE

getpw() is thread-safe. It is not async-cancel-safe. A cancellation point may occur when a thread is executing **getpw()**.

NETWORKING FEATURES

NFS

This routine is implemented using **getpwuid()** (see *getpwent*(3C)) and therefore uses the Network Information Service network database as described in *passwd*(4).

RETURN VALUE

getpw() returns non-zero on error.

WARNINGS

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

AUTHOR

getpw() was developed by AT&T and HP.

FILES

/etc/passwd

SEE ALSO

getpwent(3C), passwd(4).

STANDARDS CONFORMANCE

getpw(): XPG2

NAME

getpwent(), getpwuid(), getpwuid_r(), getpwnam(), getpwnam_r(), setpwent(), endpwent(), fgetpwent(), - get password file entry

SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwent(void);

struct passwd *getpwuid(uid_t uid);

int  getpwuid_r(uid_t uid, struct passwd *pwd, char *buffer,
     size_t buflen, struct passwd **result);

struct passwd *getpwnam(const char *name);

int  getpwnam_r(char *name, struct passwd *pwd, char *buffer,
     size_t buflen, struct passwd **result);

void setpwent(void);

void endpwent(void);

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

Obsolescent Interfaces

```
#include <pwd.h>

int  getpwent_r(struct passwd *result, char *buffer, int buflen,
     FILE **pwfp);

void setpwent_r(FILE **pwfp);

void endpwent_r(FILE **pwfp);

int fgetpwent_r(FILE *f, struct passwd *result, char *buffer,
     int buflen);
```

DESCRIPTION

getpwent(), **getpwuid()**, and **getpwnam()** are used to obtain password entries, and return a pointer to an object of **passwd** structure. An entry may come from any of the sources for **passwd** specified in the **/etc/nsswitch.conf** file. See *nsswitch.conf*(4).

The **passwd** structure is defined in <**pwd.h**> and includes the following members:

```
char    *pw_name;     /* user name */
char    *pw_passwd;   /* encrypted password */
uid_t   pw_uid;       /* user id */
gid_t   pw_gid;       /* group id */
char    *pw_age;      /* password aging */
char    *pw_comment;  /* unused */
char    *pw_gecos;    /* user fullname, office, extension, homephone*/
char    *pw_dir;      /* initial directory */
char    *pw_shell;    /* initial shell */
aid_t   pw_audid;     /* numerical audit id */
int     pw_audflg;    /* numerical audit flag */
```

The *pw_comment* field is unused; for more information on the other fields, refer to *passwd*(4).

getpwent() When first called, **getpwent**() returns a pointer to the first **passwd** structure in the password database. Thereafter, it returns a pointer to the next **passwd** structure in the database;

setpwent() Has the effect of rewinding the password database to allow repeated searches;

endpwent() Can be called to indicate that password database processing is complete;

getpwuid() Searches from the beginning of the password database until a numeric 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 password database until a login name matching *name* is found, and returns a pointer to the particular structure in which it was found;

fgetpwent() unlike the other funcions above, does not use **nsswitch.conf**. It returns a pointer to the next **passwd** structure in the standard I/O stream *stream*, which should be open for reading, and its contents should match the format of **/etc/passwd**.

Obsolescent Interfaces
getpwent_r(), **setpwent_r()**, **endpwent_r()**, **fgetpwent_r()** get password file entry.

Reentrant Interfaces
getpwuid_r(), and **getpwnam_r()** both update the **struct**passwd pointed to by **pwd** and store a pointer to that structure at the location pointed to by **result**. The structure shall contain an entry from the user database with a matching **uid** or **name**. Storage referenced by the passwd structure pointed to by **pwd** shall be allocated from the memory provided with the **buffer** parameter, which is **buflen** in size. The maximum size needed for this buffer can be determined with the **_SC_GETPW_R_SIZE_MAX sysconf()** parameter. A **NULL** pointer is returned at the location pointed to by **result** on error or if the requested entry is not found.

A buffer length of 1024 is recommended.

SECURITY FEATURES

If the system has been converted to a trusted system, the password, audit ID, and audit flag are not returned. The password will be the default * that is in **/etc/passwd** and the audit ID and audit flag will be set to –1. On trusted systems, if it is not necessary to obtain information from the regular password file, **/etc/passwd**, users should use **getprpwent**() to access the protected password database. See *getprpwent*(3) and *getspwent*(3X).

putpwent() affects only **/etc/passwd**; the audit ID and audit flag in the password structure are ignored. **putprpwnam()** must be used to modify the protected password databse entries. See *getprpwent*(3).

APPLICATION USAGE

getpwuid() and **getpwnam()** are not thread-safe. **getpwent()**, **setpwent()**, **endpwent()**, **fgetpwent()**, **getpwuid_r()**, and **getpwnam_r()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing any of these interfaces.

RETURN VALUE

getpwent(), **getpwuid()**, **getpwnam()**, and **fgetpwent()** return a **NULL** pointer if an end-of-file or error is encountered on reading. Otherwise, the return value points to an internal static area containing a valid **passwd** structure.

getpwuid_r() and **getpwnam_r()** return zero upon success. Otherwise, an error number is returned to indicate the error.

ERRORS

getpwent(), **getpwuid()**, **getpwnam()**, and **fgetpwent()** fail if any of the following are true:

[EIO] An I/O error has occurred.

[EMFILE] *OPEN_MAX* descriptors are currently open in the calling process.

[ENFILE] The maximum allowable number of files is currently open in the system.

WARNINGS

The value returned by **getpwent()**, **getpwuid()**, **getpwnam()**, and **fgetpwent()** points to a single static area that is overwritten by each call to any of the functions, so it must be copied if it is to be saved.

The following fields have numerical limitations as noted:

- The user ID is an integer value between -2 and UID_MAX inclusive.
- The group ID is an integer value between 0 and UID_MAX inclusive.

Users of **getpwuid_r()** and **getpwnam_r()** should note that these interfaces now conform with POSIX.1c. **getpwent_r()**, **setpwent_r()**, **endpwent_r()** and **fgetpwent_r()** are obsolescent interfaces. These interfaces and the old prototypes of **getpwuid_r()** and **getpwnam_r()** are

supported for compatibility with existing DCE applications only.

The interfaces **getpwuid()**, **getpwnam()**, **getpwent()**, **setpwent()**, **endpwent()**, **fgetpwent()**, **getpwuid_r()** and **getpwnam_r()** use the Dynamic Name Service Switch.(See *nsswitch.conf*(4).) An application that uses these interfaces cannot be fully archive bound.

EXAMPLE

The following code excerpt prints name and uid of user logged in on this terminal:

```
struct passwd pwd;
struct passwd *result;
char logBuffer [1024];
char pwdBuffer [1024];

if (getlogin_r (loginBuffer, 1024) == 0)
  if (getpwnam_r (logBuffer, &pwd, pwdBuffer, 1024, &result) == 0)
    printf ("Name = %s; uid = %d\n", pwd.pw_name, pwd.pw_uid);
```

DEPENDENCIES

NFS

Files

/var/yp/*domainname***/passwd.byname**
/var/yp/*domainname***/passwd.byuid**
/var/nis/*hostname***/passwd.org_dir**

See Also

niscat(1), ypcat(1).

AUTHOR

getpwent(), **getpwuid()**, **getpwnam()**, **setpwent()**, **endpwent()**, and **fgetpwent()** were developed by Sun and HP.

FILES

/etc/passwd System Password file

SEE ALSO

niscat(1), ypcat(1), cuserid(3S), getgrent(3C), getlogin(3C), getprpwent(3), getspwent(3X), stdio(3S), putpwent(3C), nsswitch.conf(4), passwd(4), limits(5).

STANDARDS CONFORMANCE

getpwent(): SVID2, SVID3, XPG2

endpwent(): SVID2, SVID3, XPG2

fgetpwent(): SVID2, SVID3, XPG2

getpwnam(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

getpwuid(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

setpwent(): SVID2, SVID3, XPG2

NAME

getrpcent(), getrpcbyname(), getrpcbynumber(), - get rpc entry

SYNOPSIS

cc [*flag* ...] *file* ... **-lnsl** [*library* ...]

```
#include <netdb.h>
struct rpcent *getrpcent();
struct rpcent *getrpcbyname(char *name);
struct rpcent *getrpcbynumber(int number);
int setrpcent(int stayopen);
int endrpcent();
```

DESCRIPTION

These functions are used to obtain entries for RPC (Remote Procedure Call) services. An entry may come from any of the sources for **rpc** specified in the **/etc/nsswitch.conf** file (see *nsswitch.conf*(4)).

getrpcbyname() searches for an entry with the RPC service name specified by the parameter *name*.

getrpcbynumber() searches for an entry with the RPC program number *number*.

The functions **setrpcent()**, **getrpcent()**, and **endrpcent()** are used to enumerate RPC entries from the database.

setrpcent() sets (or resets) the enumeration to the beginning of the set of RPC entries. This function should be called before the first call to **getrpcent()**. Calls to **getrpcbyname()** and **getrpcbynumber()** leave the enumeration position in an indeterminate state. If the *stayopen* flag is non-zero, the system may keep allocated resources such as open file descriptors until a subsequent call to **endrpcent()**.

Successive calls to **getrpcent()** return either successive entries or NULL, indicating the end of the enumeration.

endrpcent() may be called to indicate that the caller expects to do no further RPC entry retrieval operations; the system may then deallocate resources it was using. It is still allowed, but possibly less efficient, for the process to call more RPC entry retrieval functions after calling **endrpcent()**.

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

getrpcbyname(), **getrpcbynumber()** and **getrpcent()** use thread specific storage that is re-used in each call. The return value, **struct rpcent**, should be unique for each thread and should be saved, if desired, before the thread makes the next **getrpc*()** call.

For enumeration in multithreaded applications, the position within the enumeration is a process-wide property shared by all threads. **setrpcent()** may be used in a multithreaded application but resets the enumeration position for all threads. If multiple threads interleave calls to **getrpcent()**, the threads will enumerate disjoint subsets of the RPC entry database.

RETURN VALUE

RPC entries are represented by the **struct rpcent** structure defined in **<netdb.h>**:

```
struct rpcent {
   char *r_name; /* name of this rpc service */
   char **r_aliases; /* zero-terminated list of alternate names */
```

```
    long r_number; /* rpc program number */
};
```

The functions **getrpcbyname()**, and **getrpcbynumber()** each return a pointer to a **struct rpcent** if they successfully locate the requested entry; otherwise they return NULL.

The function **getrpcent()** returns a pointer to a **struct rpcent** if it successfully enumerates an entry; otherwise it returns NULL, indicating the end of the enumeration.

WARNINGS

Programs that use the interfaces described in this manual page cannot be linked statically since the implementations of these functions employ dynamic loading and linking of shared objects at run time.

AUTHOR

getrpcent() was developed by Sun Microsystems, Inc.

FILES

/etc/rpc /etc/nsswitch.conf

SEE ALSO

rpcinfo(1M), rpc(4), nsswitch.conf(4).

NAME

getrpcport() - get RPC port number

SYNOPSIS

```
int getrpcport(
      char *host,
      int prognum,
      int versnum,
      int proto
);
```

DESCRIPTION

getrpcport() returns the port number for version *versnum* of the RPC program *prognum* running on *host* and using protocol *proto*. It returns **0** if it cannot contact *portmap* or if *prognum* is not registered. If *prognum* is registered but not with version *versnum*, it returns the port number of the last registered (*prognum*, *proto*) pair.

WARNINGS

User applications that call this routine must be linked with **/usr/lib/librpcsvc.a**. For example,

```
cc my_source.c -lrpcsvc
```

AUTHOR

getrpcport() was developed by Sun Microsystems, Inc.

SEE ALSO

portmap(1M).

NAME

gets(), fgets() - get a string from a stream

SYNOPSIS

```
#include <stdio.h>
char *gets(char *s);
char *fgets(char *s, int n, FILE *stream);
```

Obsolescent Interface

```
char *fgets_unlocked(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, 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.

Obsolescent Interface

fgets_unlocked() gets a string from a stream.

APPLICATION USAGE

gets() and **fgets()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing **gets()** or **fgets()**.

RETURN VALUE

Upon sucessful completion, **fgets()**, **fgets_unlocked()**, and **gets()** return *s*. If the stream is at end-of-file, the end-of-file indicator for the stream is set and a null pointer is returned. If a read error occurs, the error indicator for the stream is set, **errno** is set to indicate the error, and a null pointer is returned.

ferror() and **feof()** can be used to distinguish between an error condition and an end-of-file condition.

ERRORS

fgets(), **fgets_unlocked()**, and **gets()** fail if data needs to be read into the *stream*'s buffer, and:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying *stream* and the process would be delayed in the read operation.

[EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for reading.

[EINTR] The read operation was terminated due to the receipt of a signal, and either no data was transferred or the implementation does not report partial transfer for this file.

[EIO] The process is a member of a background process and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the **SIGTTIN** signal or the process group of the process is orphaned.

Additional **errno** values can be set by the underlying **read()** function (see *read*(2)).

WARNINGS

fgets_unlocked() is an obsolescent interface supported only for compatibility with existing DCE applications. New multithreaded applications should use **fgets()**.

SEE ALSO

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

STANDARDS CONFORMANCE

gets(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

fgets(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

getservent(), getservbyport(), getservbyname(), setservent(), endservent() - get service entry

SYNOPSIS

```
#include <netdb.h>

struct servent *getservent(void);

struct servent *getservbyname(const char *name,
                              const char *proto);

struct servent *getservbyport(int port, const char *proto);

int setservent(int stayopen);

int endservent(void);

_XOPEN_SOURCE_EXTENDED only
void setservent(int stayopen);
void endservent(void);
```

MULTITHREAD USAGE

Thread Safe:	Yes
Cancel Safe:	Yes
Async-cancel Safe:	No
Async-signal Safe:	No

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

DESCRIPTION

The **getservent()**, **getservbyname()**, and **getservbyport()** functions each return a pointer to a structure of type *servent* containing the broken-out fields of a line in the network services data base, **/etc/services**.

The members of this structure are:

s_name	The official name of the service.
s_aliases	A null-terminated list of alternate names for the service.
s_port	The port number at which the service resides.
s_proto	The name of the protocol to use when contacting the service.

Functions behave as follows:

getservent()	Reads the next line of the file, opening the file if necessary.
setservent()	Opens and rewinds the file. If the *stayopen* flag is non-zero, the services data base is not closed after each call to **getservent()** (either directly or indirectly through one of the other **getserv*** calls).
endservent()	Closes the file.
getservbyname() **getservbyport()**	Each sequentially searches from the beginning of the file until a matching service name (among either the official names or the aliases) or port number is found, or until EOF is encountered. If a non-NULL protocol name is also supplied (such as **tcp** or **udp**), searches must also match the protocol.

If the system is running the Network Information Service (NIS) or Network Information Service Plus (NIS+), **getservbyname()** gets the service information from the NIS server (see *ypserv*(1M) and *ypfiles*(4)) or from the NIS+ server (see *nis+*(1)), respectively.

In a multithreaded application, **getservent()**, **getservbyaddr()**, and **getservbyname()** use thread-specific storage that is re-used in each call. The return value **struct servent** should be unique for each thread and should be saved, if desired, before the thread makes the next **getserv*()** call.

For enumeration in multithreaded applications, the position within the enumeration is a process-wide property shared by all threads. **setservent()** may be used in a multithreaded application, but resets the enumeration position for all threads. If multiple threads interleave calls to **getservent()**, the threads will enumerate disjoint subsets of the service database.

Name Service Switch-Based Operation

The library routines **getservbyname()**, **getservbyport()**, **getservent()**, and their reentrant counterparts, internally call the name service switch to access the "services" database lookup policy configured in the **/etc/nsswitch.conf** file (see *nsswitch.conf*(4)). The lookup policy defines the order and the criteria of the supported name services used to resolve service names and ports.

OBSOLESCENT INTERFACES

```
int getservent_r(struct servent *result,
                 struct servent_data *buffer);

int getservbyname_r(const char *name,
                    const char *proto,
                    struct servent *result,
                    struct servent_data *buffer);

int getservbyport_r(int port,
                    const char *proto,
                    struct servent *result,
                    struct servent_data *buffer);

int setservent_r(int stayopen, struct servent_data *buffer);

int endservent_r(struct servent_data *buffer);
```

The above reentrant interfaces have been moved from **libc** to **libd4r**. They are included to support existing applications and may be removed in a future release. New multithreaded applications should use the regular APIs without the **_r** suffix.

The reentrant interfaces performs the same operations as their regular counterpart (those without the **_r** suffix.) However, **getservent_r()**, **getservbyname_r()**, and **getservbyport_r()** expect to be passed the address of a **struct servent** and will store the address of the result at this supplied parameter. An additional parameter, an address to a *struct servent_data*, which is defined in the file **<netdb.h>** cannot be a NULL pointer.

The reentrant routines return **−1** if the operation is unsuccessful, or, in the case of **getservent_r()**, if the end of the services list has been reached. **0** is returned otherwise.

RETURN VALUE

getservent(), **getservbyname()**, and **getservbyport()** return a null pointer (0) on EOF or when they are unable to open **/etc/services**.

EXAMPLES

The following code excerpt counts the number of service entries:

```
int count = 0;

(void) setservent(0);
while (getservent() != NULL)
     count++;
(void) endservent();
```

WARNINGS

Programs that use the interfaces described in this manpage cannot be linked statically because the implementations of these functions employ dynamic loading and linking of shared objects at run time.

AUTHOR

getservent() was developed by Sun Microsystems Inc.

FILES

/etc/services

SEE ALSO
ypserv(1M), services(4), ypfiles(4), nsswitch.conf(4), nis+(1).

STANDARDS CONFORMANCE
`getservent()`: XPG4

NAME

getspent, **getspnam**, **setspent**, **endspent** - access secure password entries, for trusted systems only.

SYNOPSIS

```
#include <shadow.h>
struct spwd * getspent (void);
struct spwd * getspnam (const char *name);
void setspent (void);
void endspent (void);
```

DESCRIPTION

The routines **getspent()** and **getspnam()** return a pointer to the next secured password entry. Each entry is a **spwd** structure, declared in the **shadow.h** header file with the following members:

```
char   *sp_namp;   /* the user's login name */
char   *sp_pwdp;   /* the encrypted password for the user */
long   sp_lstchg;  /* # of days from 1/1/70 when passwd was last modified */
long   sp_min;     /* min # of days allowed between password changes */
long   sp_max;     /* max # of days allowed between password changes */
long   sp_warn;    /* # of days before password expires and warning issued*/
long   sp_inact;   /* # of days between account inactive and disabled */
long   sp_expire;  /* # of days from 1/1/70 when account is locked */
unsigned long   sp_flag; /* currently unused */
```

The **getspent()** routine returns a pointer to the first **spwd** structure when first called. Subsequent calls return pointers to successive **spwd** structures. Repeated calls to **getspent()** can be used to search all entries in the protected password database. The **getspnam()** routine searches password entries from beginning to end until a login name matching *name* is found, and returns a pointer to that entry.

If the fields corresponding to **sp_min**, **sp_max**, **sp_lstchg**, **sp_warn**, **sp_inact**, **sp_expire**, or **sp_flag** are not specified in the entry, they default to -1. If an end-of-file or an error is encountered in reading or a format error is detected, these functions return a null pointer and; for an error, **errno** is set to **EINVAL**.

The **setspent()** routine is used to reset access to the secured password entries. After **setspent()** is called, the subsequent call to **getspent()** returns the first secured password entry. This mechanism is used to allow repeated searches of the secured password entries. The **endspent()** routine is used to indicate that processing of secured password entries is complete.

getspent() is only supported on trusted systems.

The secured password facility is implemented without the use of the **/etc/shadow** file. **getspent()**, **getspnam()**, **setspent()**, and **endspent()** read from the trusted system's protected password database (**/tcb/files/auth/*/***) and not **/etc/shadow**. The file **/etc/shadow** is not used in any way by the HP-UX login facility.

These routines return a null pointer and sets ERRNO to ENOENT if the system has not been converted to trusted system. In all other cases, the return value is set similarly to **getprpwent()**. See *getprpwent*(3) for more information.

Programs using these routines must be compiled with **-lsec**.

DIAGNOSTICS

getspent(), **getspnam()**, and **fgetspent()** return a null pointer on EOF or error.

FILES

/etc/passwd	System Password file.
/tcb/files/auth/*/*	Protected password database, for trusted systems.

SEE ALSO

getpwent(3C), getprpwent(3), passwd(4).

STANDARDS CONFORMANCE

getspent : SVID3

NAME

getspwent(), getspwuid(), getspwaid(), getspwnam(), setspwent(), endspwent(), fgetspwent() - get secure password file entry on trusted systems

SYNOPSIS

```
#include <pwd.h>
struct s_passwd *getspwent(void);
struct s_passwd *getspwuid(uid_t uid);
struct s_passwd *getspwaid(aid_t aid);
struct s_passwd *getspwnam(const char *name);
void setspwent(void);
void endspwent(void);
struct s_passwd *fgetspwent(FILE *stream);
```

Deprecated Interfaces

The use of the following re-entrant interfaces is deprecated: **getspwent_r()**, **getspwuid_r()**, **getspwaid_r()**, **getspwnam_r()**, **setspwent_r()**, **endspwent_r()**, **fgetspwent_r()**

```
#include <pwd.h>
int  getspwent_r(struct s_passwd *result, char *buffer, int buflen,
     FILE **pwfp);
int  getspwuid_r(uid_t uid, struct s_passwd *result,
     char *buffer, int buflen);
int  getspwaid_r(aid_t aid, struct s_passwd *result,
     char *buffer, int buflen);
int  getspwnam_r(char *name, struct s_passwd *result,
     char *buffer, int buflen);
void setspwent_r(FILE **pwfp);
void endspwent_r(FILE **pwfp);
int  fgetspwent_r(FILE *f, struct s_passwd *result,
     char *buffer, int buflen);
```

DESCRIPTION

These privileged routines provide access to the protected password database in a manner similar to the way *getpwent*(3C) routines handle the regular password file, **/etc/passwd**.

These routines are particularly useful in situations where it is not necessary to get information from the regular password file. *getspwent*(3X) can be used on a trusted system to return the password, audit ID, and audit flag information. Programs using these routines must be linked with the security library, **libsec**.

Note that **getspwent()** routines are no longer supported. They are temporarily available for backward compatibility. New applications accessing the protected password database on trusted systems should use the **getprpwent()** routines. See *getprpwent*(3).

getspwent(), **getspwuid()**, **getspwaid()**, and **getspwnam()** each returns a pointer to an object of *s_passwd* structure. The *s_passwd* structure is maintained for compatibility with existing software and consists of five fields as follows:

```
struct s_passwd {
    char  *pw_name;     /* login name */
    char  *pw_passwd;   /* encrypted password */
    char  *pw_age;      /* password age */
    int   pw_audid;     /* audit ID   */
    int   pw_audflg;    /* audit flag 1=on, 0=off */
};
```

Since the *s_passwd* structure is declared in the <**pwd.h**> header file, it is unnecessary to redeclare it.

To access other fields in the protected password database that are not included in the *s_passwd* structure, use **getprpwent()**. See *getprpwent*(3) for more information.

getspwent() When first called, **getspwent()** returns a pointer to each *s_passwd* structure obtained from the protected password database for each user in sequence. Subsequent calls can be used to search the entire database.

getspwuid() Searches for an entry that matches the specified *uid*. It then returns a pointer to the particular structure in which *uid* is found.

getspwaid() Similarly searches for a numerical audit ID matching *aid* and returns a pointer to the particular structure in which *aid* is found (see *passwd*(4) for details on this field).

getspwnam() Searches for an entry that matches the specified *name*. Returns a pointer to the particular structure in which *name* is found.

setspwent() Resets the protected password database pointer to the beginning of the file to allow repeated searches.

endspwent() Should be called to close the protected password database file when processing is complete.

fgetspwent() Is no longer supported. It is provided for those applications that did not use **/.secure/etc/passwd**.

Reentrant Interfaces

getspwuid_r(), **getspwaid_r()**, **getspwnam_r()**, and **fgetspwent_r()** expect to be passed three extra parameters:

1. The address of an *s_passwd* structure where the result will be stored;
2. A buffer to store character strings (such as the password) to which fields in the *s_passwd* structure will point;
3. The length of the user-supplied buffer. A buffer length of 1024 is recommended.

In addition to the above three parameters, **getspwent_r()** requires a pointer to a (**FILE ***) variable. **setspwent_r()** and **endspwent_r()** are to be used only in conjunction with **getspwent_r()** and take the same pointer to a (**FILE ***) variable as a parameter. **setspwent_r()** can be used to rewind or open the protected password database. **endspwent_r()** should be called when done to close the file.

Note that the (**FILE ***) variable must be initialized to NULL before it is passed to **getspwent_r()** or **setspwent_r()** for the first time. Thereafter it should not be modified in any way.

APPLICATION USAGE

In a multithreaded application, these routines are safe to be called only from one dedicated thread. These routines are not POSIX.1c async-cancel safe nor async-signal safe.

RETURN VALUE

getspwent() returns a NULL pointer if any of its routines encounters an end-of-file or error while searching, or if the effective user ID of the calling process is not zero.

getspwent_r() returns a –1 if any of its routines encounters an end-of-file or error, or if the supplied buffer has insufficient length. If the operation is successful, 0 is returned.

WARNINGS

The above routines use <**stdio.h**>, which causes them to increase the size of programs by more than might otherwise be expected.

Since all information for **getspwent()**, **getspwuid()**, **getspwaid()**, **getspwnam()**, **setspwent()**, **endspwent()**, and **fgetspwent()** is contained in a static area, it must be copied to be saved.

Network Information Service is not supported on trusted systems.

The routines described in this manpage are no longer supported. They are temporarily available for backward compatibility and will be obsoleted in a future release.

EXAMPLE

The following code excerpt counts the number of entries in the protected password database:

```
int count = 0;
struct s_passwd *pwbuf;

setspwent();
while (pwbuf=getspwent())
    count++;
endspwent();
```

AUTHOR

getspwent() was developed by HP.

FILES

/tcb/files/auth/*/* Protected Password database

SEE ALSO

ypcat(1), getgrent(3C), getlogin(3C), getpwent(3C), getprpwent(3), putspwent(3X), passwd(4).

NAME

getstr, mvgetstr, mvwgetstr, wgetstr — get a multi-byte character string from the terminal

SYNOPSIS

```
#include <curses.h>
int getstr(char *str);
int mvgetstr(int y, int x, char *str);
int mvwgetstr(WINDOW *win, int y, int x, char *str);
int wgetstr(WINDOW *win, char *str);
```

DESCRIPTION

The effect of **getstr()** is as though a series of calls to **getch()** were made, until a newline or carriage return is received. The resulting value is placed in the area pointed to by *str*. The user's erase and kill characters are interpreted, as well as any special keys (such as function keys, home key, clear key, and so on).

The **mvgetstr()** function is identical to **getstr()** except that it is as though it is a call to **move()** and then a series of calls to **getch()**. The **mvwgetstr()** function is identical to **getstr()** except it is as though a call to **wmove()** is made and then a series of calls to **wgetch()**.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

Reading a line that overflows the array pointed to by *str* with **getstr()**, **mvgetstr()**, **mvwgetstr()** or **wgetstr()** causes undefined results.

Traditional implementations often limited the number of bytes returned to 256.

SEE ALSO

Input Processing in curses_intro(3X), beep(3X), getch(3X), getnstr(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 2.

Issue 3

In X/Open Curses, Issue 3, the **getstr()**, **mvgetstr()**, **mvwgetstr()** and **wgetstr()** functions were described in the **addstr()** entry. In X/Open Curses, Issue 4, the DESCRIPTION of these functions is rewritten for clarity and is updated to indicate that they will handle multi-byte sequences correctly.

NAME

getsubopt() - parse suboptions from a string.

SYNOPSIS

```
#include <stdlib.h>
int getsubopt(char **optionp, char * const *tokens, char **valuep);
```

DESCRIPTION

getsubopt() parses suboptions in a flag argument that were initially parsed by **getopt()** (see *getopt*(3C)). These suboptions are separated by commas, and may consist of either a single token, or a token-value pair separated by an equal sign. Because commas delimit suboptions in the option string, they are not allowed to be part of the suboption or the value of a suboption. Similarly, because the equal sign separates a token from its value, a token must not contain an equals sign. An example command that uses this syntax is **mount**. **mount** allows parameters to be specified with the **-** switch as follows:

```
mount -o rw,hard,bg,wsize=1024 speed:/usr /usr
```

In this example there are four suboptions: **rw**, **hard**, **bg**, and **wsize**, the last of which has an associated value of 1024.

getsubopt() takes the address of a pointer to the option string, a vector of possible tokens, and the address of a value string pointer. It returns the index of the token that matched the suboption in the input string or –1 if there was no match. If the option string at ***optionp* contains only one suboption, **getsubopt()** updates ***optionp* to point to the null at the end of the string, otherwise it isolates the suboption by replacing the comma separator with a null, and updates ***optionp* to point to the start of the next suboption. If the suboption has an associated value, **getsubopt()** updates ***valuep* to point to the value of the first character. Otherwise it sets ***valuep* to NULL.

The token vector is organized as a series of pointers to NULL-terminated strings. The end of the token vector is identified by NULL.

When **getsubopt()** returns, if ***valuep* is not NULL then the suboption processed included a value. The calling program can use this information to determine if the presence or lack of a value for this suboption is an error.

Additionally, when **getsubopt()** fails to match the suboption with the tokens in the *tokens* array, the calling program should decide if this is an error, or if the unrecognized option should be passed on to another program.

Application Usage

getsubopt() is thread-safe and async-cancel-safe.

EXTERNAL INFLUENCES

Locale

The **LC_CTYPE** category determines the interpretation of option letters as single and/or multi-byte characters.

International Code Set Support

Single- and multi-byte character code sets are supported with the exception of multi-byte-character file names.

EXAMPLES

The following code fragment shows how options can be processed to the **mount** command by using **getsubopt()**.

```
char *myopts[] = {
#define READONLY        0
                        "ro",
#define READWRITE       1
                        "rw",
#define WRITESIZE       2
                        "wsize",
#define READSIZE        3
                        "rsize",
                        NULL};
```

```
main (int argc, char **argv)
{
    int sc, c, errflag;
    char *options, *value;
    extern char *optarg;
    extern int optind;
    .
    .
    .
    while ((c = getopt(argc, argv, "abf:o:")) != EOF)
        switch (c) {
        case 'a':  /* process 'a' option */
            break;
        case 'b':  /* process 'b' option */
            break;
        case 'f':
            ofile = optarg;
            break;
        case '?':
            errflag++;
            break;
        case 'o':
            options = optarg;
            while (*options != '\0') {
                switch(getsubopt(&options, myopts, &value)) {
                case READONLY: /* process ro option */
                    break;
                case READWRITE: /* process rw option */
                    break;
                case WRITESIZE: /* process wsize option */
                    if (value == NULL) {
                        error_no_arg();
                        errflag++;
                    }
                    else
                        write_size = atoi(value);
                    break;
                case READSIZE: /* process rsize option */
                    if (value == NULL) {
                        error_no_arg();
                        errflag++;
                    }
                    else
                        read_size = atoi(value);
                    break;
                default:
                    /* process unknown token */
                    error_bad_token(value);
                    errflag++;
                    break;
                }
            }
            break;
        }
    }
    if (errflg) {
        fprintf(stderr, "usage: . . . ");
        exit (2);
    }
    for ( ; optind < argc; optind++) {
        /* process remaining arguments */
         .
```

```
                .
                .
        }
```

SEE ALSO
getopt(3C).

STANDARDS CONFORMANCE
`getsubopt()`: SVID3

NAME

gettimer - get value of a per-process timer

SYNOPSIS

```
#include <sys/timers.h>
int gettimer(timer_t timerid, struct itimerspec *value);
```

DESCRIPTION

The **gettimer()** function returns an **itimerspec** structure value to the **value** argument. The **it_value** member of the structure represents the amount of time in the current interval before the timer expires for the timer specified in **timerid**, or zero if the timer is disabled. The **it_interval** member has the value last set by **reltimer()** (see *reltimer*(3C)). The members of **value** are subject to the resolution of the timer (see *mktimer*(3C)).

The behavior of this function is undefined if **value** is NULL.

APPLICATION USAGE

gettimer() is thread-safe. It is not async-cancel-safe.

RETURN VALUE

Upon successful completion, **gettimer()** returns zero; otherwise, it returns –1 and sets **errno** to indicate the error.

ERRORS

gettimer() fails if any of the following conditions are encountered:

[EINVAL] **timerid** does not correspond to an ID returned by **mktimer()**.

[EIO] An error occurred while accessing the clock device.

FILES

/usr/include/sys/timers.h

SEE ALSO

timer_gettime(2), mktimer(3C), reltimer(3C).

STANDARDS CONFORMANCE

gettimer(): AES

NAME

gettxt() - read text string from message file

SYNOPSIS

```
#include <unistd.h>
char *gettxt(char *msg_id, char *def_str);
```

DESCRIPTION

The **gettxt()** routine retrieves a text string from a message file for the current locale.

msg_id has the following syntax:

msgfilename:*msgnumber*

where *msgfilename* is the name of the message file generated by *mkmsgs*(1). If *msgfilename* is **NULL**, **gettxt()** uses the message file specified in the last call to *setcat*(3C). *msgnumber* is the sequence number of the text string in the message file (the sequence begins at 1).

gettxt() returns the message

```
Message not found!!
```

under any of the following conditions:

- *msgfilename* is an invalid message catalog name.
- No catalog is specified in *msg_id* or through *setcat*(3C).
- *msgnumber* is not a positive number.
- No message could be retrieved and *def_str* is **NULL**.

APPLICATION USAGE

gettxt() is thread-safe. It is not async-cancel-safe. A cancellation point may occur when a thread is executing **gettxt()**.

EXTERNAL INFLUENCES

Environment Variables

gettxt() uses the environment variable **LC_MESSAGES** to determine the locale to use to search for the *msgfilename* message file. If **LC_MESSAGES** is not set, the environment variable **LANG** is used. If **LANG** is not set, the "C" locale will be used. The user can also change the locale via the *setlocale*(3C) routine.

If the *msgfilename* message file is not found in the specified locale or if the *msgnumber* is out of bounds, **gettxt()** attempts to retrieve the text string from the "C" locale. *def_str* is the string returned if a text string cannot be retrieved even from the "C" locale.

EXAMPLES

The following code fragments are equivalent:

```
gettxt("mytest:1", "my default message");
setcat("mytest");
gettxt(":1", "my default message");
```

SEE ALSO

mkmsgs(1), setcat(3C), setlocale(3C), environ(5).

STANDARDS COMPLIANCE

gettxt(): SVID3

NAME

getusershell(), setusershell(), endusershell() - get legal user shells

SYNOPSIS

```
#include <unistd.h>
char *getusershell(void);
void setusershell(void);
void endusershell(void);
```

Obsolescent Interfaces

```
char *getusershell_r(char **shell_datap);
void setusershell_r(char **shell_datap);
void endusershell_r(char **shell_datap);
```

DESCRIPTION

getusershell() Returns a pointer to the first legal user shell as defined in the file **/etc/shells** (see *shells*(4)). If **/etc/shells** does not exist or is not readable, **getusershell()** returns the following standard system shells:

```
/sbin/sh
/usr/bin/sh
/usr/bin/rsh
/usr/bin/ksh
/usr/bin/rksh
/usr/bin/csh
/usr/bin/keysh
```

as if they were contained in **/etc/shells**. The file is left open so that the next call returns the next shell. A null pointer (0) is returned on EOF or error.

setusershell() Rewinds the file.

endusershell() Closes the file.

Obsolescent Interfaces

getusershell_r(), **setusershell_r()**, **endusershell_r()** get legal user shells.

APPLICATION USAGE

getusershell(), **setusershell()** and **endusershell()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing **getusershell()** or **setusershell()**.

WARNINGS

getusershell_r(), **setusershell_r()** and **endusershell_r()** are obsolescent interfaces supported only for compatibility with existing DCE applications. New multithreaded applications should use **getusershell()**, **setusershell()** and **endusershell()**.

AUTHOR

getusershell() was developed by HP and the University of California, Berkeley.

FILES

/etc/shells

SEE ALSO

shells(4).

NAME

getutent(), getutid(), getutline(), pututline(), _pututline(), setutent(), endutent(), utmpname() - access utmp file entry

SYNOPSIS

```
#include <utmp.h>
struct utmp *getutent(void);
struct utmp *getutid(const struct utmp *id);
struct utmp *getutline(const struct utmp *line);
struct utmp *_pututline(const struct utmp *utmp);
void pututline(const struct utmp *utmp);
void setutent(void);
void endutent(void);
int utmpname(const char *file);
```

Obsolescent Interfaces

```
int getutent_r(struct utmp **utmp, struct utmp_data *ud);
int getutid_r(
     struct utmp *id,
     struct utmp **utmp,
     struct utmp_data *ud);
int getutline_r(
     struct utmp *line,
     struct utmp **utmp,
     struct utmp_data *ud);
int pututline_r(const struct utmp *utmp, struct utmp_data *ud);
void setutent_r(struct utmp_data *ud);
void endutent_r(struct utmp_data *ud);
int utmpname_r(const 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) */
    pid_t 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. */
    unsigned short ut_reserved1;  /* Reserved for future use */
    time_t ut_time;           /* time entry was made */
    char ut_host[16];         /* host name, if remote;NOTSUPPORTED*/
    unsigned long ut_addr;    /* Internet addr of host, if remote */
};
```

getutent() Reads in the next entry from a **utmp**-like file. If the file is not already open, **getutent()** opens it. If it reaches the end of the file, **getutent()** 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, **getutid()** returns a pointer to the first entry whose type is one of these four, and whose **ut_id** field matches *id->ut_id*. If end-of-file is reached without a match, **getutid()** fails.

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

pututline() Writes out the supplied **utmp** structure into the *utmp* file, translates the supplied *utmp* structure int a *utmpx* structure and writes it to a *utmpx* file. **pututline()** uses **getutid()** to search forward for the proper location if it is not already there. It is normally expected that the application program has already searched for the proper entry by using one of the **getut()** routines before calling **pututline()**. If the search as already been made, **pututline()** does not repeat it. If **pututline()** does not find a matching slot for the new entry, it adds a new entry to the end of the file.

_pututline() Performs the same actions as **pututline()**, except that it returns a value useful for error checking.

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 being examined from **/etc/utmp** and **/etc/utmpx** to any other file(s). In this case, the name provided to **utmpname** will be used for the *utmp* functions. An "x" will be appended to this name, and will be used by the *getutx* functions. The one exception to this are the *putut(x)* functions as they access both files in an attempt to keep the *utmp* and *utmpx* files in sync. The other file(s) are usually **/var/adm/wtmp** and **/var/adm/wtmpx**. If the file(s) do not exist, its absence is not discovered until the first subsequent attempt to reference the file. **utmpname()** does not open the file — it merely closes the old file if it is currently open, and saves the new file name.

The most current entry is saved in a static structure. Multiple accesses require that the structure be copied before further accesses are made. During each call to either **getutid()** or **getutline()**, the static structure is examined before performing more I/O. If the contents of the static structure match what the routine is searching for, no additional searching is done. Therefore, if using **getutline()** to search for multiple occurrences, it is necessary to zero out the static structure after each success; otherwise **getutline()** simply returns the same pointer over and over again. There is one exception to the rule about removing the structure before a new read: The implicit read done by **pututline()** (if it finds that it is not already at the correct place in the file) does not alter the contents of the static structure returned by **getutent()**, **getutid()**, or **getutline()** if the user has just modified those contents and passed the pointer back to **pututline()**.

Obsolescent Interfaces

getutent_r(), **getutid_r()**, **getutline_r()**, **pututline_r()**, **setutent_r()**, **endutent_r()**, **utmpname_r()** access utmp file entry.

APPLICATION USAGE

getutent(), **getutid()**, **getutline()**, **pututline()**, **_pututline()**, **setutent()**, **endutent()** and **utmpname()** are not thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing **getutent()**, **getutid()**, **getutline()**, **pututline()**, **_pututline()**, **setutent()**, **endutent()** and **utmpname()**.

RETURN VALUE

These functions return a NULL pointer upon failure to read (whether for permissions or having reached end-of-file), or upon failure to write. They also return a NULL pointer if the size of the file is not an integral multiple of **sizeof(struct utmp)**.

_pututline() behaves the same as **pututline()**, except that it returns a pointer to a static location containing the most current utmp entry if the **_pututline()** call succeeds. The contents of this structure is identical to the contents of the supplied **utmp** structure if successful. If **_pututline()** fails upon writing to **utmp** , it returns a NULL pointer. If **_putuline()** is successful in writing to the

utmp file and fails in writing to the **utmpx** file, then _pututline will behave as if it succeeded. Please note that the **utmp** file and the **utmpx** file may not be in sync due to the above behavior. **pututline()** and **_pututline()** are only guaranteed to have written to the **utmp** file upon successful completion.

Reentrant Interfaces

Upon successful completion, **getutent_r()**, **getutid_r()**, **getutline_r()** and **pututline_r()** return 0. Otherwise, they all return -1 and set **errno**.

ERRORS

Reentrant Interfaces

[EINVAL] *utmp* or *ud* parameter is equal to NULL .

WARNINGS

getutent_r(), **getutid_r()**, **getutline_r()**, **pututline_r()**, **setutent_r()**, **endutent_r()**, and **utmpname_r()** are obsolescent interfaces supported only for compatibility with existing DCE applications. New multithreaded applications should use use the getutx functions that provide equivalent functionality.

Some vendors' versions of **getutent()** erase the *utmp* file if the file exists but is not an integral multiple of **sizeof(struct utmp)**. Given the possibility of user error in providing a name to *utmpname* (such as giving improper arguments to *who*(1)), HP-UX does not do this, but instead returns an error indication.

For portability, **getutx** functions are preferred over these functions.

FILES

/etc/utmp
/etc/utmpx
/var/adm/wtmp

SEE ALSO

ttyslot(3C), utmp(4), pututxline(3).

STANDARDS CONFORMANCE

endutent(): SVID2, SVID3, XPG2

getutent(): SVID2, SVID3, XPG2

getutid(): SVID2, SVID3, XPG2

getutline(): SVID2, SVID3, XPG2

pututline(): SVID2, SVID3, XPG2

setutent(): SVID2, SVID3, XPG2

utmpname(): SVID2, SVID3, XPG2

NAME

getutxent(), getutxid(), getutxline(), pututxline(), setutxent(), endutxent() - access utmpx file entry

SYNOPSIS

```
#include <utmpx.h>
struct utmpx *getutxent(void);
struct utmpx *getutxid(const struct utmpx *id);
struct utmpx *getutxline(const struct utmpx *line);
struct utmpx *pututxline(const struct utmpx *utmpx);
void setutxent(void);
void endutxent(void);
```

DESCRIPTION

getutxent(), **getutxid()**, and **getutxline()** each return a pointer to a structure of the following type:

```
struct utmpx {
    char ut_user[24];          /* User login name */
    char ut_id[4];             /* /etc/inittab id (usually line #) */
    char ut_line[12];          /* device name (console, lnxx) */
    pid_t 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. */
    unsigned short ut_reserved1;  /* Reserved for future use */
    struct timeval {
        time_t  tv_sec;        /* seconds */
        long    tv_usec;       /* and microseconds */
        } ut_tv;               /* time entry was made */
    char ut_host[64];          /* host name, if remote; NOT SUPPORTED */
    unsigned long ut_addr;     /* Internet addr of host, if remote */
    char ut_reserved2[12] ;    /* Reserved for future use */
};
```

getutxent() Reads in the next entry from a **utmpx**-like file. If the file is not already open, **getutxent()** opens it. If it reaches the end of the file, **getutxent()** fails.

getutxid() Searches forward from the current point in the *utmpx* 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, **getutxid()** returns a pointer to the first entry whose type is one of these four, and whose **ut_id** field matches *id->ut_id*. If end-of-file is reached without a match, **getutxid()** fails.

getutxline() Searches forward from the current point in the *utmpx* file until it finds an entry of type **LOGIN_PROCESS** or **USER_PROCESS** that also has a **ut_line** string matching the *line->ut_line* string. If end-of-file is reached without a match, **getutxline()** fails.

pututxline() Writes out the supplied **utmpx** structure into the *utmpx* file, translates the supplied **utmpx** structure into a **utmp** structure and writes it to a *utmp* file. **pututxline()** uses **getutxid()** to search forward for the proper location if it is not already there. It is normally expected that the application program has already searched for the proper entry by using one of the **getutx()** routines before calling **pututxline()**. If the search has already been made, **pututxline()** does not repeat it. If **pututxline()** does not find a matching slot for the new entry, it adds a new entry to the end of the file.

setutxent() 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.

endutxent() Closes the currently open file.

The most current entry is saved in a static structure. Multiple accesses require that the structure be copied before further accesses are made. During each call to either **getutxid()** or **getutxline()**, the static structure is examined before performing more I/O. If the contents of the static structure match what the routine is searching for, no additional searching is done. Therefore, if using **getutxline()** to search for multiple occurrences, it is necessary to zero out the static structure after each success; otherwise **getutxline()** simply returns the same pointer over and over again. There is one exception to the rule about removing the structure before a new read: The implicit read done by **pututxline()** (if it finds that it is not already at the correct place in the file) does not alter the contents of the static structure returned by **getutxent()**, **getutxid()**, or **getutxline()** if the user has just modified those contents and passed the pointer back to **pututxline()**.

RETURN VALUE

These functions return a NULL pointer upon failure to read (whether for permissions or having reached end-of-file), or upon failure to write. They also return a NULL pointer if the size of the file is not an integral multiple of **sizeof(struct utmpx)**.

If **pututxline()** is successful, it returns a pointer to a static location containing the most current **utmpx** entry. The contents of this structure are identical to the contents of the supplied **utmpx** structure if successful. If **pututline()** fails upon writing to **utmpx**, it returns a NULL pointer. If **pututline()** is successful in writing to the **utmpx** file but fails in writing to the **utmp** file, then **pututline()** will behave as if it succeeded. Note that the **utmpx** file and the **utmp** file may not be in sync due to the above behavior. **pututxline()** is only guaranteed to have written to the **utmpx** file upon successful completion.

APPLICATION USAGE

getutxent(), **getutxid()**, **getutxline()**, **pututxline()**, **setutxent()** and **endutxent()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing these interfaces.

FILES

/etc/utmp
/etc/utmpx
/var/adm/wtmp

SEE ALSO

ttyslot(3C), utmp(4), getut(3C), getutent(3C).

STANDARDS CONFORMANCE

endutxent(): XPG4.2

getutxent(): XPG4.2

getutxid(): XPG4.2

getutxline(): XPG4.2

pututxline(): XPG4.2

setutxent(): XPG4.2

NAME

getwc(), getwchar(), fgetwc() - get a wide character from a stream file

SYNOPSIS

```
#include <wchar.h>
wint_t getwc(FILE *stream);
wint_t getwchar(void);
wint_t fgetwc(FILE *stream);
```

Obsolescent Interfaces

```
wint_t getwc_unlocked(FILE *stream);
wint_t getwchar_unlocked(void);
wint_t fgetwc_unlocked(FILE *stream);
```

Remarks:

These functions are compliant with the XPG4 Worldwide Portability Interface wide-character I/O functions. They parallel the 8 bit character I/O functions defined in *getc*(3S).

DESCRIPTION

getwc() Returns the next character from the named input *stream*, converts that to the corresponding wide character and moves the file pointer ahead one character in *stream*. **getwchar()** is defined as **getwc(stdin)**. **getwc()** and **getwchar()** are defined both as macros and as functions.

fgetwc() Behaves like **getwc()**, but is a function rather than a macro.

Definitions for these functions, the types **wint_t**, **wchar_t** and the value **WEOF** are provided in header file <**wchar.h**>.

Obsolescent Interfaces

getwc_unlocked(), **getwchar_unlocked()**, **fgetwc_unlocked()** get a wide character from a stream file.

APPLICATION USAGE

getwc(), **getwchar()** and **fgetwc()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing these interfaces.

RETURN VALUE

Upon successful completion, **getwc()**, **getwc_unlocked()**, **getwchar()**, **getwchar_unlocked()**, **fgetwc()**, and **fgetwc_unlocked()** return the next wide character read from *stream* (**stdin** for **getwchar()**) converted to a type **wint_t**. If the stream is at end-of-file, the end-of-file indicator for the stream is set and **WEOF** is returned. If a read error occurs, the error indicator for the stream is set, **errno** is set to indicate the error, and **WEOF** is returned.

ferror() and **feof()** can be used to distinguish between an error condition and an end-of-file condition.

ERRORS

getwc(), **getwc_unlocked()**, **getwchar()**, **getwchar_unlocked()**, **fgetwc()**, and **fgetwc_unlocked()** fail if data needs to be read into the *stream*'s buffer, and:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying *stream* and the process would be delayed in the read operation.

[EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for reading.

[EINTR] The read operation was terminated due to the receipt of a signal, and either no data was transferred or the implementation does not report partial transfer for this file.

[EIO] A physical I/O error has occurred, or the process is a member of a background process and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the **SIGTTIN** signal or the process group of the process is orphaned.

[EILSEQ] The data obtained from the input stream does not form a valid wide character.

Additional **errno** values may be set by the underlying **read()** function (see *read*(2)).

EXTERNAL INFLUENCES

Locale

The **LC_CTYPE** category determines how wide character conversions are done.

International Code Set Support

Single- and multi-byte character code sets are supported.

WARNINGS

If the value returned by **getwc()**, **getwchar()**, **fgetwc()**, or **fgetwc_unlocked()** is stored into a type **wchar_t** variable then compared against the constant **WEOF**, the comparison may never succeed because extension of a **wchar_t** to a **wint_t** is machine-dependent.

getwc_unlocked(), **getwchar_unlocked()** and **fgetwc_unlocked()** are obsolescent interfaces supported only for compatibility with existing DCE applications. New multithreaded applications should use **getwc()**, **getwchar()** and **fgetwc()**.

AUTHOR

getwc() was developed by OSF and HP.

SEE ALSO

fclose(3S), ferror(3S), flockfile(3S), fopen(3S), fread(3S), fgetws(3C), putwc(3C), read(2), scanf(3S).

STANDARDS CONFORMANCE

getwc(): XPG4

fgetwc(): XPG4

getwchar(): XPG4

NAME

getwd() - get pathname of current working directory

SYNOPSIS

```
#include <unistd.h>
char *getwd(char *buf);
```

DESCRIPTION

getwd() places the absolute pathname of the current working directory in the array pointed to by *buf*, and returns *buf*.

If the length of the pathname of the current working directory is greater than PATH_MAX+1 bytes, **getwd()** fails and returns a null pointer.

RETURN VALUE

Upon successful completion, **getwd()** returns a pointer to the current directory pathname. Otherwise, it returns NULL with **errno** set.

ERRORS

getwd() fails if the following condition is encountered:

[ENAMETOOLONG] The length of the specified path name exceeds **PATH_MAX+1** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.

getwd() may fail if any of the following conditions are encountered:

[EACCES] Read or search permission is denied for a component of pathname.

[EFAULT] *buf* points outside the allocated address space of the process. **getwd()** may not always detect this error.

EXAMPLES

```
#include <stdio.h>
#include <unistd.h>
char *cwd;
char buf[PATH_MAX+1];
        ...
if ((cwd = getwd(buf)) == NULL) {
    perror("getwd");
    exit(1);
}
puts(cwd);
```

APPLICATION USAGE

getwd() is thread-safe. It is not async-cancel-safe. A cancellation point may occur when a thread is executing **getwd()**.

WARNINGS

For portability, **getcwd()** is preferred over this function.

AUTHOR

getwd() was developed by HP and the University of California, Berkeley.

SEE ALSO

getcwd(3C).

STANDARDS CONFORMANCE

getwd(): XPG4.2

NAME
getwin, putwin — dump window to, and reload window from, a file

SYNOPSIS
```
#include <curses.h>
WINDOW *getwin(FILE *filep);
int putwin(WINDOW *win, FILE *filep);
```

DESCRIPTION
The **getwin()** function reads window-related data stored in the file by **putwin()**. The function then creates and initialises a new window using that data.

The **putwin()** function writes all data associated with *win* into the *stdio* stream to which *filep* points, using an unspecified format. This information can be retrieved later using **getwin()**.

RETURN VALUE
Upon successful completion, **getwin()** returns a pointer to the window it created. Otherwise, it returns a null pointer.

Upon successful completion, **putwin()** returns OK. Otherwise, it returns ERR.

ERRORS
No errors are defined.

SEE ALSO
scr_dump(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 4.

NAME
getyx — get cursor and window coordinates

SYNOPSIS
```
#include <curses.h>
void getyx(WINDOW *win, int y, int x);
```

DESCRIPTION
The **getyx()** macro stores the cursor position of the specified window in *y* and *x*.

RETURN VALUE
No return values are defined.

ERRORS
No errors are defined.

APPLICATION USAGE
These interfaces are macros and '**&**' cannot be used before the *y* and *x* arguments.

SEE ALSO
getbegyx(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The entry is rewritten for clarity.

NAME

glob(), globfree() - file name generation function

SYNOPSIS

```
#include <glob.h>

int glob(
     const char *pattern,
     int flags,
     int (*errfunc)(const char *, int),
     glob_t *pglob
);

void globfree(glob_t *pglob);
```

DESCRIPTION

glob() is a path name generator. *pattern* is a pointer to a path name pattern to be expanded. If *pattern* contains any of the special characters *****, **?**, or **[**, *pattern* is matched against all accessible path names. In order to have access to a path name, **glob()** requires:

- Search permission on every component of a path except the last.
- Read permission on each directory of any file name component of *pattern* that contains any of the above special characters.

glob() stores the number of matched path names in *pglob –> gl_pathc* and a pointer to a sorted list of path names in *pglob–>gl_pathv*. The first pointer after the last path name is a NULL pointer.

It is the caller's responsibility to allocate space for the structure pointed to by *pglob*. **glob()** allocates other space as needed, including the memory pointed to by *gl_pathv*. **globfree()** frees any space associated with *pglob* from a previous call to **glob()**.

The *flags* argument is used to control the behavior of **glob()**. The value of *flags* is the bit-wise inclusive OR of the following constants defined in **<glob.h>**:

GLOB_NOESCAPE Disable backslash escaping.

GLOB_ERR Causes **glob()** to return when it first encounters a directory that it cannot open or read. Ordinarily, **glob()** continues to find matches.

GLOB_MARK Each path name that matches *pattern* and is a directory, has a **/** appended.

GLOB_NOSORT Ordinarily, **glob()** sorts the matching path names according to the currently active *collation sequence* as defined by the **LC_COLLATE** category. When this flag is used, the order of path names returned is unspecified.

GLOB_NOCHECK If *pattern* does not match any path name, **glob()** returns a list consisting of only *pattern*, and the number of matched path names is 1.

GLOB_DOOFFS Make use of *pglob –> gl_offs*. If this flag is set, *pglob –> gl_offs* is used to specify how many NULL pointers to add to the beginning of *pglob –> gl_pathv*. In other words, *pglob –> gl_pathv* points to *pglob –> gl_offs* NULL pointers, followed by *pglob –> gl_pathc* path name pointers, followed by a NULL pointer.

GLOB_APPEND Append path names generated to the ones from a previous call to **glob()**.

The **GLOB_APPEND** flag can be used to append a new set of path names to those found in a previous call to **glob()**. The following rules apply when two or more calls to **glob()** are made with the same value of *pglob* and without intervening calls to **globfree()**:

- The first call must not set **GLOB_APPEND**. All subsequent calls must set it.
- All of the calls must set **GLOB_DOOFFS**, or all must not set it.
- After the second call, *pglob –> gl_pathv* points to a list containing the following:
  - Zero or more NULL pointers, as specified by **GOB_DOOFFS** and *pglob –> gl_offs*.
  - Pointers to the path names that were in the *pglob –> gl_pathv* list before the call, in the same order as before.
  - Pointers to the new path names generated by the second call, in the specified order.

- The count returned in *pglob–>gl_pathc* will be the total number of path names from the two calls.
- The application can change any of the fields after a call to **glob()**. If it does, it must reset them to the original value before a subsequent call, using the same *pglob* value, to **globfree()** or **glob()** with the **GLOB_APPEND** flag.

If, during the search, a directory is encountered that cannot be opened or read and *errfunc* is not NULL, **glob()** calls (**errfunc*)() with two arguments:

- A pointer to the path that failed.
- The value of **errno** from the failure.

If *errfunc* is called and returns nonzero, or if the **GLOB_ERR** flag is set in *flags*, **glob()** stops the scan and returns **GLOB_ABORTED** after setting **gl_pathc** and **gl_pathv** in *pglob* to reflect the paths already scanned. If **GLOB_ERR** is not set and either *errfunc* is NULL or (**errfunc*)() returns zero, the error is ignored.

Pattern Matching Notation

The form of the patterns is the Pattern Matching Notation as qualified for Filename Expansion (see *regexp*(5)) with the following exceptions:

- Tilde (~) expansion is not performed.
- Variable expansion is not performed.

APPLICATION USAGE

glob() and **globfree()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing **glob()** or **globfree()**.

EXTERNAL INFLUENCES

Locale

The **LC_COLLATE** category determines the collating sequence used in compiling and executing regular expressions, and also the order of the returned paths if **GLOB_NOSORT** is not selected.

The **LC_CTYPE** category determines the interpretation of text as single byte and/or multibyte characters, and determines which characters are matched by character class expressions in regular expressions.

International Code Set Support

Single byte and multibyte character code sets are supported.

RETURN VALUE

If **glob()** terminates due to an error, it returns one of the following constants (defined in **<glob.h>**); otherwise, it returns zero.

GLOB_NOSPACE	An attempt to allocate memory failed.
GLOB_ABORTED	The scan was stopped because **GLOB_ERR** was set or (**errfunc*)() returned nonzero.
GLOB_NOMATCH	The *pattern* does not match any existing path name, and **GLOB_NOCHECK** was not set in *flags*.

In any case, the argument *pglob–>gl_pathc* returns the number of matched path names and the argument *pglob–>gl_pathv* contains a pointer to a null-terminated list of matched and sorted path names.

However, if *pglob–>gl_pathc* is zero, the content of *pglob–>gl_pathv* is undefined.

If the *pattern* argument passed to **glob()** is badly constructed, **glob()** returns zero and sets **gl_pathc** to zero unless **GLOB_NOCHECK** was set, in which case *pattern* is returned and **gl_pathc** is set to 1.

WARNINGS

GLOB_APPEND must not be set in an initial call to **glob()**.

AUTHOR

glob() and **globfree()** were developed by OSF and HP.

SEE ALSO
sh(1), fnmatch(3C), regexp(5).

STANDARDS CONFORMANCE
glob(): XPG4, POSIX.2
globfree(): XPG4, POSIX.2

(Series 800 Only)

NAME

gpio_get_status - return status lines of GPIO card

SYNOPSIS

```
#include <dvio.h>
int gpio_get_status(int eid);
```

DESCRIPTION

gpio_get_status() reads the status register of the GPIO interface associated with the device file identified by *eid*. *eid* is an entity identifier of an open GPIO device file obtained from an **open()**, **dup()**, **fcntl()**, or **creat()** call (see *open*(2), *dup*(2), *fcntl*(2), or *creat*(2)). The current state of each status line on the interface card is mapped to the value returned, with **STS0** mapped to the least significant bit. Only *x* least-significant bits are used, where *x* is the number of status lines available on the hardware interface being used.

DEPENDENCIES

Series 800

For the HP 27114A, *x* is 2.

For the HP 27114B, *x* is 6.

For the HP 28651A, *x* is 5.

RETURN VALUE

gpio_get_status() returns the value of the status register of the GPIO interface associated with *eid*, and -1 if an error was encountered.

ERRORS

gpio_get_status() fails if any of the following conditions are encountered and sets **errno** accordingly:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to a GPIO device file.

(Series 800 Only)

NAME

gpio_set_ctl - set control lines on GPIO card

SYNOPSIS

```
#include <dvio.h>
int gpio_set_ctl(int eid, int value);
```

DESCRIPTION

gpio_set_ctl() sets the control register of a GPIO interface. *eid* is an entity identifier of an open GPIO device file obtained from an **open()**, **dup()**, **fcntl()**, or **creat()** call (see *open*(2), *dup*(2), *fcntl*(2), and *creat*(2)). *value* is the value to be written into the control register of the GPIO interface associated with *eid*.

value is mapped onto the control lines on the interface card, with the least significant bit mapped to **CTL0**. Only the *x* least significant bits are used, where *x* is the number of control lines available on the hardware interface being used.

DEPENDENCIES

Series 800

For the HP 27114A, *x* is 3.

For the HP 27114B, *x* is 6.

For the HP 28651A, *x* is 5.

RETURN VALUE

gpio_set_ctl() returns 0 if successful and -1 if an error was encountered.

ERRORS

gpio_set_ctl() fails if any of the following conditions are encountered, and sets **errno** accordingly:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to a GPIO device file.

NAME

grantpt - grant access to the STREAMS slave pty

SYNOPSIS

```
int grantpt (int fildes);
```

DESCRIPTION

The passed parameter, *fildes*, is a file descriptor that is returned from a successful open of a STREAMS master pty (pseudo-terminal) device. The **grantpt()** function modifies the ownership and mode of the slave pty device special file associated with its master pty counterpart.

A **setuid()** root program is spawned to change ownership and mode of the pty slave device file in the following way: The group ID is set to a reserved group named "tty". The slave user ID is set to the effective owner of the calling process. The permissions of the slave device are set so that the owner is allowed read and write access and the group is allowed write access.

RETURN VALUE

Upon successful completion, the **grantpt()** function returns a value of 0 (zero). Otherwise, it returns a value of -1.

Failure may result under the following conditions:

- The file descriptor specified by the *fildes* parameter is not an open file descriptor.
- The file descriptor specified by the *fildes* parameter is not associated with a STREAMS master pty device.
- The corresponding slave pty device cannot be accessed.

WARNINGS

The **grantpt()** function may also fail if the application has installed a signal handler to catch the SIGCHLD (death of a child) signal.

EXAMPLES

The following example shows how **grantpt()** is typically used.

```
int fd_master, fd_slave;
char *slave;
   ...
      fd_master = open("/dev/ptmx", O_RDWR);
      grantpt(fd_master);
      unlockpt(fd_master);
      slave = ptsname(fd_master);
      fd_slave = open(slave, O_RDWR);
      ioctl(fd_slave, I_PUSH, "ptem");
      ioctl(fd_slave, I_PUSH, "ldterm");
```

AUTHOR

grantpt() was developed by HP and OSF.

SEE ALSO

open(2), unlockpt(3C), ptsname(3C), ptm(7), pts(7), ptem(7), ldterm(7).

(ENHANCED CURSES)

NAME

halfdelay — control input character delay mode

SYNOPSIS

```
#include <curses.h>
int halfdelay(int tenths);
```

DESCRIPTION

The **halfdelay()** function sets the input mode for the current window to Half-Delay Mode and specifies *tenths* tenths of seconds as the half-delay interval. The *tenths* argument must be in a range from 1 up to and including 255.

RETURN VALUE

Upon successful completion, **halfdelay()** returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

The application can call **nocbreak()** to leave Half-Delay mode.

SEE ALSO

Input Mode in curses_intro(3X), cbreak(3X), <curses.h>, *X/Open System Interface Definitions, Issue 4, Version 2* specification, Chapter 9, *General Terminal Interface*.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME
has_ic, has_il — query functions for terminal insert and delete capability

SYNOPSIS
```
#include <curses.h>
bool has_ic(void);
bool has_il(void);
```

DESCRIPTION
The **has_ic()** function indicates whether the terminal has insert- and delete-character capabilities.

The **has_il()** function indicates whether the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions.

RETURN VALUE
The **has_ic()** function returns TRUE if the terminal has insert- and delete-character capabilities. Otherwise, it returns FALSE.

The **has_il()** function returns TRUE if the terminal has insert- and delete-line capabilities. Otherwise, it returns FALSE.

ERRORS
No errors are defined.

APPLICATION USAGE
The **has_il()** function may be used to determine if it would be appropriate to turn on physical scrolling using **scrollok()**.

SEE ALSO
<curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The **has_il()** function is merged with this entry. In previous issues, it appeared in an entry of its own.

The entry is rewritten for clarity. The argument list for the **has_ic()** and **has_il()** functions is explicitly declared as **void**.

(ENHANCED CURSES)

NAME

hline, mvhline, mvvline, mvwhline, mvwvline, vline, whline, wvline — draw lines from single-byte characters and renditions

SYNOPSIS

```
#include <curses.h>
int hline(chtype ch, int n);
int mvhline(int y, int x, chtype ch, int n);
int mvvline(int y, int x, chtype ch, int n);
int mvwhline(WINDOW *win, int y, int x, chtype ch, int n);
int mvwvline(WINDOW *win, int y, int x, chtype ch, int n);
int vline(chtype ch, int n);
int whline(WINDOW *win, chtype ch, int n);
int wvline(WINDOW *win, chtype ch, int n);
```

DESCRIPTION

These functions draw a line in the current or specified window starting at the current or specified position, using *ch*. The line is at most *n* positions long, or as many as fit into the window.

These functions do not advance the cursor position. These functions do not perform special character processing. These functions do not perform wrapping.

The **hline()**, **mvhline()**, **mvwhline()** and **whline()** functions draw a line proceeding toward the last column of the same line.

The **vline()**, **mvvline()**, **mvwvline()** and **wvline()** functions draw a line proceeding toward the last line of the window.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the A_ prefix.

SEE ALSO

border(3X), box(3X), hline_set(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

(ENHANCED CURSES)

NAME

hline_set, mvhline_set, mvvline_set, mvwhline_set, mvwvline_set, vline_set, whline_set, wvline_set — draw lines from complex characters and renditions

SYNOPSIS

```
#include <curses.h>
int hline_set(const cchar_t *wch, int n);
int mvhline_set(int y, int x, const cchar_t *wch, int n);
int mvvline_set(int y, int x, const cchar_t *wch, int n);
int mvwhline_set(WINDOW *win, int y, int x, const cchar_t *wch, int n);
int mvwvline_set(WINDOW *win, int y, int x, const cchar_t *wch, int n);
int vline_set(const cchar_t *wch, int n);
int whline_set(WINDOW *win, const cchar_t *wch, int n);
int wvline_set(WINDOW *win, const cchar_t *wch, int n);
```

h

DESCRIPTION

These functions draw a line in the current or specified window starting at the current or specified position, using *ch*. The line is at most *n* positions long, or as many as fit into the window.

These functions do not advance the cursor position. These functions do not perform special character processing. These functions do not perform wrapping.

The **hline_set()**, **mvhline_set()**, **mvwhline_set()** and **whline_set()** functions draw a line proceeding toward the last column of the same line.

The **vline_set()**, **mvvline_set()**, **mvwvline_set()** and **wvline_set()** functions draw a line proceeding toward the last line of the window.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

SEE ALSO

border_set(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

(Series 800 Only)

NAME

hpib_abort() - stop activity on specified HP-IB bus (OBSOLETED AT 10.30)

SYNOPSIS

```
#include <dvio.h>
int hpib_abort(int eid);
```

DESCRIPTION

hpib_abort() terminates activity on the addressed HP-IB bus by pulsing the IFC line. *eid* is an entity identifier of an open HP-IB raw bus device file obtained from an **open()**, **dup()**, **fcntl()**, or **creat()** call.

hpib_abort() also sets the REN line and clears the ATN line. The status of the SRQ line is not affected. The interface must be the system controller of the bus.

RETURN VALUE

hpib_abort() returns 0 (zero) if successful, or –1 if an error was encountered.

ERRORS

hpib_abort() fails under the following circumstances, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	the specified interface is not the system controller (see DEPENDENCIES below).
[ETIMEDOUT]	a timeout occurred.
[EACCES]	The interface associated with this *eid* is locked by another process and **O_NDELAY** is set for this *eid* (see io_lock(3I)).

DEPENDENCIES

If the interface is not currently the system controller, **hpib_abort()** sets **errno** to [EACCES] instead of to [EIO].

AUTHOR

hpib_abort() was developed by HP.

SEE ALSO

dup(2), creat(2), fcntl(2), open(2).

(Series 800 Only)

NAME

hpib_bus_status() - return status of HP-IB interface (OBSOLETED AT 10.30)

SYNOPSIS

```
#include <dvio.h>
int hpib_bus_status(int eid, int status);
```

DESCRIPTION

hpib_bus_status() obtains status information about an HP-IB channel. *eid* is an entity identifier of an open HP-IB raw bus device file obtained from an **open()**, **dup()**, **fcntl()**, or **creat()** call. *status* is an integer determining what status information is returned for a particular call. The values defined for *status* and their associated meanings are:

REMOTE_STATUS	Is the channel currently in remote state?
SRQ_STATUS	What is the current state of the SRQ line?
NDAC_STATUS	What is the current state of the NDAC line?
SYS_CONT_STATUS	Is the channel currently system controller?
ACT_CONT_STATUS	Is the channel currently active controller?
TALKER_STATUS	Is the channel currently addressed as talker?
LISTENER_STATUS	Is the channel currently addressed as listener?
CURRENT_BUS_ADDRESS	What is the channel's bus address?

The remote-state status is not defined when the interface is the active controller, although reading remote-state status in such a situation is not an error. Determining the status of the NDAC line is not available on all machines, and its use is therefore discouraged to ensure compatibility among various systems. Machines that do not support sensing the NDAC line return an error.

RETURN VALUE

The value returned by **hpib_bus_status()** depends upon the value of *status*. If *status* is **CURRENT_BUS_ADDRESS**, the return value is either the HP-IB bus address or –1 if an error occurred. If *status* is any of the other values, the return value is 0 if the condition is false (the line is clear), 1 if the condition is true (the line is set), or –1 if an error occurred.

ERRORS

hpib_bus_status() fails under the following conditions, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EINVAL]	*status* is not one of the values specified above.

AUTHOR

hpib_bus_status() was developed by HP.

(Series 800 Only)

NAME

hpib_card_ppoll_resp() - control response to parallel poll on HP-IB (OBSOLETED AT 10.30)

SYNOPSIS

```
#include <dvio.h>

int hpib_card_ppoll_resp(int eid, int flag);
```

DESCRIPTION

hpib_card_ppoll_resp() enables or disables an interface for parallel polls. It also controls the sense, and determines the line on which the response is sent. This provides a means for the interface to ignore or respond to a parallel poll according to whether it is enabled to respond.

eid is an entity identifier of an open HP-IB raw bus device file obtained from an **open()**, **dup()**, **fcntl()**, or **creat()** call. *flag* is an integer having one of the following bit patterns:

Bit Pattern	Meaning
10000	Disable parallel poll response.
0SPPP	Enable parallel poll response, where
	S = sense of the response, and
	PPP = 3-bit binary number specifying the line on which the response is sent where the octal values 0 through 7 correspond to lines DIO1 through DIO8.

RETURN VALUE

hpib_card_ppoll_resp() returns 0 (zero) if successful, or –1 if an error was encountered.

ERRORS

hpib_card_ppoll_resp() fails under the following circumstances, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EACCES]	The interface associated with this *eid* is locked by another process and **O_NDELAY** is set for this *eid* (see *io_lock*(3I)).
[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EINVAL]	the device cannot respond on the line number specified by *flag*.
[ETIMEDOUT]	a timeout occurred.

DEPENDENCIES

Since the sense and response line number are not programmable on the HP 27110B HP-IB interface, the equivalent parallel poll configuration commands are sent over the HP-IB to the interface. Therefore, this function fails if the interface is not active controller.

AUTHOR

hpib_card_ppoll_resp() was developed by HP.

SEE ALSO

dup(2), creat(2), fcntl(2), open(2), hpib_spoll(3I), hpib_pass_ctl(3I).

(Series 800 Only)

NAME

hpib_eoi_ctl() - control EOI mode for HP-IB file (OBSOLETED AT 10.30)

SYNOPSIS

```
#include <dvio.h>

int hpib_eoi_ctl(int eid, int flag);
```

DESCRIPTION

hpib_eoi_ctl() enables you to turn EOI mode on or off. *eid* is an entity identifier of an open HP-IB raw device file obtained from an **open()**, **dup()**, **fcntl()**, or **creat()** call. *flag* is an integer which, if non-zero, enables EOI mode, and otherwise disables it.

EOI mode causes the last byte of all subsequent write operations to be written out with the EOI line asserted, signifying the end of the data transmission. By default, EOI mode is disabled when the device file is opened.

Entity identifiers for the same device file obtained by separate **open()** requests have their own EOI modes associated with them. Entity identifiers for the same device file obtained by **dup()** or inherited by a **fork()** request share the same EOI mode. In the latter case, if one process enables EOI mode, then EOI mode is in effect for all such entity identifiers.

RETURN VALUE

hpib_eoi_ctl() returns a 0 (zero) if successful, or –1 if an error was encountered.

ERRORS

hpib_eoi_ctl() fails under any of the following circumstances and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to an HP-IB device file.

DEPENDENCIES

EOI mode is enabled when the device file is first opened.

AUTHOR

hpib_eoi_ctl() was developed by HP.

(Series 800 Only)

NAME

hpib_io() - perform I/O with an HP-IB channel from buffers (OBSOLETED AT 10.30)

SYNOPSIS

```
#include <dvio.h>
int hpib_io(int eid, struct iodetail *iovec, size_t iolen);
```

DESCRIPTION

hpib_io() performs and controls read and/or write operations on the specified HP-IB bus. *eid* is an entity identifier of an open HP-IB raw bus device file obtained from an **open()**, **dup()**, **fcntl()**, or **creat()** call.

Parameters are as follows:

iovec Pointer to an array of structures of the form:

```
struct iodetail {
    char  mode;
    char  terminator;
    int   count;
    char  *buf;
};
```

The *iodetail* structure is defined in the include file <**dvio.h**>.

iolen Specifies the number of structures in *iovec*.

iodetail Structure

Elements in the **iodetail** structure are:

mode Describes what is to be done during I/O on the buffer pointed to by *buf*. *mode* is constructed by OR-ing flags from the following list:

One and only one of the following two flags *must* be specified:

HPIBREAD Perform a read of the HP-IB bus, placing data into the accompanying buffer.

HPIBWRITE Perform a write to the HP-IB bus, using data from the accompanying buffer.

The following flags can be used in most combinations (not all combinations are valid), or not at all:

HPIBATN Data is written with ATN enabled.

HPIBEOI Data written is terminated with EOI (this flag is ignored when **HPIBATN** is enabled).

HPIBCHAR Data read is terminated with the character given in the *terminator* element of the *iodetail* structure.

terminator Describes the termination character, if any, that should be checked for on input. *count* is an integer specifying the maximum number of bytes to be transferred.

A read operation terminates when either *count* is matched, an EOI is detected, or the designated *terminator* is detected (if **HPIBCHAR** is set in *mode*).

A write operation terminates when *count* is matched, and the final byte is sent with EOI asserted (if **HPIBEOI** is set in *mode*).

If **HPIBATN** is set in *mode*, write operations occur with ATN enabled. Setting **HPIBATN** for a read operation is ignored and has no effect.

The members of the **iovec** array are accessed in order.

RETURN VALUE

If all transactions are successful, **hpib_io()** returns a zero and updates the *count* element in each structure in the **iovec** array to reflect the actual number of bytes read or written.

If an error is encountered during a transaction defined by an element of **iovec**, **hpib_io()** returns without completing any transactions that might follow. In particular, if an error occurs, **hpib_io()** returns a –1, and the *count* element of the transaction that caused the error is set to –1.

ERRORS

hpib_io() fails under any of the following circumstances, and sets **errno** (see *errno*(2)) to the value indicated:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to an HP-IB raw bus device file.

[ETIMEDOUT] a timeout occurred.

[EIO] *eid* is not the active controller.

DEPENDENCIES

If the interface is not currently the active controller, **hpib_io()** sets **errno** to [EACCES] instead of to [EIO].

AUTHOR

hpib_io() was developed by HP.

(Series 800 Only)

NAME

hpib_pass_ctl() - change active controllers on HP-IB (OBSOLETED AT 10.30)

SYNOPSIS

```
#include <dvio.h>
int hpib_pass_ctl(int eid, int ba);
```

DESCRIPTION

hpib_pass_ctl() passes control of a bus to an inactive controller on that bus. The inactive controller becomes the active controller of that bus. *eid* is an entity identifier of an open HP-IB raw bus device file obtained from an **open()**, **dup()**, **fcntl()**, or **creat()** call. *ba* is the bus address of the intended device.

Not all devices can accept control. Pass control passes only active control of the bus; it cannot pass system control of the bus. The specified interface must be the current active controller but need not be the system controller. The pass control operation does not suspend program execution if the inactive controller does not take active control of the bus. However, the interface is no longer active controller.

RETURN VALUE

hpib_pass_ctl() returns 0 (zero) if successful, or −1 if an error was encountered.

ERRORS

hpib_pass_ctl() fails under any of the following circumstances, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	the interface is not the active controller.
[ETIMEDOUT]	a timeout occurred.
[EINVAL]	*ba* is not a valid HP-IB bus address.
[EACCES]	The interface associated with this *eid* is locked by another process and **O_NDELAY** is set for this *eid* (see *io_lock*(3I)).

DEPENDENCIES

If the interface is not currently the active controller, **hpib_pass_ctl()** sets **errno** to [EACCES] instead of to [EIO].

AUTHOR

hpib_pass_ctl() was developed by HP.

(Series 800 Only)

NAME

hpib_ren_ctl( ) - control the Remote Enable line on HP-IB (OBSOLETED AT 10.30)

SYNOPSIS

```
#include <dvio.h>
int hpib_ren_ctl(int eid, int flag);
```

DESCRIPTION

hpib_ren_ctl() enables/disables the Remote Enable (REN) line depending upon the value of *flag*. *eid* is an entity identifier of an open HP-IB raw bus device file obtained from an **open()**, **dup()**, **fcntl()**, or **creat()** call. *flag* is an integer which, if non-zero, enables the REN line, and otherwise disables it.

hpib_ren_ctl() can be used in conjunction with **hpib_send_cmnd()** to place devices into the remote state or local state. The REN line is normally enabled at all times, and is in this state at power-up. Only the system controller can enable or disable the REN line.

RETURN VALUE

hpib_ren_ctl() returns 0 (zero) if successful, or –1 if an error was encountered.

ERRORS

hpib_ren_ctl() fails under the following circumstances, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	the interface is not the system controller.

AUTHOR

hpib_ren_ctl() was developed by HP.

(Series 800 Only)

NAME

hpib_rqst_srvce() - allow interface to enable SRQ line on HP-IB (OBSOLETED AT 10.30)

SYNOPSIS

```
#include <dvio.h>

int hpib_rqst_srvce(int eid, int cv);
```

DESCRIPTION

hpib_rqst_srvce() specifies a response byte to be sent by the interface when it is serially polled by the active controller. *eid* is an entity identifier of an open HP-IB raw bus device file obtained from an **open()**, **dup()**, **fcntl()**, or **creat()** call. *cv* is an integer control value representation of the desired response byte.

hpib_rqst_srvce() optionally enables the SRQ line depending upon the response byte. If bit 6 of the response byte is set, the SRQ line is enabled. It remains enabled until the active controller conducts a serial poll or until the computer executes the request function with bit 6 cleared. However, the SRQ line is not enabled as long as the interface is active controller. If bit 6 is set, the interface remembers its response byte, and enables the SRQ line when control is passed to another device on the bus.

The response byte is structured as follows:

Bit	Meaning
0	SPOLL bit (least significant bit of response byte)
1	SPOLL bit
2	SPOLL bit
3	SPOLL bit
4	SPOLL bit
5	SPOLL bit
6	SRQ line
7	SPOLL bit (most significant bit of response byte)

RETURN VALUE

hpib_rqst_srvce() returns 0 (zero) if successful, or −1 if an error was encountered.

ERRORS

hpib_rqst_srvce() fails under the following circumstances, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to an HP-IB raw bus device file.

[ETIMEDOUT] a timeout occurred.

[EACCES] The interface associated with this *eid* is locked by another process and **O_NDELAY** is set for this *eid* (see io_lock(3I)).

DEPENDENCIES

The HP 27110B HP-IB interface card allows only bit 6 to be set. All other bits are cleared.

AUTHOR

hpib_rqst_srvce() was developed by HP.

(Series 800 Only)

NAME

hpib_send_cmnd() - send command bytes over HP-IB (OBSOLETED AT 10.30)

SYNOPSIS

```
#include <dvio.h>

int hpib_send_cmnd(int eid, const char *ca, int length);
```

DESCRIPTION

hpib_send_cmnd() sends specified arbitrary bytes of information on the HP-IB with the ATN line asserted, providing a means to configure and control the bus. *eid* is an entity identifier of an open HP-IB raw bus device file obtained from an **open()**, **dup()**, **fcntl()**, or **creat()** call. *ca* is a character pointer to a string of bytes to be written to the HP-IB bus as commands. *length* is an integer specifying the number of bytes in the string pointed to by *ca*.

The interface must currently be the active controller in order to send commands over the bus.

Note that for all HP-IB interfaces, both built-in and plug-in, the most significant bit of each byte is overwritten with a parity bit. All commands are written with odd parity.

RETURN VALUE

hpib_send_cmnd() returns 0 (zero) if successful, or −1 if an error was encountered.

ERRORS

hpib_send_cmnd() fails under the following circumstances, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	the interface is not currently the active controller.
[ETIMEDOUT]	a timeout occurred.
[EACCES]	The interface associated with this *eid* is locked by another process and **O_NDELAY** is set for this *eid* (see *io_lock*(3I)).
[EINVAL]	The value specified for *length* is invalid, either less than or equal to 0 or greater than **MAX_HPIB_COMMANDS** as defined in <**dvio.h**>.

DEPENDENCIES

If the interface is not currently the active controller, **hpib_send_cmnd()** sets **errno** to [EACCES] instead of [EIO].

AUTHOR

hpib_send_cmnd() was developed by HP.

SEE ALSO

dup(2), creat(2), fcntl(2), open(2), hpib_parity_ctl(3I).

(Series 800 Only)

NAME

hpib_spoll() - conduct a serial poll on HP-IB bus (OBSOLETED AT 10.30)

SYNOPSIS

```
#include <dvio.h>
int hpib_spoll(int eid, int ba);
```

DESCRIPTION

hpib_spoll() conducts a serial poll of the specified device. *eid* is an entity identifier of an open HP-IB raw bus device file obtained from an **open()**, **dup()**, **fcntl()**, or **creat()** call. *ba* is the bus address of the intended device.

hpib_spoll() Polls a single device for its response byte. The information stored in the response byte is device specific with the exception of bit 6. If bit 6 of the response byte is set, the addressed device has asserted the SRQ line, and is requesting service (note that the least significant bit of the response byte is bit 0).

Not all devices respond to the serial poll function. Consult device documentation. Specifying a device that does not support serial polling may cause a timeout error or suspend your program indefinitely. The interface cannot serial poll itself. The interface must be the active controller.

h

RETURN VALUE

If **hpib_spoll()** is successful, the device response byte is returned in the least significant byte of the return value. Otherwise, –1 is returned, indicating an error.

ERRORS

hpib_spoll() fails under the following circumstances, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	the interface is not the active controller.
[ETIMEDOUT]	the device polled did not respond before timeout.
[EINVAL]	*ba* is the address of the polling interface itself.
[EACCES]	The interface associated with this *eid* is locked by another process and **O_NDELAY** is set for this *eid* (see io_lock(3I)).

DEPENDENCIES

If the interface is not currently the active controller, **hpib_spoll()** sets **errno** to [EACCES] instead of to [EIO].

AUTHOR

hpib_spoll() was developed by HP.

SEE ALSO

dup(2), creat(2), fcntl(2), open(2), hpib_rqst_srvce(3I).

(Series 800 Only)

NAME

hpib_status_wait() - wait until the requested status condition becomes true (OBSOLETED AT 10.30)

SYNOPSIS

```
#include <dvio.h>
int hpib_status_wait(int eid, int status); #include <dvio.h>
```

DESCRIPTION

hpib_status_wait() waits until a specific condition has occurred before returning. *eid* is an entity identifier of an open HP-IB raw bus device file obtained from an **open()**, **dup()**, **fcntl()**, or **creat()** call. *status* is an integer specifying what information is returned. The possible values for *status* and their associated meanings are:

WAIT_FOR_SRQ	Wait until SRQ line is enabled.
WAIT_FOR_CONTROL	Wait until this channel is active controller.
WAIT_FOR_TALKER	Wait until this channel is addressed as talker.
WAIT_FOR_LISTENER	Wait until this channel is addressed as listener.

The wait is subject to the current timeout in effect. If a timeout occurs before the desired condition occurs, the function returns with an error.

RETURN VALUE

hpib_status_wait() returns zero when the condition requested becomes true. A value of –1 is returned if an error occurs. A –1 is also returned if a timeout occurs before the desired condition becomes true.

ERRORS

hpib_status_wait() fails under the following circumstances, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[ETIMEDOUT]	a timeout occurred.
[EINVAL]	*status* contains an invalid value.
[EACCES]	the interface associated with this *eid* is locked by another process and **O_NDELAY** is set for this *eid* (see io_lock(3I)).

AUTHOR

hpib_status_wait() was developed by HP.

(Series 800 Only)

NAME

hpib_wait_on_ppoll() - wait until a particular parallel poll value occurs (OBSOLETED AT 10.30)

SYNOPSIS

```
#include <dvio.h>
int hpib_wait_on_ppoll(int eid, int mask, int sense);
```

DESCRIPTION

hpib_wait_on_ppoll() waits for a parallel poll response to occur on one or more lines. *eid* is an entity identifier of an open HP-IB raw bus device file.

The *mask* argument specifies on which lines the parallel poll response is expected. The value of *mask* is treated as an eight-bit binary number where the least significant bit corresponds to line DIO1; the most significant bit to DIO8. For example, if you want to wait for a response on lines DIO2 and DIO6, the corresponding binary number is 00010010, so a hexadecimal value of 12 should be passed as the *mask* argument.

The *sense* argument specifies what response is expected on the selected lines. The value of *sense* is constructed in the same way as *mask*; eight bits for eight lines. If a bit in *sense* is set, the function returns when the line corresponding to that bit is *cleared*. If a bit in *sense* is clear, the function returns when the corresponding line is *set*. Using the previous example, if *mask* is 0x12 and *sense* is 00000010 (0x02 hexadecimal), the function returns when line DIO5 is set, or when line DIO2 is clear.

RETURN VALUE

hpib_wait_on_ppoll() returns a value of –1 if an error or timeout condition occurs. Upon successful completion, the function returns the response byte XOR-ed with the *sense* value and AND-ed with the *mask*.

ERRORS

hpib_wait_on_ppoll() fails and sets **errno** to indicate the error if any of the following is true:

[EACCES]	The interface associated with this *eid* is locked by another process and *O_NDELAY* is set for this *eid* (see *io_lock*(3I)).
[EBADF]	The *eid* argument is not a valid open entity identifier.
[ENOTTY]	The *eid* argument does not refer to an HP-IB raw bus device file.
[EINVAL]	An invalid mask is received.
[EIO]	The interface is not currently the active controller.
[ETIMEDOUT]	A timeout occurred (Series 800 only).

DEPENDENCIES

If the interface is not currently the active controller, **hpib_wait_on_ppoll()** sets **errno** to [EACCES] instead of to [EIO].

AUTHOR

hpib_wait_on_ppoll() was developed by HP.

NAME

HPPACADDD, HPPACCMPD, HPPACCVAD, HPPACCVBD, HPPACCVDA, HPPACCVDB, HPPACDIVD, HPPACLONGDIVD, HPPACMPYD, HPPACNSLD, HPPACSLD, HPPACSRD, HPPACSUBD - 3000-mode packed-decimal library

SYNOPSIS

```
#include <hppac.h>

void HPPACADDD(
     unsigned char *operand2,
     int op2digs,
     unsigned char *operand1,
     int op1digs,
     enum HPPAC_CC *compcode,
     int *pacstatus
);

void HPPACCMPD(
     unsigned char *operand1,
     int op1digs,
     unsigned char *operand2,
     int op2digs,
     enum HPPAC_CC *compcode,
     int *pacstatus
);

void HPPACCVAD(
     unsigned char *target,
     int targetdigs,
     unsigned char *source,
     int sourcedigs,
     enum HPPAC_CC *compcode,
     int *pacstatus
);

void HPPACCVBD(
     unsigned char *target,
     int targetdigs,
     unsigned short *source,
     int sourcewords,
     enum HPPAC_CC *compcode,
     int *pacstatus
);

void HPPACCVDA(
     unsigned char *target,
     int targetdigs,
     unsigned char *source,
     int sign_control,
     enum HPPAC_CC *compcode,
     int *pacstatus
);

void HPPACCVDB(
     unsigned short *target,
     unsigned char *source,
     int sourcedigs,
     enum HPPAC_CC *compcode,
     int *pacstatus
);

void HPPACDIVD(
     unsigned char *operand2,
     int op2digs,
     unsigned char *operand1,
     int op1digs,
     enum HPPAC_CC *compcode,
```

```
        int *pacstatus
    );
    void HPPACLONGDIVD(
        unsigned char *operand2,
        int op2digs,
        unsigned char *operand1,
        int op1digs,
        enum HPPAC_CC *compcode,
        int *pacstatus
    );
    void HPPACMPYD(
        unsigned char *operand2,
        int op2digs,
        unsigned char *operand1,
        int op1digs,
        enum HPPAC_CC *compcode,
        int *pacstatus
    );
    void HPPACNSLD(
        unsigned char *operand2,
        int op2digs,
        unsigned char *operand1,
        int op1digs,
        int *shift_amt,
        enum HPPAC_CC *compcode,
        int *pacstatus,
        int *carry
    );
    void HPPACSLD(
        unsigned char *operand2,
        int op2digs,
        unsigned char *operand1,
        int op1digs,
        int shift_amt,
        enum HPPAC_CC *compcode,
        int *pacstatus,
        int *carry
    );
    void HPPACSRD(
        unsigned char *operand2,
        int op2digs,
        unsigned char *operand1,
        int op1digs,
        int shift_amt,
        enum HPPAC_CC *compcode,
        int *pacstatus
    );
    void HPPACSUBD(
        unsigned char *operand2,
        int op2digs,
        unsigned char *operand1,
        int op1digs,
        enum HPPAC_CC *compcode,
        int *pacstatus
    );
```

h

DESCRIPTION

This set of calls invokes the library functions for emulating 3000-mode (MPE V/E) packed-decimal operations. These functions are in library **libcl** which is searched when the option **-lcl** is used with *cc*(1) or *ld*(1).

HPPACADDD()	Performs packed-decimal addition.
HPPACCMPD()	Compares two packed-decimal numbers.
HPPACCVAD()	Converts an ASCII representation to packed-decimal.
HPPACCVBD()	Converts a binary representation to packed-decimal.
HPPACCVDA()	Converts a packed-decimal number to ASCII.
HPPACCVDB()	Converts a packed-decimal number to binary.
HPPACDIVD()	Performs packed-decimal division.
HPPACLONGDIVD()	Performs packed-decimal division (alternate routine).
HPPACMPYD()	Performs packed-decimal multiplication.
HPPACNSLD()	Performs a packed-decimal normalizing left shift.
HPPACSLD()	Performs a packed-decimal left shift.
HPPACSRD()	Performs a packed-decimal right shift.
HPPACSUBD()	Performs packed-decimal subtraction.

h

For all operations, the value returned in the variable to which the *compcode* argument points is one of the following values of type **enum HPPAC_CC**:

HPPAC_CCG	Result > 0 or operand1 > operand2
HPPAC_CCL	Result < 0 or operand1 < operand2
HPPAC_CCE	Result == 0 or operand1 == operand2

For all operations, the value returned in the variable to which the *pacstatus* argument points is one of the following values of type **enum HPPAC_STATUS**. Their meanings are intended to be obvious:

HPPAC_NO_ERROR
HPPAC_DECIMAL_OVERFLOW
HPPAC_INVALID_ASCII_DIGIT
HPPAC_INVALID_PACKED_DECIMAL_DIGIT
HPPAC_INVALID_SOURCE_WORD_COUNT
HPPAC_INVALID_DECIMAL_OPERAND_LENGTH
HPPAC_DECIMAL_DIVIDE_BY_ZERO

AUTHOR

The HPPAC library was developed by HP.

SEE ALSO

Compiler Library/XL Reference Manual

NAME

hsearch(), hcreate(), hdestroy() - manage hash search tables

SYNOPSIS

```
#include <search.h>

ENTRY *hsearch(ENTRY item, ACTION action);

int hcreate(size_t nel);

void hdestroy(void);
```

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. Only pointers are copied, so the calling routine must store the data (the value of the "key" must be unique). *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 can be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

hdestroy() destroys the search table, and can be followed by another call to **hcreate()**.

EXAMPLE

The following example reads in strings followed by two numbers and stores them in a hash table, discarding duplicates. It then reads in strings and finds the matching entry in the hash table and prints 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);
            }
        }
    }
```

APPLICATION USAGE

hcreate(), **hdestroy()** and **hsearch()** are thread-safe. These interfaces are not async-cancel-safe.

RETURN VALUE

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

hcreate() returns zero if it cannot allocate sufficient space for the table.

WARNINGS

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

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

SEE ALSO

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

STANDARDS CONFORMANCE

hsearch(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

hcreate(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

hdestroy(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

NAME
hypot( ) - Euclidean distance function

SYNOPSIS
```
#include <math.h>
double hypot(double x, double y);
```

DESCRIPTION
hypot() returns **sqrt(***x*******x***+***y*******y***)**, taking precautions against unwarranted overflows.

The ISO/ANSI C committee has approved the **hypot()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE
If *x* or *y* is NaN, **hypot()** returns NaN.

If the correct value would overflow, **hypot()** returns **HUGE_VAL**.

If the correct value after rounding would be smaller in magnitude than **MINDOUBLE**, **hypot()** returns zero.

ERRORS
No errors are defined.

SEE ALSO
sqrt(3M), math(5), values(5).

STANDARDS CONFORMANCE
hypot(): SVID3, XPG4.2

NAME

iconv, iconv_open, iconv_close - codeset conversion routines

SYNOPSIS

```
#include <iconv.h>

iconv_t iconv_open(const char *tocode, const char *fromcode);

size_t iconv(
     iconv_t      cd,
     const char   **inbuf,
     size_t       *inbytesleft,
     char         **outbuf,
     size_t       *outbytesleft
);

int iconv_close(iconv_t cd);
```

Remarks

These interfaces conform to the XPG4 standard, and should be used instead of the the 9.0 **iconv** interfaces, such as, **iconvopen()**, **iconvclose()**, **iconvsize()**, **iconvlock()**, **ICONV()**, **ICONV1()**, and **ICONV2()**.

Refer to the white paper entitled, *iconv Customization*, for an understanding of the conversion process. The white paper explains how iconv uses tables and methods to do the conversions. This white paper also shows you how to customize your own conversions. The white paper is in **/usr/share/doc/iconv.ps** (postscript file) and **/usr/share/doc/iconv.txt** (plain ASCII text file).

DESCRIPTION

The entries in the **config.iconv** file are the set of conversions that are supported by *iconv*(3C). The first two columns correspond to the *fromcode* and *tocode* names. These names may be directly used or their corresponding aliases may be used as parameters to **iconv_open()**.

iconv_open() Returns a conversion descriptor that describes a conversion from the codeset specified by the string pointed to by the *fromcode* argument to the codeset specified by the *tocode* argument.

A conversion descriptor remains valid in a process until that process closes it.

The *fromcode* and *tocode* arguments must have a corresponding entry in the configuration file **/usr/lib/nls/iconv/config.iconv**. (See FILES section.)

iconv() Converts a sequence of characters from one codeset that is contained in the array specified by *inbuf*, into a sequence of corresponding characters in another codeset, contained in the array specified by *outbuf*. The codesets are those specified in the **iconv_open()** call that returned the conversion descriptor *cd*. The *inbuf* argument points to a variable that points to the first character in the input buffer and *inbytesleft* indicates the number of remaining bytes in the buffer being converted. The *outbuf* argument points to a variable that points to the first available byte in the output buffer and *outbytesleft* indicates the number of the available remaining bytes in the buffer.

If a sequence of input bytes does not form a valid character in the specified codeset, conversion stops after the previous successfully converted character. If the input buffer ends with an incomplete character or shift sequence (see section on Special Usage), conversion stops after the previous successfully converted character. If the output buffer is not large enough to hold the entire converted output, conversion stops just prior to the character that would cause the output buffer to overflow. The variable pointed to by *inbuf* is updated to point to the byte following the last byte successfully used in the conversion. The value pointed to by *inbyesleft* is reduced to reflect the number of bytes still not converted in the input buffer. The variable pointed to by *outbuf* is updated to point to the byte following the last byte of converted output data. The value pointed to by *outbytesleft* is reduced to reflect the number of bytes still available in the output buffer.

If **iconv()** encounters a character in the input buffer that is legal but for which an identical character does not exist in the target codeset, **iconv()** maps this character

to a pre-defined character, called the "galley character" that is defined at the time of table generation. (See *genxlt*(1)).

iconv_close() Deallocates the conversion descriptor *cd* and all other associated resources allocated by **iconv_open()**.

APPLICATION USAGE

iconv_open(), **iconv()** and **iconv_close()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing these interfaces.

Portable applications must assume that conversion descriptors are not valid after calls to any of the **exec** functions.

Special Usage

In state-dependent encodings, the characters are interpreted depending on "state" of the input. State shifts occur when a specific sequence of bytes are seen in the input. These sequences will change the way subsequent characters are interpreted (that is, initially the characters may be single-byte characters, after a state shift, subsequent characters may be interpreted as two-byte characters). For state-dependent encodings, the conversion descriptor after **iconv_open()** is in a codeset-dependent initial shift state, ready for immediate use with **iconv()**.

For state-dependent encodings, the conversion descriptor *cd* is placed into its initial shift state by a call to **iconv()** for which the *inbuf* is a null pointer, or for which *inbuf* points to a null pointer. When **iconv()** is called in this way, and *outbuf* is not a null pointer or a pointer to a null pointer, and *outbytesleft* points to a positive value, **iconv()** places the byte sequence to change the output buffer to its initial shift state. If the output buffer is not large enough to hold the entire reset sequence, **iconv()** fails and sets **errno** to **[E2BIG]**. Subsequent calls with *inbuf* set to other than a null pointer or a pointer to a null pointer cause the conversion to take place from the current state of the conversion descriptor.

For state-dependent encodings, the conversion descriptor is updated to reflect the shift state in effect at the end of the last successfully converted byte sequence.

RETURN VALUE

iconv_open() Upon successful completion, **iconv_open()** returns a conversion descriptor for use on subsequent calls to **iconv()**. Otherwise **iconv_open()** returns (**iconv_t**)–1 and sets **errno** to indicate the error.

iconv() **iconv()** updates the variables pointed to by the arguments to reflect the extent of conversion, and returns the the number of non-identical conversions performed. If the entire string in the input buffer is converted, the value pointed to by *inbytesleft* is zero. If an error occurs, **iconv()** returns (**size_t**)–1 and sets **errno** to indicate the error.

iconv_close() Upon successful completion, **iconv_close()** returns a value of zero. Otherwise it returns –1 and sets **errno** to indicate the error.

ERRORS

iconv_open() fails if any of the following conditions are encountered:

[ENOMEM] Insufficient storage space is available.

[EINVAL] The conversion specified by the *fromcode* and *tocode* is not supported, or the table or method specified in the configuration file could not be read or loaded correctly. This error will also occur if the configuration file itself is faulty.

iconv() fails if any of the following conditions are encountered:

[EILSEQ] Input conversion stopped due to an input character that does not belong to the input codeset, or if the conversion table does not contain an entry corresponding to this input character and a galley character was not defined for that particular table.

[E2BIG] Input conversion stopped due to lack of space in the output buffer.

[EINVAL] Input conversion stopped due to an incomplete character or shift sequence at the end of the input buffer.

[EBADF] The *cd* argument is not a valid open conversion descriptor.

iconv_close() fails if any of the following conditions are encountered:

[EBADF] The conversion descriptor is invalid.

EXAMPLES

The following example shows how the *iconv*(3C) interfaces maybe used for conversions.

```
#include <iconv.h>
#include <errno.h>

main()
{
        ...
        convert("roman8", "iso88591", fd);
        ...
}

int
convert(tocode, fromcode, Input)
char *tocode;                       /* tocode name */
char *fromcode                      /* fromcode name */
int Input;                          /* input file descriptor */
{
    extern void error();            /* local error message */

    iconv_t cd;                     /* conversion descriptor */
    unsigned char *table;           /* ptr to translation table */
    int bytesread;                  /* num bytes read into input buffer */
    unsigned char inbuf[BUFSIZ];    /* input buffer */
    unsigned char *inchar;          /* ptr to input character */
    int inbytesleft;                /* num bytes left in input buffer */
    unsigned char outbuf[BUFSIZ];   /* output buffer */
    unsigned char *outchar;         /* ptr to output character */
    int outbytesleft;               /* num bytes left in output buffer */
    size_t ret_val;                 /* number of conversions */

    /* Initiate conversion -- get conversion descriptor */
    if ((cd = iconv_open(tocode, fromcode)) == (iconv_t)-1) {
        error(FATAL, BAD_OPEN);
    }

    inbytesleft = 0;                        /* no. of bytes converted */
    /* translate the characters */
    for ( ;; ) {
                /*
                 * if any bytes are leftover, they will be in the
                 * beginning of the buffer on the next read().
                 */

                inchar = inbuf;          /* points to input buffer */
                outchar = outbuf;        /* points to output buffer */
                outbytesleft = BUFSIZ;   /* no of bytes to be converted */
                if ((bytesread = read(Input, inbuf+inbytesleft,
                                      (size_t)BUFSIZ-inbytesleft)) < 0) {
                        perror("prog");
                        return BAD;
                }
                if (!(inbytesleft += bytesread)) {
                        break;           /* end of conversions */
                }

                ret_val = iconv(cd, &inchar, &inbytesleft,
                                &outchar, &outbytesleft);
```

```
                if (write(1, outbuf, (size_t)BUFSIZ-outbytesleft) < 0) {
                        perror("prog");
                        return BAD;
                }

                /* iconv() returns the number of non-identical conversions
                 * performed. If the entire string in the input buffer is
                 * converted, the value pointed to by inbytesleft will be
                 * zero. If the conversion stopped due to any reason, the
                 * value pointed to by inbytesleft will be non-zero and
                 * errno is set to indicate the condition.
                 */
                if ((ret_val == -1) && (errno == EINVAL)) {
                        /* Input conversion stopped due to an incomplete
                         * character or shift sequence at the end of the
                         * input buffer.
                         */
                        /* Copy data left, to the start of buffer */
                        memcpy((char *)inbuf, (char *)inchar,
                               (size_t)inbytesleft);
                } else if ((ret_val == -1) && (errno == EILSEQ)) {
                        /* Input conversion stopped due to an input byte
                         * that does not belong to the input codeset.
                         */
                        error(FATAL, BAD_CONVERSION);
                } else if ((ret_val == -1) && (errno == E2BIG)) {
                        /* Input conversion stopped due to lack of space
                         * in the output buffer. inbytesleft has the
                         * number of bytes to be converted.
                         */
                        memcpy((char *)inbuf, (char *)inchar,
                               (size_t)inbytesleft);
                }
                /* Go back and read from the input file. */
        }

        /* end conversion & get rid of the conversion table */
        if (iconv_close(cd) == BAD) {
                error(FATAL, BAD_CLOSE);
        }
        return GOOD;
}
```

WARNINGS

If you use *iconv*(3C) and compile/link your application archive, please note that *iconv*(3C) has a dependency on *libdld.sl* that will require a change to the compile/link command:

Compile :

cc -Wl,-a,archive -Wl,-E -Wl,+n -l:libdld.sl -o *outfile source*

Or compile with **CCOPTS** and **LDOPTS**:

export CCOPTS="-Wl,-a,archive *options* **-Wl,-E -l:libdld.sl"**

export LDOPTS="*options* **-E +n -l:libdld.sl"**

cc -o *outfile source*

The option **-Wl,-a,archive** is positionally dependent and should occur at the beginning of the compile line. For optimum compatibility in future releases, you should avoid using archive libc with other shared libraries except for libdld.sl as needed above.

There is a corner-case situation for multi-byte characters that is not correctly handled by *iconv*(3C). If the last character in the file being converted is an invalid multi-byte character, *iconv*(3C) returns EINVAL instead of EILSEQ. The application can get around this by checking whether EOF is reached or if this is

the last buffer being converted. In this case, EINVAL should be treated as EILSEQ.

AUTHOR

iconv was developed by HP.

FILES

/usr/lib/nls/iconv/tables	Directory containing tables used for conversion.
/usr/lib/nls/iconv/methods	Directory containing methods used for conversion.
/usr/lib/nls/iconv/config.iconv	Configuration file is used by **iconv_open()** to check if the requested conversion is supported, and if so, to determine which table and/or method is used for the conversion.

SEE ALSO

iconv(1), genxlt(1).

STANDARDS CONFORMANCE

iconv_open(): XPG4
iconv(): XPG4
iconv_close(): XPG4

i

(ENHANCED CURSES)

NAME

idcok — enable or disable use of hardware insert- and delete-character features

SYNOPSIS

```
#include <curses.h>
void idcok(WINDOW *win, bool bf);
```

DESCRIPTION

The **idcok()** function specifies whether the implementation may use hardware insert- and delete-character features in *win* if the terminal is so equipped. If *bf* is TRUE, use of these features in *win* is enabled. If *bf* is FALSE, use of these features in *win* is disabled. The initial state is TRUE.

RETURN VALUE

The **idcok()** function does not return a value.

ERRORS

No errors are defined.

SEE ALSO

clearok(3X), doupdate(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

i

NAME

ilogb() - returns an unbiased exponent

SYNOPSIS

```
#include <math.h>
int ilogb(double x);
```

DESCRIPTION

The **ilogb()** function returns the exponent part of *x*. Formally, the return value is the integral part of log base *r* of |*x*| as a signed integral value, for nonzero *x*, where *r* is the radix of the machine's floating point arithmetic. The argument *x* is a double-precision floating-point value.

Note: **ilogb(***x***)** is equivalent to **(int)logb(***x***)** for all values of *x* except NaN, ±INFINITY, and zero.

The ISO/ANSI C committee has approved the **ilogb()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

If *x* is NaN, **ilogb()** returns **INT_MIN**.

If *x* is ±INFINITY, **ilogb()** returns **INT_MAX**.

If *x* is zero, **ilogb()** returns **INT_MIN**.

ERRORS

No errors are defined.

SEE ALSO

logb(3M), scalb(3M), scalbn(3M), math(5), limits(5).

STANDARDS CONFORMANCE

ilogb(): XPG4.2

(ENHANCED CURSES)

NAME

immedok — enable or disable immediate terminal refresh

SYNOPSIS

```
#include <curses.h>
void immedok(WINDOW *win, bool bf);
```

DESCRIPTION

The **immedok()** function specifies whether the screen is refreshed whenever the window pointed to by win is changed. If *bf* is TRUE, the window is implicitly refreshed on each such change. If *bf* is FALSE, the window is not implicitly refreshed. The initial state is FALSE.

RETURN VALUE

The **immedok()** function does not return a value.

ERRORS

No errors are defined.

APPLICATION USAGE

The **immedok()** function is useful for windows that are used as terminal emulators.

SEE ALSO

clearok(3X), doupdate(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

(ENHANCED CURSES)

NAME

in_wch, mvin_wch, mvwin_wch, win_wch — input a complex character and rendition from a window

SYNOPSIS

```
#include <curses.h>
int in_wch(cchar_t *wcval);
int mvin_wch(int y, int x, cchar_t *wcval);
int mvwin_wch(WINDOW *win, int y, int x, cchar_t *wcval);
int win_wch(WINDOW *win, cchar_t *wcval);
```

DESCRIPTION

These functions extract the complex character and rendition from the current or specified position in the current or specified window into the object pointed to by *wcval*.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

SEE ALSO

<curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

in_wchnstr, in_wchstr, mvin_wchnstr, mvin_wchstr, mvwin_wchnstr, mvwin_wchstr, win_wchnstr, win_wchstr — input an array of complex characters and renditions from a window

SYNOPSIS

```
#include <curses.h>
int in_wchnstr(cchar_t *wchstr, int n);
int in_wchstr(cchar_t *wchstr);
int mvin_wchnstr(int y, int x, cchar_t *wchstr, int n);
int mvin_wchstr(int y, int x, cchar_t *wchstr);
int mvwin_wchnstr(WINDOW *win, int y, int x, cchar_t *wchstr, int n);
int mvwin_wchstr(WINDOW *win, int y, int x, cchar_t *wchstr);
int win_wchnstr(WINDOW *win, cchar_t *wchstr, int n);
int win_wchstr(WINDOW *win, cchar_t *wchstr);
```

DESCRIPTION

These functions extract characters from the current or specified window, starting at the current or specified position and ending at the end of the line, and place them in the array pointed to by *wchstr* .

The **in_wchnstr()**, **mvin_wchnstr()**, **mvwin_wchnstr()** and **win_wchnstr()** fill the array with at most *n* **cchar_t** elements.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

Reading a line that overflows the array pointed to by *wchstr* with **in_wchstr()**, **mvin_wchstr()**, **mvwin_wchstr()** or **win_wchstr()** causes undefined results. The use of **in_wchnstr()**, **mvin_wchnstr()**, **mvwin_wchnstr()** or **win_wchnstr()**, respectively, is recommended.

SEE ALSO

in_wch(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME
inch, mvinch, mvwinch, winch — input a single-byte character and rendition from a window

SYNOPSIS
```
#include <curses.h>
chtype inch(void);
chtype mvinch(int y, int x);
chtype mvwinch(WINDOW *win, int y, int x);
chtype winch(WINDOW *win);
```

DESCRIPTION
These functions return the character and rendition, of type *chtype*, at the current or specified position in the current or specified window.

RETURN VALUE
Upon successful completion, the functions return the specified character and rendition. Otherwise, they return (**chtype**)ERR.

ERRORS
No errors are defined.

APPLICATION USAGE
These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the A_ prefix.

SEE ALSO
<curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The entry is rewritten for clarity. The argument list for the **inch()** function is explicitly declared as **void**.

(ENHANCED CURSES)

NAME

inchnstr, inchstr, mvinchnstr, mvinchstr, mvwinchnstr, mvwinchstr, winchnstr, winchstr — input an array of single-byte characters and renditions from a window

SYNOPSIS

```
#include <curses.h>
int inchnstr(chtype *chstr, int n);
int inchstr(chtype *chstr);
int mvinchnstr(int y, int x, chtype *chstr, int n);
int mvinchstr(int y, int x, chtype *chstr);
int mvwinchnstr(WINDOW *win, int y, int x, chtype *chstr, int n);
int mvwinchstr(WINDOW *win, int y, int x, chtype *chstr);
int winchnstr(WINDOW *win, chtype *chstr, int n);
int winchstr(WINDOW *win, chtype *chstr);
```

DESCRIPTION

These functions place characters and renditions from the current or specified window into the array pointed to by *chstr*, starting at the current or specified position and ending at the end of the line.

The **inchnstr()**, **mvinchnstr()**, **mvwinchnstr()** and **winchnstr()** functions store at most *n* elements from the current or specified window into the array pointed to by *chstr*.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

Reading a line that overflows the array pointed to by *chstr* with **inchstr()**, **mvinchstr()**, **mvwinchstr()** or **winchstr()** causes undefined results. The use of **inchnstr()**, **mvinchnstr()**, **mvwinchnstr()** or **winchnstr()**, respectively, is recommended.

SEE ALSO

inch(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

inet_addr(), inet_network(), inet_ntoa(), inet_makeaddr(), inet_lnaof(), inet_netof() - Internet address manipulation routines

SYNOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

in_addr_t inet_addr(const char *cp);

in_addr_t inet_network(const char *cp);

char *inet_ntoa(struct in_addr in);

struct in_addr inet_makeaddr(in_addr_t net, in_addr_t lna);

in_addr_t inet_lnaof(struct in_addr in);

in_addr_t inet_netof(struct in_addr in);
```

MULTITHREAD USAGE

Thread Safe:	Yes
Cancel Safe:	Yes
Async-cancel Safe:	No
Async-signal Safe:	No

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

DESCRIPTION

inet_addr()
inet_network() Interpret character strings representing numbers expressed in the Internet standard "dot" notation.

inet_addr() returns numbers suitable for use as Internet addresses.

inet_network() returns numbers suitable for use as Internet network numbers.

Return values can be assigned to a **struct in_addr** (defined in **/usr/include/netinet/in.h**) by using a technique similar to the following:

```
struct in_addr addr;
char *cp;

addr.s_addr = inet_addr(cp);
```

inet_ntoa() Take an Internet address and return an ASCII string representing the address in "**.**" (dot) notation.

inet_makeaddr() Take an Internet network number and a local network address and construct an Internet address from it.

inet_netof() Break apart Internet host addresses, returning the network number part.

inet_lnaof() Break apart Internet host addresses, returning the local network address part.

All Internet addresses are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine-format integer values. Bytes in HP-UX systems are ordered from left to right.

Internet Addresses:

Values specified using dot notation take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.

When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses, as in **128.net.host**.

When a two-part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right-most three bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as in **net.host**.

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as parts in dot notation can be decimal, octal, or hexadecimal, as specified in the C language (i.e., a leading 0x or 0X implies hexadecimal; a leading 0 implies octal; otherwise, the number is interpreted as decimal).

In a multithreaded application, **inet_ntoa()** uses thread-specific storage that is re-used in each call. The return value, the character string, should be unique for each thread and should be saved, if desired, before the thread makes the next **inet_ntoa()** call.

OBSOLESCENT INTERFACES

```
int inet_ntoa_r(struct in_addr in, char *buffer, int buflen);
```

The above reentrant interface has been moved from **libc** to **libd4r**. It is included to support existing applications and may be removed in a future release. New multithreaded applications should use the regular API (those without the **-r** suffix.)

The reentrant interface functions the same as the regular interface without the **-r** suffix. However, **inet_ntoa_r()** expects to be passed the address of a character buffer and will store the result at the supplied location. If the buffer is of insufficient length, –1 is returned. If the operation is successful, the length of the result string (not including the terminating null character) is returned.

RETURN VALUE

inet_addr() and **inet_network()** return –1 for malformed requests.

AUTHOR

inet routines were developed by the University of California, Berkeley.

SEE ALSO

gethostent(3N), getnetent(3N), hosts(4), networks(4).

NAME

initgroups() - initialize group access list

SYNOPSIS

```
#include <unistd.h>

int initgroups(const char *name, gid_t basegid);
```

DESCRIPTION

initgroups() reads the login group file, **/etc/logingroup**, and sets up the group access list for the user specified by *name*, using the *setgroups*(2) system call. If the value of *basegid* is zero or positive, it is automatically included in the groups list. Typically this value is given as the group number from the password file. If the login group file does not exist or is empty, *basegid* is the only member of the list.

APPLICATION USAGE

initgroups() is thread-safe. It is not async-cancel-safe. A cancellation point may occur when a thread is executing **initgroups()**.

RETURN VALUE

initgroups() returns **-1** if it was not invoked by a user with appropriate privileges.

WARNINGS

initgroups() uses the routines based on *getgrent*(3C). If the invoking program uses any of these routines, the group structure is overwritten by the call to **initgroups()**. Subsequent calls to **initgroups()** with the same *name* parameter override the actions of previous calls.

On many systems, no one seems to keep **/etc/logingroup** up to date.

initgroups() uses the Dynamic Name Service Switch. (See *nsswitch.conf*(4).) An application that uses this interface cannot be fully archive bound.

NETWORKING FEATURES

NFS

If **/etc/logingroup** is linked to **/etc/group**, **initgroups()** tries to use the Network Information Service (NIS) for entries beginning with a plus sign (**+**). If group membership for *name* is managed by NIS, and no NIS server is able to respond, a call to **initgroups()** does not return until a server does respond. This causes commands such as *login*(1) and *su*(1) to wait indefinitely.

See *group*(4) for proper syntax and operation.

AUTHOR

initgroups() was developed by the University of California, Berkeley.

FILES

/etc/logingroup login group file

SEE ALSO

login(1), su(1), getgroups(2), setgroups(2), group(4).

NAME

initscr, newterm — screen initialisation functions

SYNOPSIS

```
#include <curses.h>

WINDOW *initscr(void);

SCREEN *newterm(char *type, FILE *outfile, FILE *infile);
```

DESCRIPTION

The **initscr()** function determines the terminal type and initialises all implementation data structures. The **TERM** environment variable specifies the terminal type. The **initscr()** function also causes the first refresh operation to clear the screen. If errors occur, **initscr()** writes an appropriate error message to standard error and exits. The only functions that can be called before **initscr()** or **newterm()** are **filter()**, **ripoffline()**, **slk_init()**, **use_env()** and the functions whose prototypes are defined in **<term.h>**. Portable applications must not call **initscr()** twice.

The **newterm()** function can be called as many times as desired to attach a terminal device. The *type* argument points to a string specifying the terminal type, except that if *type* is a null pointer, the **TERM** environment variable is used. The *outfile* and *infile* arguments are file pointers for output to the terminal and input from the terminal, respectively. It is unspecified whether Curses modifies the buffering mode of these file pointers. The **newterm()** function should be called once for each terminal.

The **initscr()** function is equivalent to:

```
newterm(getenv("TERM"), stdout, stdin);
return stdscr;
```

If the current disposition for the signals SIGINT, SIGQUIT or SIGTSTP is SIGDFL, then **initscr()** may also install a handler for the signal, which may remain in effect for the life of the process or until the process changes the disposition of the signal.

The **initscr()** and **newterm()** functions initialise the *cur_term* external variable.

RETURN VALUE

Upon successful completion, **initscr()** returns a pointer to *stdscr* . Otherwise, it does not return.

Upon successful completion, **newterm()** returns a pointer to the specified terminal. Otherwise, it returns a null pointer.

ERRORS

No errors are defined.

APPLICATION USAGE

A program that outputs to more than one terminal should use **newterm()** for each terminal instead of **initscr()**. A program that needs an indication of error conditions, so it can continue to run in a line-oriented mode if the terminal cannot support a screen-oriented program, would also use this function.

Applications should perform any required handling of the SIGINT, SIGQUIT or SIGTSTP signals before calling **initscr()**.

SEE ALSO

Selecting a Terminal in terminfo(4), delscreen(3X), doupdate(3X), del_curterm(3X), filter(3X), slk_attroff(3X), use_env(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4

The **newterm()** function is merged with this entry. In previous issues, it appeared in an entry of its own.

The entry is rewritten for clarity. The argument list for the **initscr()** function is explicitly declared as **void**.

(ENHANCED CURSES)

NAME

innstr, instr, mvinnstr, mvinstr, mvwinnstr, mvwinstr, winnstr, winstr — input a multi-byte character string from a window

SYNOPSIS

```
#include <curses.h>
int innstr(char *str, int n);
int instr(char *str);
int mvinnstr(int y, int x, char *str, int n);
int mvinstr(int y, int x, char *str);
int mvwinnstr(WINDOW *win, int y, int x, char *str, int n);
int mvwinstr(WINDOW *win, int y, int x, char *str);
int winnstr(WINDOW *win, char *str, int n);
int winstr(WINDOW *win, char *str);
```

DESCRIPTION

These functions place a string of characters from the current or specified window into the array pointed to by *str*, starting at the current or specified position and ending at the end of the line.

The **innstr()**, **mvinnstr()**, **mvwinnstr()** and **winnstr()** functions store at most *n* bytes in the string pointed to by *str*.

The **innstr()**, **mvinnstr()**, **mvwinnstr()** and **winnstr()** functions will only store the entire multi-byte sequence associated with a character. If the array is large enough to contain at least one character the array is filled with complete characters. If the array is not large enough to contain any complete characters, the function fails.

RETURN VALUE

Upon successful completion, **instr()**, **mvinstr()**, **mvwinstr()** and **winstr()** return OK.

Upon successful completion, **innstr()**, **mvinnstr()**, **mvwinnstr()** and **winnstr()** return the number of characters actually read into the string.

Otherwise, all these functions return ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

Since multi-byte characters may be processed, there might not be a one-to-one correspondence between the number of column positions on the screen and the number of bytes returned.

These functions do not return rendition information.

Reading a line that overflows the array pointed to by *str* with **instr()**, **mvinstr()**, **mvwinstr()** or **winstr()** causes undefined results. The use of **innstr()**, **mvinnstr()**, **mvwinnstr()** or **winnstr()**, respectively, is recommended.

SEE ALSO

<curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

(ENHANCED CURSES)

NAME

innwstr, inwstr, mvinnwstr, mvinwstr, mvwinnwstr, mvwinwstr, winnwstr, winwstr — input a string of wide characters from a window

SYNOPSIS

```
#include <curses.h>
int innwstr(wchar_t *wstr, int n);
int inwstr(wchar_t *wstr);
int mvinnwstr(int y, int x, wchar_t *wstr, int n);
int mvinwstr(int y, int x, wchar_t *wstr);
int mvwinnwstr(WINDOW *win, int y, int x, wchar_t *wstr, int n);
int mvwinwstr(WINDOW *win, int y, int x, wchar_t *wstr);
int winnwstr(WINDOW *win, wchar_t *wstr, int n);
int winwstr(WINDOW *win, wchar_t *wstr);
```

DESCRIPTION

These functions place a string of **wchar_t** characters from the current or specified window into the array pointed to by *wstr* starting at the current or specified cursor position and ending at the end of the line.

These functions will only store the entire wide character sequence associated with a spacing complex character. If the array is large enough to contain at least one complete spacing complex character, the array is filled with complete characters. If the array is not large enough to contain any complete characters this is an error.

The **innwstr()**, **mvinnwstr()**, **mvwinnwstr()** and **winnwstr()** functions store at most *n* characters in the array pointed to by *wstr*.

RETURN VALUE

Upon successful completion, **inwstr()**, **mvinwstr()**, **mvwinwstr()** and **winwstr()** return OK.

Upon successful completion, **innwstr()**, **mvinnwstr()**, **mvwinnwstr()** and **winnwstr()** return the number of characters actually read into the string.

Otherwise, all these functions return ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

Reading a line that overflows the array pointed to by *wstr* with **inwstr()**, **mvinwstr()**, **mvwinwstr()** or **winwstr()** causes undefined results. The use of **innwstr()**, **mvinnwstr()**, **mvwinnwstr()** or **winnwstr()**, respectively, is recommended.

These functions do not return rendition information.

SEE ALSO

<curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

ins_nwstr, ins_wstr, mvins_nwstr, mvins_wstr, mvwins_nwstr, mvwins_wstr, wins_nwstr, wins_wstr — insert a wide-character string into a window

SYNOPSIS

```
#include <curses.h>
int ins_nwstr(const wchar_t *wstr, int n);
int ins_wstr(const wchar_t *wstr);
int mvins_nwstr(int y, int x, const wchar_t *wstr, int n);
int mvins_wstr(int y, int x, const wchar_t *wstr);
int mvwins_nwstr(WINDOW *win, int y, int x, const wchar_t *wstr, int n);
int mvwins_wstr(WINDOW *win, int y, int x, const wchar_t *wstr);
int wins_nwstr(WINDOW *win, const wchar_t *wstr, int n);
int wins_wstr(WINDOW *win, const wchar_t *wstr);
```

DESCRIPTION

These functions insert a **wchar_t** character string (as many **wchar_t** characters as will fit on the line) in the current or specified window immediately before the current or specified position.

Any non-spacing characters in the string are associated with the first spacing character in the string that precedes the non-spacing characters. If the first character in the string is a non-spacing character, these functions will fail.

These functions do not perform wrapping. These functions do not advance the cursor position. These functions perform special-character processing.

The **ins_nwstr()**, **mvins_nwstr()**, **mvwins_nwstr()** and **wins_nwstr()** functions insert at most *n* **wchar_t** characters. If *n* is less than 1, then the entire string is inserted.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

SEE ALSO

<curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

(ENHANCED CURSES)

NAME

ins_wch, mvins_wch, mvwins_wch, wins_wch — insert a complex character and rendition into a window

SYNOPSIS

```
#include <curses.h>
int ins_wch(const cchar_t *wch);
int wins_wch(WINDOW *win, const cchar_t *wch);
int mvins_wch(int y, int x, const cchar_t *wch);
int mvwins_wch(WINDOW *win, int y, int x, const cchar_t *wch);
```

DESCRIPTION

These functions insert the complex character *wch* with its rendition in the current or specified window at the current or specified cursor position.

These functions do not perform wrapping. These functions do not advance the cursor position. These functions perform special-character processing, with the exception that if a newline is inserted into the last line of a window and scrolling is not enabled, the behaviour is unspecified.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

For non-spacing characters, **add_wch()** can be used to add the non-spacing characters to a spacing complex character already in the window.

SEE ALSO

add_wch(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME
insch, mvinsch, mvwinsch, winsch — insert a single-byte character and rendition into a window

SYNOPSIS
```
#include <curses.h>
int insch(chtype ch);
int mvinsch(int y, int x, chtype ch);
int mvwinsch(WINDOW *win, int y, int x, chtype ch);
int winsch(WINDOW *win, chtype ch);
```

DESCRIPTION
These functions insert the character and rendition from *ch* into the current or specified window at the current or specified position.

These functions do not perform wrapping. These functions do not advance the cursor position. These functions perform special-character processing, with the exception that if a newline is inserted into the last line of a window and scrolling is not enabled, the behaviour is unspecified.

RETURN VALUE
Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS
No errors are defined.

APPLICATION USAGE
These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the A_ prefix.

SEE ALSO
ins_wch(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The entry is rewritten for clarity.

(ENHANCED CURSES)

NAME

insdelln, winsdelln — delete or insert lines into a window

SYNOPSIS

```
#include <curses.h>
int insdelln(int n);
int winsdelln(WINDOW *win, int n);
```

DESCRIPTION

The **insdelln()** and **winsdelln()** functions perform the following actions:

- If *n* is positive, these functions insert *n* lines into the current or specified window before the current line. The *n* last lines are no longer displayed.
- If *n* is negative, these functions delete *n* lines from the current or specified window starting with the current line, and move the remaining lines toward the cursor. The last *n* lines are cleared.

The current cursor position remains the same.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

SEE ALSO

deleteln(3X), insertln(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME
insertln, winsertln — insert lines into a window

SYNOPSIS
```
#include <curses.h>
int insertln(void);
int winsertln(WINDOW *win);
```

DESCRIPTION
The **insertln()** and **winsertln()** functions insert a blank line before the current line in the current or specified window. The bottom line is no longer displayed. The cursor position does not change.

RETURN VALUE
Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS
No errors are defined.

SEE ALSO
insdelln(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The entry is rewritten for clarity. The argument list for the **insertln()** function is explicitly declared as **void**.

(ENHANCED CURSES)

NAME

insnstr, insstr, mvinsnstr, mvinsstr, mvwinsnstr, mvwinsstr, winsnstr, winsstr — insert a multi-byte character string into a window

SYNOPSIS

```
#include <curses.h>
int insnstr(const char *str, int n);
int insstr(const char *str);
int mvinsnstr(int y, int x, const char *str, int n);
int mvinsstr(int y, int x, const char *str);
int mvwinsnstr(WINDOW *win, int y, int x, const char *str, int n);
int mvwinsstr(WINDOW *win, int y, int x, const char *str);
int winsnstr(WINDOW *win, const char *str, int n);
int winsstr(WINDOW *win, const char *str);
```

DESCRIPTION

These functions insert a character string (as many characters as will fit on the line) before the current or specified position in the current or specified window.

These functions do not advance the cursor position. These functions perform special-character processing. The **innstr()** and **innwstr()** functions perform wrapping. The **instr()** and **inswstr()** functions do not perform wrapping.

The **insnstr()**, **mvinsnstr()**, **mvwinsnstr()** and **winsnstr()** functions insert at most *n* bytes. If *n* is less than 1, the entire string is inserted.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

Since the string may contain multi-byte characters, there might not be a one-to-one correspondence between the number of column positions occupied by the characters and the number of bytes in the string.

SEE ALSO

<curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

insque(), remque() - insert or remove an element in a queue

SYNOPSIS

```
#include <search.h>
void insque(void *element, void *pred);
void remque(void *element);
```

DESCRIPTION

The **insque()** and **remque()** functions manipulate queues built from doubly-linked lists. An application using these functions must define a structure in which the first two members of the structure are pointers to the same type of structure. Any additional members of the structure are application specific. The first two members of the structure are used for forward and backward pointers. The names of the structure and of the pointers are not subject to any restrictions.

The **insque()** function inserts the object pointed to by the *element* argument into a queue immediately after the object pointed to by the *pred* argument.

The **remque()** function removes the object pointed to by the *element* argument from a queue.

APPLICATION USAGE

insque() and **remque()** are thread-safe and async-cancel-safe.

AUTHOR

insque() and **remque()** were developed by HP and the University of California, Berkeley.

STANDARDS COMPLIANCE

insque(): XPG4.2

remque(): XPG4.2

NAME
intrflush — enable or disable flush on interrupt

SYNOPSIS
```
#include <curses.h>
int intrflush(WINDOW *win, bool bf);
```

DESCRIPTION
The **intrflush()** function specifies whether pressing an interrupt key (interrupt, suspend or quit) will flush the input buffer associated with the current screen. If *bf* is TRUE, pressing an interrupt key will flush this input buffer. If *bf* is FALSE, pressing an interrupt key will not flush this input buffer. The default for the option is inherited from the display driver settings. The *win* argument must refer to a window associated with the current screen.

RETURN VALUE
Upon successful completion, **intrflush()** returns OK. Otherwise, it returns ERR.

ERRORS
No errors are defined.

APPLICATION USAGE
The same effect is achieved outside Curses using the NOFLSH local mode flag specified in the *X/Open System Interface Definitions, Issue 4, Version 2* specification (*General Terminal Interface*).

SEE ALSO
Input Processing in curses_intro(3X), <curses.h>, *X/Open System Interface Definitions, Issue 4, Version 2* specification, Section 9.2, *Parameters That Can Be Set*.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The entry is rewritten for clarity.

(Series 800 Only)

NAME

io_eol_ctl( ) - set up read termination character on special file (OBSOLETED AT 10.30)

SYNOPSIS

```
#include <dvio.h>
int io_eol_ctl(int eid, int flag, int match);
```

DESCRIPTION

io_eol_ctl() specifies a character to be used in terminating a read operation from the specified file (entity identifier).

eid is an entity identifier of an open HP-IB raw bus, Centronics-compatible parallel, or GPIO device file obtained from an **open()**, **dup()**, **fcntl()**, or **creat()** call. *flag* is an integer that enables or disables character-match termination. A non-zero value enables character-match termination, while a zero value disables it. *match* is an integer containing the numerical equivalent of the termination character. *match* is ignored if *flag* is zero. When in 8-bit mode, the lower 8 bits of *match* are used as the termination character. In 16-bit mode, the lower 16 bits are used.

Upon opening a file, the default condition is character-match termination disabled. When enabled, the character specified by *match* is checked for during read operations. The read is terminated upon receipt of this character, or upon any of the other termination conditions normally in effect for this file. Examples of other conditions are satisfying the specified byte count, and receiving a character when the EOI line is asserted (HP-IB). When the read is terminated by a *match* character, this character is the last character returned in the buffer.

Entity identifiers for the same device file obtained by separate **open()** calls have their own termination characters associated with them. Entity identifiers for the same device file inherited by a **fork()** call share the same termination character. In the latter case, if one process changes the termination character, the new termination character is in effect for all such entity identifiers.

RETURN VALUE

io_eol_ctl() returns 0 (zero) if successful, or –1 if an error was encountered.

ERRORS

io_eol_ctl() fails under the following circumstances, and sets **errno** (see *errno*(2)) to the value indicated:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to a channel device file.

AUTHOR

io_eol_ctl() was developed by HP.

SEE ALSO

dup(2), creat(2), fcntl(2), open(2), io_width_ctl(3I).

(Series 800 Only)

NAME

io_get_term_reason() - determine how last read terminated (OBSOLETED AT 10.30)

SYNOPSIS

```
#include <dvio.h>
int io_get_term_reason(int eid);
```

DESCRIPTION

io_get_term_reason() returns the termination reason for the last read made on this entity id. *eid* is an entity identifier of an open HP-IB raw bus, Centronics-compatible parallel interface, or GPIO device file obtained from an **open()**, **dup()**, **fcntl()**, or **creat()** call.

All entity identifiers descending from an **open()** request (such as from **dup()** or **fork())** set this status. For example, if the calling process had opened this entity identifier and later forked, the status returned would be from the last read done by either the calling process or its child.

RETURN VALUE

io_get_term_reason() returns a value indicating how the last read on the specified entity identifier was terminated. This value is interpreted as follows (note that combinations are possible):

Value	Description
–1	An error was encountered while making this function request.
0	Last read encountered some abnormal termination reason not covered by any of the other reasons.
1	Last read terminated by reading the number of bytes requested.
2	Last read terminated by detecting the specified termination character.
4	Last read terminated by detecting some device-imposed termination condition. Examples are: EOI for HP-IB, PSTS line on GPIO, or some other end-of-record condition, such as the physical end-of-record mark on a 9-track tape.

ERRORS

io_get_term_reason() fails under the following circumstances, and sets **errno** (see *errno*(2)) to the value indicated:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to a channel device file.

AUTHOR

io_get_term_reason() was developed by HP.

SEE ALSO

dup(2), creat(2), fcntl(2), open(2), read(2), io_eol_ctl(3I).

(Series 800 Only)

NAME

io_interrupt_ctl() - enable/disable interrupts for the associated eid (OBSOLETED AT 10.30)

SYNOPSIS

```
#include <dvio.h>
int io_interrupt_ctl(int eid, int enable_flag);
```

DESCRIPTION

eid is the entity identifier of an open HP-IB raw bus, Centronics-compatible parallel, or GPIO device file obtained from an **open()**, **dup()**, **fcntl()**, or **creat()** call. *flag* is an integer which enables or disables interrupts for the associated *eid*. A non-zero value enables interrupts.

Interrupts can be disabled or enabled as desired. When an interrupt occurs for a given *eid* the interrupts associated with this *eid* are automatically disabled from recurring. To re-enable interrupts for this *eid*, use **io_interrupt_ctl()**.

RETURN VALUE

io_interrupt_ctl() returns 0 (zero) if successful, or –1 if an error was encountered.

ERRORS

io_interrupt_ctl() fails under the following situations, and sets **errno** (see *errno*(2)) to the value indicated:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to a device that supports interrupts.
[EINVAL]	No interrupt conditions were specified for this *eid*.

AUTHOR

io_interrupt_ctl() was developed by HP.

SEE ALSO

dup(2), creat(2), fcntl(2), open(2), io_on_interrupt(3I).

(Series 800 Only)

NAME

io_lock, io_unlock - lock and unlock an interface (OBSOLETED AT 10.30)

SYNOPSIS

```
#include <dvio.h>
int io_lock(int eid);
int io_unlock(int eid);
```

DESCRIPTION

iolock() attempts to lock the interface associated with an entity identifier for the requesting process. Locking an interface gives exclusive use of the interface associated with the *eid* to the requesting process, thus avoiding unintended interference from other processes during a series of separate I/O requests. All locks for a process are removed when the process closes the file or terminates.

eid is an entity identifier of an open HP-IB, Centronics-compatible parallel, or GPIO device file, obtained from an **open()**, **dup()**, **fcntl()**, or **creat()** call (see *open*(2), *dup*(2), *fcntl*(2), and *creat*(2)).

Other processes that attempt to access or lock a locked interface either return an error or sleep until the interface becomes unlocked. The action taken is determined by the current setting of the **O_NDELAY** flag (see *open*(2). If the **O_NDELAY** flag is set, accesses to a locked interface fail and set **errno** to indicate the error. If the **O_NDELAY** flag is not set, accesses to a locked interface block until the interface is unlocked, the current timeout expires, or the request is interrupted by a signal.

A lock is associated with a process, not with an *eid*. Locking an interface with a particular *eid* does not prevent the process that owns the lock from accessing the interface through another *eid*. A lock associated with an *eid* is not inherited by a child process during a **fork()** (see *fork*(2)).

Nested locking is fully supported. If a process owns a locked interface and calls a generic subroutine that does a lock and unlock, the calling process does not lose its lock on the interface. Locking requests produced by a given process for an interface already locked by the same process increment the current lock count for that interface.

io_unlock() allows a process to remove a lock from the interface associated with the *eid*. A locked interface can be unlocked only by the process that directly owns the lock. When an unlock operation is applied to an *eid* that is currently multiply locked, the unlock operation decrements the current lock counter for that interface, and the interface remains locked until the count is reduced to zero.

RETURN VALUE

io_lock() and **io_unlock()** return the integer value of the current lock count if successful. A lock count greater than zero indicates that the interface is still locked. A lock count of zero indicates that the interface is no longer locked. A -1 indicates that an error has occurred.

ERRORS

io_lock() and **io_unlock()** fail in the following situations, and set **errno** (see *errno*(2)) to the value indicated:

[EACCES]	An attempt was made to lock an interface locked by another process with **O_NDELAY** set.
[EBADF]	*eid* does not refer to an open file.
[EINTR]	A signal was caught while attempting to perform the lock with **O_NDELAY** clear.
[EINVAL]	an attempt was made to unlock when the interface is not locked.
[ETIMEDOUT]	A timeout occurred while attempting to perform the lock with O_NDELAY clear.
[ENOTTY]	*eid* does not refer to a channel device file.
[EPERM]	An attempt was made to unlock when lock is not owned by this user.

WARNINGS

io_lock() provides a mandatory lock enforced by the system, and should not be used with any interface supporting a system disk or swap device.

Processes that lock HP-IB or GPIO interfaces should clear all locks before exiting. The driver attempts to clear them if the process terminates unexpectedly; however, a lock might be left outstanding if the locker dies after creating new file descriptors (via **fork()** or **dup()**) that refer to the same device file.

Ensuring that all open file descriptors on a given interface are closed remedies the situation.

AUTHOR

`io_lock()` and **`io_unlock()`** were developed by HP.

SEE ALSO

io_timeout_ctl(3I), open(2).

(Series 800 Only)

NAME

io_on_interrupt() - device interrupt (fault) control (OBSOLETED AT 10.30)

SYNOPSIS

```
#include <dvio.h>

int (*io_on_interrupt(
     int eid,
     struct interrupt_struct *causevec,
     int (*handler)(int, struct interrupt_struct *)
))(int, struct interrupt_struct *);
```

DESCRIPTION

eid is an entity identifier of an open HP-IB raw bus, Centronics-compatible parallel interface, or GPIO device file, obtained from an **open()**, **dup()**, **fcntl()**, or **creat()** call.

causevec is a pointer to a structure of the form:

```
struct interrupt_struct {
    integer     cause;
    integer     mask;
};
```

The **interrupt_struct** structure is defined in the file **dvio.h**.

cause is a bit vector specifying which of the interrupt or fault events can cause the handler routine to be invoked. The interrupt causes are often specific to the type of interface being considered. Also, certain exception (error) conditions can be handled using the **io_on_interrupt()** capability. Specifying a zero valued *cause* vector effectively turns off the interrupt for that *eid*.

The *mask* parameter is used when an HP-IB parallel poll interrupt is being defined. *mask* is an integer that specifies which parallel poll response lines are of interest. The value of *mask* is viewed as an 8-bit binary number where the least significant bit corresponds to line DIO1; the most significant bit to line DIO8. For example, to activate an interrupt handler when a response occurs on lines 2 or 6, the correct binary number is 00100010. Thus a hexadecimal value of 22 is the correct argument value for *mask*.

When an enabled interrupt condition on the specified *eid* occurs, the receiving process executes the interrupt-handler function pointed to by *handler*. The entity identifier *eid* and the interrupt condition *cause* are returned as the first and second parameters, respectively.

When an interrupt that is to be caught occurs during a **read()**, **write()**, **open()**, or **ioctl()** system call on a slow device such as a terminal (but not a file), during a **pause()** system call, a **sigpause()** system call, or a **wait()** system call that does not return immediately due to the existence of a previously stopped or zombie process, the interrupt handling function is executed and the interrupted system call returns –1 to the calling process with **errno** set to EINTR.

Interrupt *handlers* are not inherited across a **fork()**. *eid*s for the same device file produced by **dup()** share the same *handler*.

An interrupt for a given *eid* is implicitly disabled after the occurrence of the event. The interrupt condition can be re-enabled by using **io_interrupt_ctl()** (see *io_interrupt_ctl*(3I)).

When an event specified by *cause* occurs, the receiving process executes the interrupt *handler* function pointed to by *handler*. When the *handler* returns, the user process resumes at the execution point where the event occurred.

Two parameters are passed to *handler*: the *eid* associated with the event, and a pointer to a **causevec** structure. The cause of the interrupt can be determined by the value returned in the *cause* field of the **causevec** structure (more than 1 bit can be set, indicating that more than 1 interrupting condition has occurred). If the interrupt *handler* was invoked due to a parallel poll interrupt, the *mask* field of the **causevec** structure contains the parallel poll response byte.

HP-IB Interrupts

This section describes interrupt causes specific to an HP-IB device. For an HP-IB device, the cause is a bit vector which is used as follows. To enable a given event, the appropriate bit (in *cause*), shown below, must be set to **1**:

SRQ SRQ and active controller

TLK	Talker addressed
LTN	Listener addressed
TCT	Controller in charge
IFC	IFC has been asserted
REN	Remote enable
DCL	Device clear
GET	Group execution trigger
PPOLL	Parallel poll

GPIO Interrupts

This section describes interrupt causes specific to a GPIO device. For a GPIO device, *cause* is a bit vector which is used as follows. To enable a given event, the appropriate bit (in *cause*), shown below, must be set to **1**:

EIR	External interrupt
SIE0	Status line 0
SIE1	Status line 1

Parallel Interrupts

This section describes interrupt causes specific to a Centronics-compatible parallel device. For a Centronics-compatible parallel device, *cause* is a bit vector which is used as follows. To enable a given event, the appropriate bit (in *cause*), shown below, must be set to **1**:

NERROR	Nerror interrupt
SELECT	Select interrupt
PE	Paper error interrupt

RETURN VALUE

io_on_interrupt() returns a pointer to the previous *handler* if the new *handler* is successfully installed; otherwise it returns a –1 and sets **errno** to indicate the error.

ERRORS

io_on_interrupt() fails for any of the following reasons and sets **errno** to the value indicated:

[EACCES]	The interface associated with this *eid* is locked by another process and **O_NDELAY** is set for this *eid* (see *io_lock*(3I)).
[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to a GPIO, Centronics-compatible parallel, or a raw HP-IB device file.
[EFAULT]	*handler* points to an illegal address. The reliable detection of this error is implementation dependent.
[EFAULT]	*causevec* points to an illegal address. The reliable detection of this error is implementation dependent.

DEPENDENCIES

For the HP 27114 AFI interface, only the EIR interrupt is available.

AUTHOR

io_on_interrupt() was developed by HP.

SEE ALSO

dup(2), creat(2), fcntl(2), open(2), pause(2), sigpause(2), io_interrupt_ctl(3I).

(Series 800 Only)

NAME

io_reset() - reset an I/O interface (OBSOLETED AT 10.30)

SYNOPSIS

```
#include <dvio.h>
int io_reset(int eid);
```

DESCRIPTION

io_reset() resets the interface associated with the device file that was opened. It also pulses the peripheral reset line on the GPIO interface, or the IFC line on the HP-IB. *eid* is an entity identifier of an open HP-IB, Centronics-compatible parallel interface, or GPIO device file obtained from an **open()**, **dup()**, **fcntl()**, or **creat()** call.

io_reset() also causes an interface to go through its self-test, and returns a failure indication if the interface fails its test.

RETURN VALUE

io_reset() returns 0 (zero) if successful, or –1 if an error was encountered.

ERRORS

io_reset() fails under the following circumstances, and sets **errno** (see *errno*(2)) to the value indicated:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to a channel device file.
[EIO]	Interface could not be reset or failed self-test.
[EACCES]	The interface associated with this *eid* is locked by another process and **O_NDELAY** is set for this *eid* (see *io_lock*(3I)).

AUTHOR

io_reset() was developed by HP.

(Series 800 Only)

NAME

io_speed_ctl() - inform system of required transfer speed (OBSOLETED AT 10.30)

SYNOPSIS

```
#include <dvio.h>
int io_speed_ctl(int eid, int speed);
```

DESCRIPTION

io_speed_ctl() selects the data transfer speed for a data path used for a particular interface. The transfer method (i.e., DMA or fast-handshake) chosen by the system is determined by the speed requirements.

eid is an entity identifier of an open HP-IB raw bus, Centronics-compatible parallel, or GPIO device file obtained from an **open()**, **dup()**, **fcntl()**, or **creat()** call. *speed* is an integer specifying the data transfer speed in Kbytes per second (one Kbyte equals 1024 bytes).

RETURN VALUE

io_speed_ctl() returns 0 if successful, and –1 otherwise.

ERRORS

io_speed_ctl() fails under the following condition, and sets **errno** to the value indicated:

[ENOTTY] *eid* does not refer to channel device file.

[EBADF] *eid* does not refer to an open file.

DEPENDENCIES

DMA is the only supported transfer method.

AUTHOR

io_speed_ctl() was developed by HP.

(Series 800 Only)

NAME

io_timeout_ctl( ) - establish a time limit for I/O operations (OBSOLETED AT 10.30)

SYNOPSIS

```
#include <dvio.h>

int io_timeout_ctl(int eid, long time);
```

DESCRIPTION

io_timeout_ctl() assigns a timeout value to the specified *eid* (entity identifier). *eid* is an entity identifier of an open HP-IB raw bus, auto-addressed, Centronics-compatible parallel, or GPIO device file obtained from an **open()**, **dup()**, **fcntl()**, or **creat()** call. *time* is a long integer value specifying the length of the timeout in microseconds. A value of 0 for *time* specifies no timeout (infinity).

This timeout applies to future read and write requests on this *eid*. If a read or write request does not complete within the specified time limit, the request is aborted and returns an error indication. If an operation is aborted due to a timeout, **errno** is set to ETIMEDOUT.

Although the timeout value is specified in microseconds, the resolution of the timeout is system-dependent. For example, a particular system might have a resolution of 10 milliseconds, in which case the specified timeout value is rounded up to the next 10 msec boundary. A timeout value of zero means that the system never causes a timeout. When a file is opened, a zero timeout value is assigned by default.

Entity identifiers for the same device file obtained by separate **open()** calls have their own timeout values associated with them. Entity identifiers for the same device file obtained by **dup()** or inherited by a **fork()** call share the same timeout value. In the latter case, if one process changes the timeout, the new timeout is in effect for all such *eid*s.

RETURN VALUE

io_timeout_ctl() returns 0 (zero) if successful, or –1 if an error was encountered.

ERRORS

io_timeout_ctl() fails under the following circumstances, and sets **errno** (see *errno*(2)) to the value indicated:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to a channel device file.

AUTHOR

io_timeout_ctl() was developed by HP.

(Series 800 Only)

NAME

io_width_ctl - set width of data path (OBSOLETED AT 10.30)

SYNOPSIS

```
#include <dvio.h>
int io_width_ctl(int eid, int width);
```

DESCRIPTION

io_width_ctl() enables you to select the width of the data path to be used for a particular interface. *eid* is an entity identifier of an open HP-IB, Centronics-compatible parallel interface, or GPIO device file obtained from an **open()**, **dup()**, **fcntl()**, or **creat()** call. *width* is an integer specifying the width of the data path in bits.

An error is given if an invalid width is specified. Specifying a width with this function sets the width for all users of the device file associated with the given entity id. When first opened, the default width is 8 bits.

For the GPIO interface only widths of 8 and 16 bits are currently supported. For the HP-IB and Centronics-compatible parallel interfaces, only a width of 8 bits is supported.

RETURN VALUE

io_width_ctl() returns 0 if successful, and –1 if an error was encountered.

ERRORS

io_width_ctl() fails under the following circumstances and sets **errno** (see *errno*(2)) to the value indicated:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to a channel device file.

[EINVAL] the specified *width* is not supported on this device file.

AUTHOR

io_width_ctl() was developed by HP.

(ENHANCED CURSES)

NAME

is_linetouched, is_wintouched, touchline, untouchwin, wtouchln — window refresh control functions

SYNOPSIS

```
#include <curses.h>
bool is_linetouched(WINDOW *win, int line);
bool is_wintouched(WINDOW *win);
int touchline(WINDOW *win, int start, int count);
int untouchwin(WINDOW *win);
int wtouchln(WINDOW *win, int y, int n, int changed);
```

DESCRIPTION

The **touchline()** function only touches *count* lines, beginning with line *start*.

The **untouchwin()** function marks all lines in the window as unchanged since the last refresh operation.

Calling **wtouchln()**, if *changed* is 1, touches *n* lines in the specified window, starting at line *y*. If *changed* is 0, **wtouchln()** marks such lines as unchanged since the last refresh operation.

The **is_wintouched()** function determines whether the specified window is touched. The **is_linetouched()** function determines whether line *line* of the specified window is touched.

RETURN VALUE

The **is_linetouched()** and **is_wintouched()** functions return TRUE if any of the specified lines, or the specified window, respectively, has been touched since the last refresh operation. Otherwise, they return FALSE.

Upon successful completion, the other functions return OK. Otherwise, they return ERR. Exceptions to this are noted in the preceding function descriptions.

ERRORS

No errors are defined.

APPLICATION USAGE

Calling **touchwin()** or **touchline()** is sometimes necessary when using overlapping windows, since a change to one window affects the other window, but the records of which lines have been changed in the other window do not reflect the change.

SEE ALSO

Screens, Windows and Terminals in curs_intro, doupdate(3X), touchwin(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

isastream() - determine if a file descriptor refers to a STREAMS device or STREAMS-based pipe

SYNOPSIS

```
#include <stropts.h>
int isastream(int fd);
```

DESCRIPTION

The **isastream()** function tests whether an open file descriptor (*fd*) corresponds to a STREAMS device or STREAMS-based pipe.

RETURN VALUE

Upon successful completion, the **isastream()** function returns a value of 1 when the file descriptor of the open file specified by *fd* is a STREAMS device or STREAMS-based pipe, and 0 (zero) if it is not a stream, but is a valid open file descriptor. Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

ERRORS

If any of the following conditions occur, the **isastream()** function sets **errno** to the corresponding value:

[EBADF] The specified file descriptor does not refer to a valid open file.

SEE ALSO

fattach(3C), fdetach(3C), streamio(7).

STANDARDS COMPLIANCE

isastream(): SVID3

(ENHANCED CURSES)

NAME

isendwin — determine whether a screen has been refreshed

SYNOPSIS

```
#include <curses.h>
bool isendwin(void);
```

DESCRIPTION

The **isendwin()** function indicates whether the screen has been refreshed since the last call to **endwin()**.

RETURN VALUE

The **isendwin()** function returns TRUE if **endwin()** has been called without any subsequent refresh. Otherwise, it returns FALSE.

ERRORS

No errors are defined.

SEE ALSO

endwin(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

i

NAME
isfinite() - floating-point finiteness macro

SYNOPSIS
```
#include <math.h>
int isfinite(floating-type x);
```

DESCRIPTION
The **isfinite()** macro determines whether its argument has a finite value (zero, denormalized, or normalized, and not infinite or NaN). The macro can be used with either **double** or **float** arguments.

The ISO/ANSI C committee has approved the **isfinite()** macro for inclusion in the C9X draft standard. The **isfinite()** macro implements the **finite()** function recommended by the IEEE-754 standard for floating-point arithmetic.

To use the **isfinite()** macro, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE
The **isfinite()** macro returns a nonzero value if and only if its argument has a finite value.

ERRORS
No errors are defined.

EXAMPLE
Make sure a value is finite before continuing operations on it:

```
#include <math.h>
/*...*/
   float x;
   /*...*/
   if (isfinite(x))
      /*...*/
```

SEE ALSO
fpclassify(3M), isinf(3M), isnan(3M), isnormal(3M), signbit(3M), math(5).

NAME

isgreater() - floating-point comparison macro (>)

SYNOPSIS

```
#include <math.h>
int isgreater( floating-expr x, floating-expr y);
```

DESCRIPTION

The **isgreater()** macro determines whether its first argument is greater than its second argument. The value of **isgreater(***x*,*y***)** is always equal to **(***x***) > (***y***)**; however, unlike **(***x***) > (***y***)**, **isgreater(***x*,*y***)** does not raise the invalid exception when *x* and *y* are unordered.

The macro can be used with either **double** or **float** arguments.

The ISO/ANSI C committee has approved the **isgreater()** macro for inclusion in the C9X draft standard.

To use the **isgreater()** macro, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

The **isgreater()** macro returns the value of **(***x***) > (***y***)**.

ERRORS

No errors are defined.

SEE ALSO

isgreaterequal(3M), isless(3M), islessequal(3M), islessgreater(3M), isunordered(3M), math(5).

NAME

isgreaterequal( ) - floating-point comparison macro (>=)

SYNOPSIS

```
#include <math.h>
int isgreaterequal(floating-expr x, floating-expr y);
```

DESCRIPTION

The **isgreaterequal()** macro determines whether its first argument is greater than or equal to its second argument. The value of **isgreaterequal(***x***,***y***)** is always equal to **(***x***) >= (***y***)**; however, unlike **(***x***) >= (***y***)**, **isgreaterequal(***x***,***y***)** does not raise the invalid exception when *x* and *y* are unordered.

The macro can be used with either **double** or **float** arguments.

The ISO/ANSI C committee has approved the **isgreaterequal()** macro for inclusion in the C9X draft standard.

To use the **isgreaterequal()** macro, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

The **isgreaterequal()** macro returns the value of **(***x***) >= (***y***)**.

ERRORS

No errors are defined.

SEE ALSO

isgreater(3M), isless(3M), islessequal(3M), islessgreater(3M), isunordered(3M), math(5).

NAME
isinf() - floating-point test for infinity

SYNOPSIS
```
#include <math.h>
int isinf(floating-type x);
```

DESCRIPTION
The **isinf()** macro determines whether its argument value is an infinity. The macro can be used with either **double** or **float** arguments.

The ISO/ANSI C committee has approved the **isinf()** macro for inclusion in the C9X draft standard.

To use this macro, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

The **isinf()** macro replaces the **isinf()** and **isinff()** functions, which are obsolete and are no longer supported.

RETURN VALUE
The **isinf()** macro returns a nonzero value if *x* is ±INFINITY. Otherwise it returns zero.

EXAMPLE
Take certain actions if **x** is infinity:

```
#include <math.h>
/*...*/
   float x;
   /*...*/
   if (isinf(x))
      /*...*/
```

ERRORS
No errors are defined.

SEE ALSO
fpclassify(3M), isfinite(3M), isnan(3M), isnormal(3M), signbit(3M), math(5).

NAME

isless( ) - floating-point comparison macro (<)

SYNOPSIS

```
#include <math.h>
int isless(floating-expr x, floating-expr y);
```

DESCRIPTION

The **isless()** macro determines whether its first argument is less than its second argument. The value of **isless(***x*,*y***)** is always equal to **(***x***) < (***y***)**; however, unlike **(***x***) < (***y***)**, **isless(***x*,*y***)** does not raise the invalid exception when *x* and *y* are unordered.

The macro can be used with either **double** or **float** arguments.

The ISO/ANSI C committee has approved the **isless()** macro for inclusion in the C9X draft standard.

To use the **isless()** macro, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

The **isless()** macro returns the value of **(***x***) < (***y***)**.

ERRORS

No errors are defined.

SEE ALSO

isgreater(3M), isgreaterequal(3M), islessequal(3M), islessgreater(3M), isunordered(3M), math(5).

NAME

islessequal() - floating-point comparison macro (<=)

SYNOPSIS

```
#include <math.h>
int islessequal( floating-expr x, floating-expr y);
```

DESCRIPTION

The **islessequal()** macro determines whether its first argument is less than or equal to its second argument. The value of **islessequal(** *x*,*y***)** is always equal to **(***x***) <= (***y***)**; however, unlike **(***x***) <= (***y***)**, **islessequal(** *x*,*y***)** does not raise the invalid exception when *x* and *y* are unordered.

The macro can be used with either **double** or **float** arguments.

The ISO/ANSI C committee has approved the **islessequal()** macro for inclusion in the C9X draft standard.

To use the **islessequal()** macro, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

The **islessequal()** macro returns the value of **(***x***) <= (***y***)**.

ERRORS

No errors are defined.

SEE ALSO

isgreater(3M), isgreaterequal(3M), isless(3M), islessgreater(3M), isunordered(3M), math(5).

NAME

islessgreater() - floating-point comparison macro (<>)

SYNOPSIS

```
#include <math.h>
int islessgreater(floating-expr x, floating-expr y);
```

DESCRIPTION

The **islessgreater()** macro determines whether its first argument is less than or greater than its second argument. The value of **islessgreater(***x*,*y***)** is always equal to **(***x***) < (***y***) || (***x***) > (***y***)**; however, **islessgreater(***x*,*y***)** does not raise the invalid exception when *x* and *y* are unordered (nor does it evaluate *x* and *y* twice).

The macro can be used with either **double** or **float** arguments.

The ISO/ANSI C committee has approved the **islessgreater()** macro for inclusion in the C9X draft standard.

To use the **islessgreater()** macro, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

The **islessgreater()** macro returns the value of **(***x***) < (***y***) || (***x***) > (***y***)**.

ERRORS

No errors are defined.

SEE ALSO

isgreater(3M), isgreaterequal(3M), isless(3M), islessequal(3M), isunordered(3M), math(5).

NAME
isnan() - floating-point test for NaN

SYNOPSIS
```
#include <math.h>
int isnan(floating-type x);
```

DESCRIPTION
The **isnan()** macro determines whether its argument value is a NaN. The macro can be used with either **double** or **float** arguments.

The ISO/ANSI C committee has approved the **isnan()** macro for inclusion in the forthcoming C9X draft standard. The **isnan()** macro implements the **isnan()** function recommended by the IEEE-754 standard for floating-point arithmetic.

To use this macro, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

The **isnan()** macro replaces the **isnan()** and **isnanf()** functions, which are obsolete and are no longer supported.

RETURN VALUE
The **isnan()** macro returns a nonzero value if and only if its argument has a NaN value.

ERRORS
No errors are defined.

EXAMPLE
Take certain actions if **x** is not a NaN:

```
#include <math.h>
/*...*/
   double x;
   /*...*/
   if (!isnan(x))
      /*...*/
```

SEE ALSO
fpclassify(3M), isfinite(3M), isinf(3M), isnormal(3M), signbit(3M), math(5).

STANDARDS CONFORMANCE
isnan(): SVID3, XPG4.2

NAME

isnormal() - floating-point test for normalized value

SYNOPSIS

```
#include <math.h>
int isnormal(floating-type x);
```

DESCRIPTION

The **isnormal()** macro determines whether its argument has a normalized value (neither zero, denormalized, infinite, nor NaN). The macro can be used with either **double** or **float** arguments, and classifies the argument based on its type.

The ISO/ANSI C committee has approved the **isnormal()** macro for inclusion in the C9X draft standard.

To use the **isnormal()** macro, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

The **isnormal()** macro returns a nonzero value if and only if its argument has a normalized value.

ERRORS

No errors are defined.

EXAMPLE

Make sure a value is normalized before continuing operations on it:

```
#include <math.h>
/*...*/
   float x;
   /*...*/
   if (isnormal(x))
      /*...*/
```

SEE ALSO

fpclassify(3M), isfinite(3M), isinf(3M), isnan(3M), signbit(3M), math(5).

NAME

isunordered() - floating-point comparison macro (unordered)

SYNOPSIS

```
#include <math.h>
int isunordered( floating-expr x, floating-expr y);
```

DESCRIPTION

The **isunordered()** macro determines whether its arguments are unordered. The arguments are unordered if at least one argument is a NaN.

The macro can be used with either **double** or **float** arguments.

The ISO/ANSI C committee has approved the **isunordered()** macro for inclusion in the C9X draft standard.

To use the **isunordered()** macro, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

The **isunordered()** macro returns 1 if its arguments are unordered (that is, if either argument is a NaN) and 0 otherwise.

ERRORS

No errors are defined.

SEE ALSO

isgreater(3M), isgreaterequal(3M), isless(3M), islessequal(3M), islessgreater(3M), math(5).

NAME

j0(), j1(), jn() - Bessel functions of the first kind

SYNOPSIS

```
#include <math.h>
double j0(double x);
double j1(double x);
double jn(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*.

To use these functions, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

If *x* is NaN, **j0()**, **j1()**, and **jn()** return NaN.

If the correct result after rounding would be smaller in magnitude than **MINDOUBLE**, **j0()**, **j1()**, and **jn()** return zero.

j

ERRORS

No errors are defined.

SEE ALSO

y0(3M), math(5), values(5).

M. Abramowitz and I. Stegun, *Handbook of Mathematical Functions* (New York: Dover Publications, 1972).

STANDARDS CONFORMANCE

j0(): SVID3, XPG4.2

j1(): SVID3, XPG4.2

jn(): SVID3, XPG4.2

NAME

keyname, key_name — get name of key

SYNOPSIS

```
#include <curses.h>
char *keyname(int c);
char *key_name(wchar_t c);
```

DESCRIPTION

The **keyname()** and **key_name()** functions generate a character string whose value describes the key *c*. The *c* argument of **keyname()** can be an 8-bit character or a key code. The *c* argument of **key_name()** must be a wide character.

The string has a format according to the first applicable row in the following table:

Input	Format of Returned String
Visible character	The same character
Control character	**^***X*
Meta-character (**keyname()** only)	**M-***X*
Key value in <**curses.h**> (**keyname()** only)	**KEY**_*name*
None of the above	**UNKNOWN KEY**

The meta-character notation shown above is used only if meta-characters are enabled.

RETURN VALUE

Upon successful completion, **keyname()** returns a pointer to a string as described above. Otherwise, it returns a null pointer.

ERRORS

No errors are defined.

APPLICATION USAGE

The return value of **keyname()** and **key_name()** may point to a static area which is overwritten by a subsequent call to either of these functions.

Applications normally process meta-characters without storing them into a window. If an application stores meta-characters in a window and tries to retrieve them as wide characters, **keyname()** cannot detect meta-characters, since wide characters do not support meta-characters.

SEE ALSO

meta(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME
keypad — enable/disable abbreviation of function keys

SYNOPSIS
```
#include <curses.h>
int keypad(WINDOW *win, bool bf);
```

DESCRIPTION
The **keypad()** function controls keypad translation. If *bf* is TRUE, keypad translation is turned on. If *bf* is FALSE, keypad translation is turned off. The initial state is FALSE.

This function affects the behaviour of any function that provides keyboard input.

If the terminal in use requires a command to enable it to transmit distinctive codes when a function key is pressed, then after keypad translation is first enabled, the implementation transmits this command to the terminal before an affected input function tries to read any characters from that terminal.

RETURN VALUE
Upon successful completion, **keypad()** returns OK. Otherwise, it returns ERR.

ERRORS
No errors are defined.

SEE ALSO
Keypad Processing in curs_intro, <curses.h>.

k **CHANGE HISTORY**
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The entry is rewritten for clarity.

NAME

l3tol(), ltol3() - convert between 3-byte integers and long integers

SYNOPSIS

```
#include <stdlib.h>
void l3tol(long int *lp, const char *cp, int n);
void ltol3(char *cp, const long int *lp, int n);
```

DESCRIPTION

l3tol() Convert 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() Perform the reverse conversion from long integers (*lp*) to three-byte integers (*cp*).

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

APPLICATION USAGE

l3tol() and **ltol3()** are thread-safe and async-cancel-safe.

WARNINGS

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

SEE ALSO

fs(4).

STANDARDS CONFORMANCE

l3tol(): XPG2

ltol3(): XPG2

NAME

lckpwdf(), ulckpwdf() - control access to /etc/passwd file

SYNOPSIS

```
#include <shadow.h>

int lckpwdf (void)

int ulckpwdf (void)
```

DESCRIPTION

The **lckpwdf()** and **ulckpwdf()** routines are used to coordinate modification access to the password file **/etc/passwd** and to the secure password entries. The lock file used by these two routines is **/etc/.pwd.lock**. A process first calls **lckpwdf()** to gain exclusive access rights for password modification. When modifications are complete, **ulckpwdf()** is called to release the lock on **/etc/.pwd.lock**. This mechanism prevents simultaneous modification of password files or entries.

APPLICATION USAGE

lckpwdf() and **ulckpwdf()** are thread-safe. These interfaces are not cancel-safe. A cancellation point may occur when a thread is executing these interfaces.

RETURN VALUE

The **lckpwdf()** routine returns zero upon successful completion. If the lock could not be obtained, it returns –1 and sets **errno** to indicate the error.

The **ulckpwdf()** routine returns zero upon successful completion. If the lock has already been released, **ulckpwdf()** returns –1 and sets **errno** to indicate the error.

FILES

/etc/passwd
/etc/.pwd.lock

SEE ALSO

getpwent(3C), passwd(4).

NAME

_ldecvt(), _ldfcvt(), _ldgcvt() - convert long-double floating-point number to string

SYNOPSIS

```
#include <stdlib.h>
char *_ldecvt(long_double value, int ndigit, int *decpt, int *sign);
char *_ldgcvt(long_double value, int ndigit, char *buf);
char *_ldfcvt(long_double value, int ndigit, int *decpt, int *sign);
```

Obsolescent Interfaces

```
int _ldecvt_r(
     long_double value,
     int ndigit,
     int *decpt,
     int *sign,
     char *buffer,
     int buflen);
int _ldfcvt_r(
     long_double value,
     int ndigit,
     int *decpt,
     int *sign,
     char *buffer,
     int buflen);
```

DESCRIPTION

_ldecvt() converts *value* to a null-terminated string of *ndigit* digits and returns a pointer to the string. The high-order digit is non-zero, unless the value is zero. The low-order digit is rounded. The position of the radix character relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). The radix character 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.

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

_ldgcvt() Convert the *value* to a null-terminated string in the array pointed to by *buf* and return *buf*. It produces *ndigit* significant digits in FORTRAN F-format if possible, or E-format otherwise. A minus sign, if required, and a radix character are included in the returned string. Trailing zeros are suppressed. The radix character is determined by the currently loaded NLS environment (see *setlocale*(3C)). If **setlocale()** has not been called successfully, the default NLS environment, "C" is used (see *lang*(5)). The default environment specifies a period (**.**) as the radix character.

Obsolescent Interfaces

_ldecvt_r(), **_ldfcvt_r()** convert long-double floating-point number to string.

APPLICATION USAGE

_ldecvt(), **_ldfcvt()** and **_ldgcvt()** are thread-safe. These interfaces are not async-cancel-safe.

RETURN VALUE

NaN is returned for Not-a-Number, and ±**INFINITY** is returned for Infinity.

WARNINGS

The values returned by **_ldecvt()** and **_ldfcvt()** point to data whose content is overwritten by subsequent calls to these interfaces by the same thread.

_ldecvt_r() and **_ldfcvt_r()** are obsolescent interfaces supported only for compatibility with existing DCE applications. New multithreaded applications should use **_ldecvt()** and **_ldfcvt()**.

EXTERNAL INFLUENCES

Locale

The **LC_NUMERIC** category determines the radix character.

International Code Set Support

Single-byte character code sets are supported.

AUTHOR

_ldecvt(), **_ldfcvt()**, and **_ldgcvt()** were developed by HP.

SEE ALSO

setlocale(3C), printf(3S), hpnls(5), lang(5).

NAME

ldexp() - load exponent of a floating-point number

SYNOPSIS

```
#include <math.h>
double ldexp(double x, int exp);
```

DESCRIPTION

The **ldexp()** function computes the quantity $x * 2^{exp}$.

To use this function, make sure your program includes <**math.h**>, and link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

Upon successful completion, the **ldexp()** function returns a **double** representing the value *x* multiplied by 2 raised to the power *exp*.

If *x* is NaN, **ldexp()** returns NaN.

If the correct value after rounding would be smaller in magnitude than **MINDOUBLE**, **ldexp()** returns zero.

If the correct value would overflow, **ldexp()** returns ±**HUGE_VAL** (according to the sign of *x*) and sets **errno** to [ERANGE].

ERRORS

If **ldexp()** fails, **errno** is set to one of the following values.

[ERANGE]	The correct value would overflow.

SEE ALSO

frexp(3M), modf(3M), scalb(3M), scalbn(3M), math(5), values(5).

STANDARDS CONFORMANCE

ldexp(): SVID3, XPG4.2, ANSI C

NAME

lgamma(), lgamma_r(), gamma(), signgam() - log gamma function

SYNOPSIS

```
#include <math.h>
double lgamma(double x);
double gamma(double x); (TO BE WITHDRAWN)
extern int signgam;
double lgamma_r(double x, int *sign);
```

DESCRIPTION

lgamma() and **gamma()** return **ln(|Γ(***x***)|)**, where Γ(*x*) is defined as the integral, as *t* goes from zero to infinity, of **exp(**−*t***)** times *t* to the power (*x*−1).

The sign of Γ(*x*) is returned in the external integer **signgam**. The argument *x* must not be zero or a negative integer. (**lgamma()** is defined over the reals excluding the non-positive integers.)

The following C program fragment can be used to calculate Γ(*x*):

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

where if *y* is greater than **LN_MAXDOUBLE**, as defined in the <**values.h**> header file, **exp()** returns a range error (see *exp*(3M)).

The log gamma function **lgamma()** is not reentrant because it uses the global variable **signgam**. The function **lgamma_r()** is a reentrant version of **lgamma()** that can be used in multi-threaded applications. The function **lgamma_r()** returns the sign of Γ(*x*) in its parameter list, through the pointer *sign*. The value pointed to by *sign* is +1 if Γ(*x*) is positive, -1 if it is negative.

The **gamma()** function is functionally equivalent to **lgamma()**, and in the XPG4.2 standard it is marked TO BE WITHDRAWN. A true **gamma()** function has been approved for inclusion in the C9X draft standard.

The ISO/ANSI C committee has approved the **lgamma()** function for inclusion in the C9X draft standard.

To use these functions, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

If *x* is a non-positive integer, **lgamma()** and **gamma()** return **HUGE_VAL**.

If the correct value would overflow, **lgamma()** and **gamma()** return **HUGE_VAL**.

If the correct value after rounding would be smaller in magnitude than **MINDOUBLE**, **lgamma()** and **gamma()** return zero.

If *x* is NaN, **lgamma()** and **gamma()** return NaN.

ERRORS

No errors are defined.

WARNINGS

lgamma() and **gamma()** are unsafe in multi-thread applications. **lgamma_r()** is MT-Safe and should be used instead.

SEE ALSO

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

STANDARDS CONFORMANCE

lgamma(): SVID3, XPG4.2

gamma(): SVID2, XPG4.2 (TO BE WITHDRAWN)

signgam: SVID3, XPG4.2

(ENHANCED CURSES)

NAME

LINES — number of lines on terminal screen

SYNOPSIS

```
#include <curses.h>
extern int LINES;
```

DESCRIPTION

The external variable *LINES* indicates the number of lines on the terminal screen.

SEE ALSO

initscr(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

llrint( ) - round to nearest long long function

SYNOPSIS

```
#include <math.h>
long long llrint(double x);
```

DESCRIPTION

llrint() rounds its argument to the nearest integral value, rounding according to the current rounding direction. If the rounded value is outside the range of **long long**, the numeric result is unspecified.

The ISO/ANSI C committee has approved the **llrint()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

The **llrint()** function returns the rounded **long long** value, using the current rounding direction.

ERRORS

No errors are defined.

SEE ALSO

ceil(3M), floor(3M), fabs(3M), fmod(3M), fegetround(3M), fesetround(3M), lrint(3M), lround(3M), llround(3M), rint(3M), round(3M), trunc(3M), math(5), fenv(5).

NAME

llround( ) - round to long long function

SYNOPSIS

```
#include <math.h>
long long llround(double x);
```

DESCRIPTION

llround() rounds its argument to the nearest integral value. An argument exactly halfway between two integers is rounded away from zero, regardless of the current rounding direction. Rounding away from zero also applies to the functions, **round** and **lround**.

If the rounded value is outside the range of **long long**, the numeric result is unspecified.

The ISO/ANSI C committee has approved the **llround()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

The **llround()** function returns the rounded **long long** value.

ERRORS

No errors are defined.

SEE ALSO

ceil(3M), floor(3M), fabs(3M), fmod(3M), fegetround(3M), fesetround(3M), lrint(3M), llrint(3M), lround(3M), rint(3M), round(3M), trunc(3M), math(5), fenv(5).

NAME

localeconv() - query the numeric formatting conventions of the current locale

SYNOPSIS

```
#include <locale.h>

struct lconv *localeconv(void);
```

DESCRIPTION

The **localeconv()** function sets the components of an object of type **struct lconv** (defined in **<locale.h>**) with values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the program's current locale (see *setlocale*(3C)).

The members of the structure with type **char *** are strings, any of which (except **decimal_point**) can point to "" (the empty string) to indicate that the value is not available in the current locale or is of zero length. The members with type **char** are nonnegative numbers, any of which can be **CHAR_MAX** (defined in **<limits.h>**) to indicate that the value is not available in the current locale. The members include the following:

char *decimal_point
: The decimal point character used to format nonmonetary quantities. This is the same value as that returned by a call to **nl_langinfo()** with **RADIXCHAR** as its argument (see *nl_langinfo*(3C)).

char *thousands_sep
: The character used to separate groups of digits to the left of the decimal point character in formatted nonmonetary quantities. This is the same value as that returned by a call to **nl_langinfo()** with **THOUSEP** as its argument (see *nl_langinfo*(3C)).

char *grouping
: A string where the numeric value of each byte indicates the size of each group of digits in formatted nonmonetary quantities.

char *int_curr_symbol
: The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in *ISO 4217 Codes for the Representation of Currency and Funds*. The fourth character (immediately preceding the null character) is the character used to separate the international currency symbol from the monetary quantity.

char *currency_symbol
: The local currency symbol applicable to the current locale. This value along with positioning information is returned by a call to **nl_langinfo()** with **CRNCYSTR** as its argument (see *nl_langinfo*(3C)).

char *mon_decimal_point
: The decimal point used to format monetary quantities.

char *mon_thousands_sep
: The separator for groups of digits to the left of the decimal point in formatted monetary quantities.

char *mon_grouping
: A string where the numeric value of each byte indicates the size of each group of digits in formatted monetary quantities.

char *positive_sign
: The string used to indicate a nonnegative valued formatted monetary quantity.

char *negative_sign
: The string used to indicate a negative valued formatted monetary quantity.

char int_frac_digits
: The number of fractional digits (those to the right of the decimal point) to be displayed in an internationally formatted monetary quantity.

char frac_digits
: The number of fractional digits (those to the right of the decimal point) to be displayed in a locally formatted monetary quantity.

char p_cs_precedes
Set to 1 or 0 if the **currency_symbol** or **int_curr_symbol** respectively precedes or succeeds the value for a nonnegative formatted monetary quantity.

char p_sep_by_space
Set to 1 or 0 if the **currency_symbol** or **int_curr_symbol** respectively is or is not separated by a space from the value for a nonnegative formatted monetary quantity; and set to 2 if a space separates the symbol and the sign string, if adjacent.

char n_cs_precedes
Set to 1 or 0 if the **currency_symbol** or **int_curr_symbol** respectively precedes or succeeds the value for a negative formatted monetary quantity.

char n_sep_by_space
Set to 1 or 0 if the **currency_symbol** or **int_curr_symbol** respectively is or is not separated by a space from the value for a negative formatted monetary quantity; and set to 2 if a space separates the symbol and the sign string, if adjacent.

char p_sign_posn
Set to a value indicating the positioning of the **positive_sign** for a nonnegative formatted monetary quantity.

char n_sign_posn
Set to a value indicating the positioning of the **negative_sign** for a negative formatted monetary quantity.

The numeric value of each byte of **grouping** and **mon_grouping** is interpreted according to the following:

CHAR_MAX
No further grouping is to be performed.

0 The previous byte is to be repeatedly used for the remainder of the digits.

other The value is the number of digits that comprise the current group. The next byte is examined to determine the size of the next group of digits to the left of the current group.

The values of **p_sign_posn** and **n_sign_posn** are interpreted according to the following:

0 Parentheses surround the quantity and **currency_symbol** or **int_curr_symbol**.

1 The sign string precedes the quantity and **currency_symbol** or **int_curr_symbol**.

2 The sign string succeeds the quantity and **currency_symbol** or **int_curr_symbol**.

3 The sign string immediately precedes the **currency_symbol** or **int_curr_symbol**.

4 The sign string immediately succeeds the **currency_symbol** or **int_curr_symbol**.

localeconv() behaves as if no library function calls **localeconv()**.

APPLICATION USAGE

localeconv() is thread-safe. It is not async-cancel-safe.

EXTERNAL INFLUENCES

Locale

The **LC_NUMERIC** category influences the **decimal_point**, **thousands_sep**, and **grouping** members of the structure referenced by the pointer returned from a call to **localeconv()**.

The **LC_MONETARY** category influences all of the other members of this structure.

International Code Set Support

Single- and multibyte character code sets are supported.

RETURN VALUE

localeconv() returns a pointer to the filled-in **struct lconv**.

EXAMPLES

The following table illustrates the formatting used in five languages for monetary quantities.

Country	Positive format	Negative format	International format
en_US.iso88591	$1,234.56	-$1,234.56	USD 1,234.56
it_IT.iso88591	L.1.234	-L.1.234	ITL.1.234
nl_NL.iso88591	F 1.234,56	F -1.234,56	NLG 1.234,56
no_NO.iso88591	kr1.234,56	kr1.234,56-	NOK 1.234,56
pt_PT.iso88591	1,234$56	-1,234$56	PTE 1,234$56

For these five languages, the respective values for the monetary members of the structure returned by **localeconv()** are:

	en_US. iso88591	it_IT. iso88591	nl_NL. .iso88591	no_NO. iso88591	pt_PT. iso88591
int_curr_symbol	USD	ITL.	NLG	NOK	PTE
currency_symbol	$	L.	F	kr	$
mon_decimal_point	.	""	,	,	$
mon_thousands_sep	,	.	.	.	,
mon_grouping	\3	\3	\3	\3	\3
positive_sign	""	""	""	""	""
negative_sign	-	-	-	-	-
int_frac_digits	2	0	2	2	2
frac_digits	2	0	2	2	2
p_cs_precedes	1	1	1	1	0
p_sep_by_space	0	0	1	0	0
n_cs_precedes	1	1	1	1	0
n_sep_by_space	0	0	1	0	0
p_sign_posn	1	1	1	1	1
n_sign_posn	4	1	4	2	1

WARNINGS

The structure returned by **localeconv()** should not be modified by the calling program. Calls to **setlocale()** with categories **LC_ALL**, **LC_MONETARY**, or **LC_NUMERIC** can overwrite the contents of the structure that **localeconv()** points to when it returns (see *setlocale*(3C)).

AUTHOR

localeconv() was developed by OSF and HP.

SEE ALSO

nl_langinfo(3C), setlocale(3C), localedef(4), hpnls(5), langinfo(5).

STANDARDS CONFORMANCE

localeconv(): AES, SVID3, XPG4, ANSI C

NAME

log(), logf() - natural logarithm functions

SYNOPSIS

```
#include <math.h>
double log(double x);
float logf(float x);
```

DESCRIPTION

log() returns the natural logarithm of *x*. The value of *x* must be greater than zero.

logf() is a **float** version of **log()**; it takes a **float** argument and returns a **float** result. To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options.

logf() is not specified by any standard, but it is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, make sure your program includes <**math.h**>, and link in the math library by specifying **-lm** on the compiler or linker command line.

Millicode versions of the **log()** and **logf()** functions are available. Millicode versions of math library functions are usually faster than their counterparts in the standard library. To use these versions, compile your program with the **+Olibcalls** or the **+Oaggressive** optimization option.

If an error occurs, the millicode versions return the value described in the RETURN VALUE section, but do not set **errno**.

For more information, see the *HP-UX Floating-Point Guide*.

RETURN VALUE

If *x* is +INFINITY, **log()** returns +INFINITY.

If *x* is zero, **log()** returns −**HUGE_VAL**.

If *x* is NaN, **log()** returns NaN.

If *x* is less than zero, **log()** returns NaN and sets **errno** to [EDOM].

ERRORS

If **log()** fails, **errno** is set to the following value.

[EDOM] *x* is less than zero.

SEE ALSO

cbrt(3M), exp(3M), log10(3M), log1p(3M), log2(3M), pow(3M), sqrt(3M), math(5).

STANDARDS CONFORMANCE

log(): SVID3, XPG4.2, ANSI C

NAME

log10(), log10f() - common logarithm functions

SYNOPSIS

```
#include <math.h>
double log10(double x);
float log10f(float x);
```

DESCRIPTION

log10() returns the logarithm base ten of *x*. The value of *x* must be greater than zero.

log10f() is a **float** version of **log10()**; it takes a **float** argument and returns a **float** result. To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options.

log10f() is not specified by any standard, but it is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, make sure your program includes <**math.h**>, and link in the math library by specifying **-lm** on the compiler or linker command line.

Millicode versions of the **log10()** function are available. Millicode versions of math library functions are usually faster than their counterparts in the standard library. To use these versions, compile your program with the **+Olibcalls** or the **+Oaggressive** optimization option.

If an error occurs, the millicode versions return the value described in the *RETURN VALUE* section, but do not set **errno**.

For more information, see the *HP-UX Floating-Point Guide*.

RETURN VALUE

If *x* is +INFINITY, **log10()** returns +INFINITY.

If *x* is zero, **log10()** returns –**HUGE_VAL**.

If *x* is NaN, **log10()** returns NaN.

If *x* is less than zero, **log10()** returns NaN and sets **errno** to [EDOM].

ERRORS

If **log10()** fails, **errno** is set to the following value.

[EDOM] *x* is less than zero.

SEE ALSO

cbrt(3M), exp(3M), log(3M), log1p(3M), log2(3M), pow(3M), sqrt(3M), math(5).

STANDARDS CONFORMANCE

log10(): SVID3, XPG4.2, ANSI C

NAME

log1p( ) - natural logarithm function

SYNOPSIS

```
#include <math.h>
double log1p(double x);
```

DESCRIPTION

The **log1p( )** function computes logarithmic functions.

The **log1p( )** function is equivalent to **log(1 +** *x***)**, but may be more accurate for very small values of *x*.

The **expm1( )** and **log1p( )** functions are useful to guarantee that financial calculations of (((1+*x*)**n)-1)/*x*, namely:

expm1(n * **log1p**(*x*))/*x*

are accurate when *x* is very small (for example, when calculating small daily interest rates). These functions also simplify writing accurate inverse hyperbolic functions.

The ISO/ANSI C committee has approved the **log1p( )** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

log1p( ) returns the natural logarithm of (1 + *x*).

If *x* is +INFINITY, **log1p( )** returns +INFINITY.

If *x* = -1.0, **log1p( )** returns –**HUGE_VAL**.

If *x* is NaN, **log1p( )** returns NaN.

If *x* < -1.0, **log1p( )** returns NaN and sets **errno** to [EDOM].

ERRORS

If **log1p( )** fails, **errno** is set to the following value.

[EDOM] *x* is less than -1.0.

SEE ALSO

log(3M), expm1(3M), math(5).

STANDARDS CONFORMANCE

log1p( ): XPG4.2

NAME

log2(), log2f() - logarithm base two functions

SYNOPSIS

```
#include <math.h>
double log2(double x);
float log2f(float x);
```

DESCRIPTION

log2() returns the logarithm base two of *x*. The value of *x* must be greater than zero.

The ISO/ANSI C committee has approved the **log2()** function for inclusion in the C9X draft standard.

log2f() is a **float** version of **log2()**; it takes a **float** argument and returns a **float** result.

log2f() is not specified by any standard, but it is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

If *x* is +INFINITY, **log2()** returns +INFINITY.

If *x* is zero, **log2()** returns −**HUGE_VAL**.

If *x* is NaN, **log2()** returns NaN.

If *x* is less than zero, **log2()** returns NaN and sets **errno** to [EDOM].

ERRORS

If **log2()** fails, **errno** is set to the following value.

[EDOM] *x* is less than zero.

SEE ALSO

cbrt(3M), exp(3M), log(3M), log10(3M), log1p(3M), pow(3M), sqrt(3M), math(5).

NAME

logb( ) - radix-independent exponent

SYNOPSIS

```
#include <math.h>
double logb(double x);
```

DESCRIPTION

The **logb()** function computes the exponent of *x*. Formally, the return value is the integral part of log base *r* of |*x*| as a signed floating point value, for nonzero *x*, where *r* is the radix of the machine's floating-point arithmetic.

The **logb()** function is recommended by the IEEE-754 standard for floating-point arithmetic. The ISO/ANSI C committee has approved the **logb()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

Upon successful completion, **logb()** returns the exponent of **x**.

If *x* is NaN, **logb()** returns NaN.

If *x* is ±INFINITY, **logb()** returns +INFINITY.

If *x* is zero, **logb()** returns −**HUGE_VAL** and sets **errno** to [EDOM].

ERRORS

If **logb()** fails, **errno** is set to the following value.

[EDOM] *x* is zero.

SEE ALSO

ilogb(3M), scalb(3M), scalbn(3M), math(5), limits(5).

STANDARDS CONFORMANCE

logb(): SVID3, XPG4.2

NAME
logname() - return login name of user

SYNOPSIS
```
#include <unistd.h>
char *logname(void);
```

DESCRIPTION
logname() returns a pointer to the null-terminated login name; it extracts the **$LOGNAME** variable from the user's environment.

APPLICATION USAGE
logname() is thread-safe. It is not async-cancel-safe.

WARNINGS
logname() returns a pointer to static data that is overwritten by each subsequent call.

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

FILES
/etc/profile

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

STANDARDS CONFORMANCE
logname(): SVID2, XPG2

NAME
longname — get verbose description of current terminal

SYNOPSIS
```
#include <curses.h>
char *longname(void);
```

DESCRIPTION
The **longname()** function generates a verbose description of the current terminal. The maximum length of a verbose description is 128 bytes. It is defined only after the call to **initscr()** or **newterm()**.

RETURN VALUE
Upon successful completion, **longname()** returns a pointer to the description specified above. Otherwise, it returns a null pointer on error.

ERRORS
No errors are defined.

APPLICATION USAGE
The return value of **longname()** may point to a static area which is overwritten by a subsequent call to **newterm()**.

SEE ALSO
initscr(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The entry is rewritten for clarity. The argument list for the **longname()** function is explicitly declared as **void**.

NAME

lrint( ) - round to nearest long function

SYNOPSIS

```
#include <math.h>
long lrint(double x);
```

DESCRIPTION

lrint() rounds its argument to the nearest integral value, rounding according to the current rounding direction. If the rounded value is outside the range of **long**, the numeric result is unspecified.

The ISO/ANSI C committee has approved the **lrint()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

The **lrint()** function returns the rounded **long** value, using the current rounding direction.

ERRORS

No errors are defined.

SEE ALSO

ceil(3M), floor(3M), fabs(3M), fmod(3M), fegetround(3M), fesetround(3M), llrint(3M), lround(3M), llround(3M), rint(3M), round(3M), trunc(3M), math(5), fenv(5).

l

NAME

lround() - round to long function

SYNOPSIS

```
#include <math.h>
long lround(double x);
```

DESCRIPTION

lround() rounds its argument to the nearest integral value. An argument exactly halfway between two integers is rounded away from zero, regardless of the current rounding direction. Rounding away from zero also applies to the functions, **round** and **llround**.

If the rounded value is outside the range of **long**, the numeric result is unspecified.

The ISO/ANSI C committee has approved the **lround()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

The **lround()** function returns the rounded **long** value.

ERRORS

No errors are defined.

SEE ALSO

ceil(3M), floor(3M), fabs(3M), fmod(3M), fegetround(3M), fesetround(3M), lrint(3M), llrint(3M), llround(3M), rint(3M), round(3M), trunc(3M), math(5), fenv(5).

NAME

lsearch(), lfind() - linear search and update

SYNOPSIS

```
#include <search.h>

void *lsearch(
     const void *key,
     void *base,
     size_t *nelp,
     size_t width,
     int (*compar)(const void *, const void *)
);

void *lfind(
     const void *key,
     const void *base,
     size_t *nelp,
     size_t width,
     int (*compar)(const void *, const void *)
);
```

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 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() 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.

APPLICATION USAGE

lfind() and **lsearch()** are thread-safe and async-cancel-safe.

EXAMPLES

This code fragment reads in ≤ **TABSIZE** strings of length ≤ **ELSIZE** and stores them in a table, eliminating duplicates.

```
#include <stdio.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);
...
```

RETURN VALUE

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

WARNINGS

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

SEE ALSO

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

STANDARDS CONFORMANCE

lsearch(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

lfind(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

NAME

ltostr(), ultostr(), ltoa(), ultoa() - convert long integers to strings

SYNOPSIS

```
#include <stdlib.h>
char *ltostr(long n, int base);
char *ultostr(unsigned long n, int base);
char *ltoa(long n);
char *ultoa(unsigned long n);
```

Obsolescent Interfaces

```
int ltostr_r(long n, int base, char *buffer, int buflen);
int ultostr_r(unsigned long n, int base, char *buffer, int buflen);
int ltoa_r(long n, char *buffer, int buflen);
int ultoa_r(unsigned long n, char *buffer, int buflen);
```

DESCRIPTION

ltostr() Convert a signed long integer to the corresponding string representation in the specified base. The argument *base* must be between 2 and 36, inclusive.

ultostr() Convert an unsigned long integer to the corresponding string representation in the specified base. The argument *base* must be between 2 and 36, inclusive.

ltoa() Convert a signed long integer to the corresponding base 10 string representation, returning a pointer to the result.

ultoa() Convert an unsigned long integer to the corresponding base 10 string representation, returning a pointer to the result.

These functions are smaller and faster than using **sprintf()** for simple conversions (see *printf*(3S)).

Obsolescent Interfaces

ltostr_r(), **ultostr_r()**, **ltoa_r()**, **ultoa_r()** convert long integers to strings.

APPLICATION USAGE

ltostr(), **ultostr()**, **ltoa()** and **ultoa()** are thread-safe. These interfaces are not async-cancel-safe.

ERRORS

If the value of *base* is not between 2 and 36, **ltostr()** and **ultostr()** return NULL and set the external variable **errno** to ERANGE.

WARNINGS

The return values for **ltostr()**, **ultostr()**, **ltoa()** and **ultoa()** point to data whose content is overwritten by subsequent calls to these functions by the same thread.

ltostr_r(), **ultostr_r()**, **ltoa_r()** and **ultoa_r()** are obsolescent interface supported only for compatibility with existing DCE applications. New multi-threaded applications should use **ltostr()**, **ultostr()**, **ltoa()** and **ultoa()**.

AUTHOR

ltostr(), **ultostr()**, **ltoa()**, and **ultoa()** were developed by HP.

SEE ALSO

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

NAME

malloc(), free(), realloc(), calloc(), valloc(), mallopt(), mallinfo(), memorymap(), alloca() - main memory allocator

SYNOPSIS

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t nelem, size_t elsize);
void *realloc(void *ptr, size_t size);
void *valloc(size_t size);
void free(void *ptr);
void memorymap(int show_stats);
```

alloca

```
#include <alloca.h>
void *alloca(size_t size);
```

SYSTEM V SYNOPSIS

```
#include <malloc.h>
char *malloc(unsigned size);
void free(char *ptr);
char *realloc(char *ptr, unsigned size);
char *calloc(unsigned nelem, unsigned elsize);
int mallopt(int cmd, int value);
struct mallinfo mallinfo(void);
```

Remarks

The functionality in the old *malloc*(3X) package has been incorporated into *malloc*(3C). The library (**/usr/lib/libmalloc.a**) corresponding to the **-lmalloc** linker option is now an empty library. Makefiles that reference this library will continue to work. Applications that used the *malloc*(3X) package should still work properly with the new *malloc*(3C) package. If the old versions must be used, they are provided in files **/usr/old/libmalloc3x.a** and **/usr/old/libmalloc3c.o** for Release 8.07 only.

DESCRIPTION

The functions described in this manual entry provide a simple, general purpose memory allocation package:

malloc() Allocates space for a block of at least *size* bytes, but does not initialize the space.

calloc() Allocates space for an array of *nelem* elements, each of size *elsize* bytes, and initializes the space to zeros. Actual amount of space allocated will be at least *nelem* * *elsize* bytes.

realloc() Changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. Existing contents are unchanged up to the lesser of the new and old sizes. If *ptr* is a NULL pointer, **realloc()** behaves like **malloc()** for the specified size. If *size* is zero and *ptr* is not a NULL pointer, the object it points to is freed and NULL is returned.

valloc() Allocates space for a block of at least *size* bytes starting on a boundary aligned to a multiple of the value returned by **sysconf**(*_SC_PAGESIZE*). This space is uninitialized.

free() Deallocates the space pointed to by *ptr* (a pointer to a block previously allocated by **malloc()**, **realloc()**, or **calloc()**) and makes the space available for further allocation. If *ptr* is a NULL pointer, no action occurs.

mallopt() Provides for control over the allocation algorithm and other options in the *malloc*(3C) package. 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, then doles them out very quickly. The default value for *maxfast* is zero.

M_NLBLKS Set *numlblks* to *value*. The above mentioned "large groups" each contain *numlblks* blocks. *numlblks* must be greater than 1. The default value for *numlblks* is **100**.

M_GRAIN Set *grain* to *value*. The sizes of all blocks smaller than *maxfast* are considered to be rounded up to the nearest multiple of *grain*. *grain* must be greater than zero. The default value of *grain* is the smallest number of bytes that can accommodate alignment of any data type. *value* is 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.

M_BLOCK Block all blockable signals in **malloc()**, **realloc()**, **calloc()**, and **free()**. This option is provided for those who need to write signal handlers that allocate memory. When set, the *malloc*(3C) routines can be called from within signal handlers (they become re-entrant). Default action is *not* to block all blockable signals.

M_UBLOCK Do not block all blockable signals in **malloc()**, **realloc()**, **calloc()**, and **free()**. This option cancels signal blocking initiated by the **M_BLOCK** option.

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

mallopt() can be called repeatedly, but must not be called after the first small block is allocated (unless *cmd* is set to **M_BLOCK** or **M_UBLOCK**).

mallinfo() Provides instrumentation describing space usage, but cannot be called until the first small block is allocated. 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.

memorymap() Can be used to display the contents of the memory allocator. A list of addresses and block descriptions is written (using **printf()**) to standard output. If the value of the *show_stats* parameter is 1, statistics concerning number of blocks and sizes used will also be written. If the value is zero, only the memory map will be written.

The addresses and sizes displayed by **memorymap()** may not correspond to those requested by an application. The size of a block (as viewed by the allocator) includes header information and padding to properly align the block. The address is also offset by a certain amount to accommodate the header information.

alloca() Allocates space from the stack of the caller for a block of at least *size* bytes, but does not initialize the space. The space is automatically freed when the calling routine

exits.

Memory returned by **alloca()** is not related to memory allocated by other memory allocation functions. Behavior of addresses returned by **alloca()** as parameters to other memory functions is undefined.

The implementation of this routine is system dependent and its use is discouraged.

APPLICATION USAGE

malloc(), **free()**, **realloc()**, **calloc()**, **valloc()**, **mallopt()**, **mallinfo()**, **memorymap()** and **alloca()** are thread-safe. These interfaces are not async-cancel-safe.

RETURN VALUE

Upon successful completion, **malloc()**, **realloc()**, **calloc()**, and **valloc()** return a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object. Otherwise, they return a NULL pointer. If **realloc()** returns a NULL pointer, the memory pointed to by the original pointer is left intact.

mallopt() returns zero for success and nonzero for failure.

DIAGNOSTICS

malloc(), **realloc()**, **calloc()**, and **valloc()** return a NULL pointer if there is no available memory, or if the memory managed by **malloc()** has been detectably corrupted. This memory may become corrupted if data is stored outside the bounds of a block, or if an invalid pointer (a pointer not generated by **malloc()**, **realloc()**, or **calloc()**) is passed as an argument to **free()** or **realloc()**.

If **mallopt()** is called after any allocation of a small block and *cmd* is not set to **M_BLOCK** or **M_UBLOCK**, or if *cmd* or *value* is invalid, nonzero is returned. Otherwise, it returns zero.

ERRORS

[ENOMEM] **malloc()**, **realloc()**, **calloc()**, and **valloc()** set **errno** to ENOMEM and return a NULL pointer when an out-of-memory condition arises.

[EINVAL] **malloc()**, **realloc()**, **calloc()**, and **valloc()** set **errno** to EINVAL and return a NULL pointer when the memory being managed by **malloc()** has been detectably corrupted.

WARNINGS

malloc() functions use **brk()** and **sbrk()** (see *brk*(2)) to increase the address space of a process. Therefore, an application program that uses **brk()** or **sbrk()** must not use them to decrease the address space, because this confuses the **malloc()** functions.

free() and **realloc()** do not check their pointer argument for validity.

If **free()** or **realloc()** is passed a pointer that was not the result of a call to **malloc()**, **realloc()**, **calloc()**, or **valloc()**, or if space assigned by an allocation function is overrun, loss of data, a memory fault, bus error, or an infinite loop may occur at that time or during any subsequent call to **malloc()**, **realloc()**, **calloc()**, **valloc()**, or **free()**.

The following actions are not supported and cause undesirable effects:

- Attempting to **free()** or **realloc()** a pointer not generated as the result of a call to **malloc()**, **realloc()**, **calloc()**, or **valloc()**.

The following actions are strongly discouraged and may be unsupported in a future implementation of **malloc()**:

- attempting to **free()** the same block twice,
- depending on unmodified contents of a block after it has been freed.

Undocumented features of earlier memory allocators have not been duplicated.

COMPATIBILITY

The only external difference between the old *malloc*(3X) allocator and the *malloc*(3C) allocator is that the old allocator would return a NULL pointer for a request of zero bytes. The *malloc*(3C) allocator returns a valid memory address. This is not a concern for most applications.

Although the current implementation of *malloc*(3C) allows for freeing a block twice and does not corrupt the contents of a block after it is freed (until the next call to **realloc()**, **calloc()**, **malloc()**, or **valloc()**), support for these features may be discontinued in a future implementation of *malloc*(3C) and should not be used.

SEE ALSO

brk(2), errno(2).

STANDARDS CONFORMANCE

malloc(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

calloc(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

free(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

mallinfo(): SVID2, XPG2

mallopt(): SVID2, SVID3, XPG2

realloc(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

memccpy(), memchr(), memcmp(), memcpy(), memmove(), memset(), bcopy(), bcmp(), bzero(), ffs() - memory operations

SYNOPSIS

```
#include <string.h>
void *memccpy(void *s1, const void *s2, int c, size_t n);
void *memchr(const void *s, int c, size_t n);
int memcmp(const void *s1, const void *s2, size_t n);
void *memcpy(void *s1, const void *s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
void *memset(void *s, int c, size_t n);
#include <strings.h>
int bcmp(const void *s1, const void *s2, size_t n);
void bcopy(const void *s1, void *s2, size_t n);
void bzero(void *s, size_t n);
int ffs(int i);
```

Remarks:

bcmp(), **bcopy()**, **bzero()**, **ffs()**, and <**strings.h**> are provided solely for portability of BSD applications, and are not recommended for new applications where portability is important. For portable applications, use **memcmp()**, **memmove()**, and **memset()**, respectively. **ffs()** has no portable equivalent.

m

DESCRIPTION

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

Definitions for all these functions, the type **size_t**, and the constant **NULL** are provided in the <**string.h**> header file.

memccpy() Copy bytes from the object pointed to by *s2* into the object pointed to by *s1*, stopping after the first occurrence of byte *c* has been copied, or after *n* bytes have been copied, whichever comes first. If copying takes place between objects that overlap, the behavior is undefined. **memccpy()** returns a pointer to the byte after the copy of *c* in *s1*, or a NULL pointer if *c* was not found in the first *n* bytes of *s2*.

memchr() Locate the first occurrence of *c* (converted to an **unsigned char**) in the initial *n* bytes (each interpreted as **unsigned char**) of the object pointed to by *s*. **memchr()** returns a pointer to the located byte, or a NULL pointer if the byte does not occur in the object.

memcmp() Compare the first *n* bytes of the object pointed to by *s1* to the first *n* bytes of the object pointed to by *s2*. **memcmp()** returns an integer greater than, equal to, or less than zero, according to whether the object pointed to by *s1* is greater than, equal to, or less than the object pointed to by *s2*. The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes (both interpreted as **unsigned char**) that differ in the objects being compared.

memcpy() Copy *n* bytes from the object pointed to by *s2* into the object pointed to by *s1*. If copying takes place between objects that overlap, the behavior is undefined. **memcpy()** returns the value of *s1*.

memmove() Copy *n* bytes from the object pointed to by *s2* into the object pointed to by *s1*. Copying takes place as if the *n* bytes from the object pointed to by *s2* are first copied into a temporary array of *n* bytes that does not overlap the objects pointed to by *s1* and *s2*, and then the *n* bytes from the temporary array are copied into the object pointed to by *s1*. **memmove()** returns the value of *s1*.

memset() Copy the value of *c* (converted to an **unsigned char**) into each of the first *n* bytes of the object pointed to by *s*. **memset()** returns the value of *s*.

bcopy() copies *n* bytes from the area pointed to by *s1* to the area pointed to by *s2*.

bcmp() Compare the first *n* bytes of the area pointed to by *s1* with the area pointed to by *s2*. **bcmp()** returns zero if they are identical; non-zero otherwise. Both areas are assumed to be *n* bytes in length.

bzero() Clear *n* bytes in the area pointed to by *s* by setting them to zero.

ffs() Find the first bit set (beginning with the least significant bit) and return the index of that bit. Bits are numbered starting at one. A return value of 0 indicates that *i* is zero.

International Code Set Support
These functions support only single-byte byte code sets.

APPLICATION USAGE
The interfaces **memccpy()**, **memchr()**, **memcmp()**, **memcpy()**, **memmove()**, **memset()**, **bcopy()**, **bcmp()**, **bzero()** and **ffs()** are thread-safe and async-cancel-safe.

WARNINGS
The functions defined in <**string.h**> were previously defined in <**memory.h**>.

FILES
/usr/include/string.h

SEE ALSO
string(3C).

STANDARDS CONFORMANCE
memccpy(): AES, SVID2, SVID3, XPG2, XPG3, XPG4
memchr(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, ANSI C
memcmp(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, ANSI C
memcpy(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, ANSI C
memmove(): AES, SVID3, XPG4, ANSI C
memset(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, ANSI C

(ENHANCED CURSES)

NAME

meta — enable/disable meta-keys

SYNOPSIS

```
#include <curses.h>
int meta(WINDOW *win, bool bf);
```

DESCRIPTION

Initially, whether the terminal returns 7 or 8 significant bits on input depends on the control mode of the display driver (see the *X/Open System Interface Definitions, Issue 4, Version 2* specification, *General Terminal Interface*). To force 8 bits to be returned, invoke *meta*(*win*, TRUE). To force 7 bits to be returned, invoke *meta*(*win*, FALSE). The *win* argument is always ignored. If the **terminfo** capabilities **smm** (meta_on) and **rmm** (meta_off) are defined for the terminal, **smm** is sent to the terminal when *meta*(*win*, TRUE) is called and **rmm** is sent when *meta*(*win*, FALSE) is called.

RETURN VALUE

Upon successful completion, **meta()** returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

The same effect is achieved outside Curses using the CS7 or CS8 control mode flag specified in the *X/Open System Interface Definitions, Issue 4, Version 2* specification (*General Terminal Interface*).

The **meta()** function was designed for use with terminals with 7-bit character sets and a "meta" key that could be used to set the eighth bit.

SEE ALSO

Input Processing in curses_intro(3X), getch(3X), <curses.h>, *X/Open System Interface Definitions, Issue 4, Version 2* specification, Section 9.2, *Parameters That Can Be Set* (ISTRIP flag).

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

mkdirp(), rmdirp() - create, remove directories in a path

SYNOPSIS

```
#include <libgen.h>
int mkdirp (const char *path, mode_t mode);
int rmdirp (char *d, char *d1);
```

DESCRIPTION

mkdirp creates all the missing directories in the given *path* with the given *mode*. [See *chmod*(2) for the values of *mode*.] The protection part of the *mode* argument is modified by the process's file creation mask (see *umask*(2)).

rmdirp removes directories in path *d*. This removal starts at the end of the path and moves back toward the root as far as possible. If an error occurs, the remaining path is stored in *d1*. **rmdirp** returns a 0 only if it is able to remove every directory in the path.

RETURN VALUE

If a needed directory cannot be created, **mkdirp** returns -1 and sets **errno** to one of the **mkdir** error numbers. If all the directories are created, or existed to begin with, it returns zero.

APPLICATION USAGE

mkdirp is thread-safe. It is not async-cancel-safe. A cancellation point may occur when a thread is executing **mkdirp**. rmdirp is thread-safe and async-cancel-safe.

EXAMPLES

```
/* create scratch directories */
if(mkdirp("/tmp/sub1/sub2/sub3", 0755) == -1) {
    fprintf(stderr, "cannot create directory");
    exit(1);
}
chdir("/tmp/sub1/sub2/sub3");
.
.
.
/* cleanup */
chdir( /tmp");"
rmdirp("sub1/sub2/sub3");
```

WARNINGS

mkdirp uses **malloc** to allocate temporary space for the string.

rmdirp returns -2 if a "**.**" or "**..**" is in the path and -3 if an attempt is made to remove the current directory. If an error occurs other than one of the above, -1 is returned.

SEE ALSO

mkdir(2), rmdir(2), umask(2).

NAME

mkfifo() - make a FIFO file

SYNOPSIS

```
#include <sys/stat.h>
int mkfifo(char *path, mode_t mode);
```

DESCRIPTION

mkfifo() creates a new FIFO (first-in-first-out) file, at the path name to which *path* points. The file permission bits of the new file are initialized from the *mode* argument, as modified by the process's file creation mask: for each bit set in the process's file mode creation mask, the corresponding bit in the new file's mode is cleared (see *umask*(2)). Bits in *mode* other than the file permission bits are ignored.

The FIFO owner ID is set to the process's effective-user-ID. The FIFO group ID is set to the group ID of the parent directory if the set-group-ID bit is set on that directory. Otherwise the FIFO group ID is set to the process's effective group ID.

For details of the I/O behavior of pipes see *read*(2) and *write*(2).

The following symbolic constants are defined in the <**sys/stat.h**> header, and should be used to construct the value of the *mode* argument. The value passed should be the bitwise inclusive OR of the desired permissions:

S_IRUSR	Read by owner.
S_IWUSR	Write by owner.
S_IRGRP	Read by group.
S_IWGRP	Write by group.
S_IROTH	Read by other users.
S_IWOTH	Write by other users.

APPLICATION USAGE

m **mkfifo()** is thread-safe and async-cancel-safe.

RETURN VALUE

mkfifo() returns 0 upon successful completion. Otherwise, it returns -1, no FIFO is created, and **errno** is set to indicate the error.

ERRORS

mkfifo() fails and the new file is not created if any of the following conditions are encountered:

[EACCES]	A component of the path prefix denies search permission.
[EEXIST]	The named file already exists.
[EFAULT]	The *path* argument points outside the process's allocated address space. The reliable detection of this error is implementation dependent.
[ELOOP]	Too many symbolic links encountered in translating the path name.
[ENAMETOOLONG]	The length of the specified path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.
[ENOENT]	A component of the path prefix does not exist.
[ENOENT]	The *path* argument is null.
[ENOSPC]	Not enough space on the file system.
[ENOTDIR]	A component of the path prefix is not a directory.
[EROFS]	The directory in which the file is being created is located in a read-only file system.

AUTHOR

mkfifo() was developed by HP.

SEE ALSO
chmod(2), mknod(2), pipe(2), stat(2), umask(2), fs(4), mknod(5), stat(5), types(5).

STANDARDS CONFORMANCE
mkfifo(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

mktemp(), mkstemp() - make a unique file name

SYNOPSIS

```
#include <stdlib.h>
char *mktemp(char *template);
int mkstemp(char *template);
```

Remarks:

These functions are provided solely for backward compatibility and importability of applications, and are not recommended for new applications where portability is important. For portable applications, use **tmpfile()** instead (see *tmpfile*(3S)).

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()** replaces the **X**s with a letter and the current process ID. The letter is chosen such that the resulting name does not duplicate the name of an existing file. If there are fewer than six **X**s, the letter is dropped first, followed by dropping the high-order digits of the process ID.

mkstemp() makes the same replacement to the template, but also returns a file descriptor for the template file after opening the file for reading and writing. **mkstemp()** thus prevents any possible race condition between testing whether the file exists and opening it for use.

APPLICATION USAGE

mktemp() and **mkstemp()** are thread-safe. **mktemp()** is async-cancel-safe. **mkstemp()** is not async-cancel-safe. A cancellation point may occur when a thread is executing **mkstemp()**.

RETURN VALUE

mktemp() returns its argument except when it runs out of letters, in which case the result is a pointer to the empty string **""**.

mkstemp() returns an open file descriptor upon successful completion, or -1 if no suitable file could be created.

WARNINGS

It is possible to run out of letters.

mktemp() and **mkstemp()** do not check to determine whether the file name part of *template* exceeds the maximum allowable file name length.

SEE ALSO

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

STANDARDS CONFORMANCE

mktemp(): SVID2, SVID3, XPG2

NAME

mktimer - allocate a per-process timer

SYNOPSIS

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

timer_t mktimer(int clock_type, int notify_type, void *itimercbp);
```

DESCRIPTION

The **mktimer()** function is used to allocate a per-process timer using the specified system-wide clock as the timing base. **mktimer()** returns an unique timer ID of type **timer_t** used to identify the timer in timer requests (see *gettimer*(3C)). **clock_type** specifies the system-wide clock to be used as the timing base for the new timer. **notify_type** specifies the mechanism by which the process is to be notified when the timer expires.

mktimer() supports one per-process timer with a **clock_type** of **TIMEOFDAY** and **notify_type** of **DELIVERY_SIGNALS**.

If **notify_type** is **DELIVERY_SIGNALS**, the system causes a **SIGALRM** signal to be sent to the process whenever the timer expires.

For **clock_type TIMEOFDAY**, the machine-dependent clock resolution and maximum value are **1/HZ** and **MAX_ALARM** seconds, respectively. These constants are defined in <**sys/param.h**>.

APPLICATION USAGE

mktimer() is thread-safe. It is not async-cancel-safe.

RETURN VALUE

Upon successful completion, **mktimer()** returns a **timer_t**, which can be passed to the per_process timer calls. If unsuccessful, **mktimer()** returns a value of (**timer_t**)−1 and sets **errno** to indicate the error.

ERRORS

mktimer() fails if any of the following conditions are encountered:

[EAGAIN] The calling process has already allocated all of the timers it is allowed.

[EINVAL] **clock_type** is not defined, or does not allow the specified notification mechanism.

FILES

/usr/include/sys/timers.h

/usr/include/sys/param.h

SEE ALSO

timers(2), getclock(3C), gettimer(3C), reltimer(3C), rmtimer(3C), setclock(3C),

STANDARDS CONFORMANCE

mktimer(): AES

NAME

modf() - decompose floating-point number

SYNOPSIS

```
#include <math.h>
double modf(double x, double *iptr);
```

DESCRIPTION

The **modf()** function breaks the argument *x* into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a **double** in the object pointed to by *iptr*.

To use this function, make sure your program includes <**math.h**>, and link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

Upon successful completion, the **modf()** function returns the signed fractional part of *x*.

If *x* is NaN, **modf()** returns NaN.

If the correct value after rounding would be smaller in magnitude than **MINDOUBLE**, **modf()** returns zero.

ERRORS

No errors are defined.

SEE ALSO

frexp(3M), ldexp(3M), scalb(3M), scalbn(3M), math(5), values(5).

STANDARDS CONFORMANCE

modf(): SVID3, XPG4.2, ANSI C

m

NAME

monitor() - prepare execution profile

SYNOPSIS

```
#include <mon.h>

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

DESCRIPTION

An executable program created by **cc -p** automatically includes calls for **monitor()** with default parameters; **monitor()** need not be called explicitly except to gain fine control over profiling.

monitor() is an interface to *profil*(2). *lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a (user-supplied) array of *bufsize* WORDs (defined in the <**mon.h**> header file). **monitor()** arranges to record in the buffer a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions. The lowest address sampled is that of *lowpc* and the highest is just below *highpc*. *lowpc* must not equal 0 for this use of *monitor*. Not more than *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 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, ((int(*)())& etext, buf, bufsize, nfunc);
```

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

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

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

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

FILES

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

SEE ALSO

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

STANDARDS CONFORMANCE

monitor(): SVID2, SVID3, XPG2

NAME

mount - keep track of remotely mounted file systems

SYNOPSIS

```
#include <rpcsvc/mount.h>
```

DESCRIPTION

Program number

MOUNTPROG

The following are the **xdr** routines provided:

```
xdr_exportbody(xdrs, ex)
    XDR *xdrs;
    struct exports *ex;

xdr_exports(xdrs, ex);
    XDR *xdrs;
    struct exports **ex;

xdr_fhandle(xdrs, fh);
    XDR *xdrs;
    fhandle_t *fp;

xdr_fhstatus(xdrs, fhs);
    XDR *xdrs;
    struct fhstatus *fhs;

xdr_groups(xdrs, gr);
    XDR *xdrs;
    struct groups *gr;

xdr_mountbody(xdrs, ml)
    XDR *xdrs;
    struct mountlist *ml;

xdr_mountlist(xdrs, ml);
    XDR *xdrs;
    struct mountlist **ml;

xdr_path(xdrs, path);
    XDR *xdrs;
    char **path;
```

Procs

MOUNTPROC_MNT Argument of **xdr_path**; returns **fhstatus**. Requires UNIX authentication.

MOUNTPROC_DUMP
No arguments; returns **struct mountlist**

MOUNTPROC_UMNT
Argument of **xdr_path**; no results. Requires UNIX authentication.

MOUNTPROC_UMNTALL
No arguments; no results. Requires UNIX authentication. Unmounts all remote mounts of sender.

MOUNTPROC_EXPORT
No arguments; returns **struct exports**

MOUNTPROC_EXPORTALL
No arguments; returns **struct exports**

Versions

MOUNTVERS_ORIG

Structures

```
struct mountlist {              /* what is mounted */
    char *ml_name;
```

```
        char *ml_path;
        struct mountlist *ml_nxt;
        };
    struct fhstatus {
        int fhs_status;
        fhandle_t fhs_fh;
        };
    /*
     * List of exported directories
     * An export entry with ex_groups
     * NULL indicates an entry which is exported to the world.
     */
    struct exports {
        dev_t           ex_dev;     /* dev of directory */
        char           *ex_name;    /* name of directory */
        struct groups  *ex_groups;  /* groups allowed to */
                                    /* mount this entry */
        struct exports *ex_next;
        };
    struct groups {
        char           *g_name;
        struct groups  *g_next;
        };
```

AUTHOR

mount(3N) was developed by Sun Microsystems, Inc.

SEE ALSO

mount(1M), mountd(1M), showmount(1M).

NAME
move, wmove — window cursor location functions

SYNOPSIS
```
#include <curses.h>
int move(int y, int x);
int wmove(WINDOW *win, int y, int x);
```

DESCRIPTION
The **move()** and **wmove()** functions move the cursor associated with the current or specified window to (*y*, *x*) relative to the window's origin. This function does not move the terminal's cursor until the next refresh operation.

RETURN VALUE
Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS
No errors are defined.

SEE ALSO
doupdate(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The entry is rewritten for clarity.

m

NAME

mblen(), mbtowc(), mbstowcs(), wctomb(), wcstombs() - multibyte characters and strings conversions

SYNOPSIS

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
int mbtowc(wchar_t *pwc, const char *s, size_t n);
int wctomb(char *s, wchar_t wchar);
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

DESCRIPTION

A multibyte character is composed of one or more bytes that represent a "whole" character in a character encoding. A wide character (type of **wchar_t**) is composed of a fixed number of bytes whose code value can represent any character in a character encoding.

mblen() Determine the number of bytes in the multibyte character pointed to by *s*. Equivalent to:

```
mbtowc((wchar_t *)0, s, n);
```

If *s* is a null pointer, *mblen* returns a nonzero or zero value, depending on whether the multibyte character encodings do or do not have state-dependent encodings, respectively. Since no character encodings currently supported by HP-UX are state-dependent, zero is always returned in this case. However, for maximum portability to other systems, application programs should not depend on this.

If *s* is not a null pointer, *mblen* returns the number of bytes in the multibyte character if the next *n* or fewer bytes form a valid multibyte character, or return -1 if they do not form a valid multibyte character. If *s* points to the null character, *mblen* returns 0.

mbtowc() Determine the number of bytes in the multibyte character pointed to by *s*, determine the code for the value of type **wchar_t** corresponding to that multibyte character, then store the code in the object pointed to by *pwc*. The value of the code corresponding to the null character is zero. At most *n* characters are examined, starting at the character pointed to by *s*.

If *s* is a null pointer, **mbtowc()** returns a non-zero or zero value, depending on whether the multibyte character encodings do or do not have state-dependent encodings, respectively. Since no character encodings currently supported by HP-UX are state-dependent, zero is always returned in this case. However, for maximum portability to other systems, application programs should not depend on this.

If *s* is not a null pointer, **mbtowc()** returns the number of bytes in the converted multibyte character if the next *n* or fewer bytes form a valid multibyte character, or -1 if they do not form a valid multibyte character. If *s* points to the null character, **mbtowc()** returns 0. The value returned is never greater than *n* or the value of the **MB_CUR_MAX** macro.

wctomb() Determine the number of bytes needed to represent the multibyte character corresponding to the code whose value is *wchar* and store the multibyte character representation in the array object pointed to by *s*. At most **MB_CUR_MAX** characters are stored.

If *s* is a null pointer, **wctomb()** returns a nonzero or zero value, depending on whether the multibyte character encodings do or do not have state-dependent encodings, respectively. Since no character encodings currently supported by HP-UX are state-dependent, zero is always returned in this case. However, for maximum portability to other systems, application programs should not depend on this.

If *s* is not a null pointer, **wctomb()** returns the number of bytes in the multibyte character corresponding to the value of *wchar*, or -1 if the value of *wchar* does not correspond to a valid multibyte character. The value returned is never greater than the value of the **MB_CUR_MAX** macro.

mbstowcs()

Convert a sequence of multibyte characters from the array pointed to by *s* into a sequence of corresponding codes and store these codes into the array pointed to by *pwcs*, stopping after either *n* codes or a code with value zero (a converted null character) is stored. Each multibyte

character is converted as if by a call to **mbtowc()**. No more than *n* elements are modified in the array pointed to by *pwcs*.

If an invalid multibyte character is encountered, **mbstowcs()** returns (size_t)−1. Otherwise, **mbstowcs()** returns the number of array elements modified, not including a terminating zero code, if any. The array is not null- or zero-terminated if the value returned is *n*. If *pwcs* is a null pointer, **mbstowcs()** returns the number of elements required for the wide-character-code array.

wcstombs()
Convert a sequence of codes corresponding to multibyte characters from the array pointed to by *pwcs* into a sequence of multibyte characters and store them into the array pointed to by *s*, stopping if a multibyte character exceeds the limit of *n* total bytes or if a null character is stored. Each code is converted as if by a call to **wctomb()**. No more than *n* bytes are modified in the array pointed to by *s*.

If a code is encountered that does not correspond to a valid multibyte character, **wcstombs()** returns (size_t)−1. Otherwise, **wcstombs()** returns the number of bytes modified, not including a terminating null character, if any. The array is not null- or zero-terminated if the value returned is *n*. If *s* is a null pointer, **wcstombs()** returns the number of bytes required for the character array.

APPLICATION USAGE

mblen(), **mbtowc()**, **mbstowcs()**, **wctomb()** and **wcstombs()** are thread-safe and async-cancel-safe.

EXTERNAL INFLUENCES

Locale

The **LC_CTYPE** category determines the behavior of the multibyte character and string functions.

ERRORS

mblen(), **mbstowcs()**, **mbtowc()**, **wcstombs()** and **wctomb()** may fail and **errno** is set if the following condition is encountered:

[EILSEQ] An invalid multibyte sequence or wide character code was found.

WARNINGS

With the exception of ASCII characters, the code values of wide characters (type of **wchar_t**) are specific to the effective locale specified by the **LC_CTYPE** environment variable. These values may not be compatible with values obtained by specifying other locales that are supported now, or which may be supported in the future. It is recommended that wide character constants and wide string literals (see the *C Reference Manual*) not be used, and that wide character code values not be stored in files or devices because future standards may dictate changes in the code value assignments of the wide characters. However, wide character constants and wide string literals corresponding to the characters of the ASCII code set can be safely used since their values are guaranteed to be the same as their ASCII code set values.

AUTHOR

The multibyte functions in this entry were developed by OSF and HP.

SEE ALSO

setlocale(3C), wctype(3C).

STANDARDS CONFORMANCE

mblen(): AES, SVID3, XPG4, ANSI C

mbstowcs(): AES, SVID3, XPG4, ANSI C

mbtowc(): AES, SVID3, XPG4, ANSI C

wcstombs(): AES, SVID3, XPG4, ANSI C

wctomb(): AES, SVID3, XPG4, ANSI C

(ENHANCED CURSES)

NAME

mvcur — output cursor movement commands to the terminal

SYNOPSIS

```
#include <curses.h>
int mvcur(int oldrow, int oldcol, int newrow, int newcol);
```

DESCRIPTION

The **mvcur()** function outputs one or more commands to the terminal that move the terminal's cursor to (*newrow*, *newcol*), an absolute position on the terminal screen. The (*oldrow*, *oldcol*) arguments specify the former cursor position. Specifying the former position is necessary on terminals that do not provide coordinate-based movement commands. On terminals that provide these commands, Curses may select a more efficient way to move the cursor based on the former position. If (*newrow*, *newcol*) is not a valid address for the terminal in use, **mvcur()** fails. If (*oldrow*, *oldcol*) is the same as (*newrow*, *newcol*), then **mvcur()** succeeds without taking any action. If **mvcur()** outputs a cursor movement command, it updates its information concerning the location of the cursor on the terminal.

RETURN VALUE

Upon successful completion, **mvcur()** returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

After use of **mvcur()**, the model Curses maintains of the state of the terminal might not match the actual state of the terminal. The application should touch and refresh the window before resuming conventional use of Curses.

SEE ALSO

doupdate(3X), is_linetouched(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

mvderwin — define window coordinate transformation

SYNOPSIS

```
#include <curses.h>
int mvderwin(WINDOW *win, int par_y, int par_x);
```

DESCRIPTION

The **mvderwin()** function specifies a mapping of characters. The function identifies a mapped area of the parent of the specified window, whose size is the same as the size of the specified window and whose origin is at (*par_y*, *par_x*) of the parent window.

- During any refresh of the specified window, the characters displayed in that window's display area of the terminal are taken from the mapped area.
- Any references to characters in the specified window obtain or modify characters in the mapped area.

That is, **mvderwin()** defines a coordinate transformation from each position in the mapped area to a corresponding position (same *y*, *x* offset from the origin) in the specified window.

RETURN VALUE

Upon successful completion, **mvderwin()** returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

SEE ALSO

derwin(3X), doupdate(3X), dupwin(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

mvprintw, mvwprintw, printw, wprintw — print formatted output in window

SYNOPSIS

```
#include <curses.h>
int mvprintw(int y, int x, char *fmt, ...);
int mvwprintw(WINDOW *win, int y, int x, char *fmt, ...);
int printw(char *fmt, ...);
int wprintw(WINDOW *win, char *fmt, ...);
```

DESCRIPTION

The **mvprintw()**, **mvwprintw()**, **printw()** and **wprintw()** functions are analogous to **printf()**. The effect of these functions is as though **sprintf()** were used to format the string, and then **waddstr()** were used to add that multi-byte string to the current or specified window at the current or specified cursor position.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

SEE ALSO

addnstr(3X), fprintf() (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4

The entry is rewritten for clarity and its name is changed from **printw()** to **mvprintw()**.

m

NAME

mvscanw, mvwscanw, scanw, wscanw — convert formatted input from a window

SYNOPSIS

```
#include <curses.h>
int mvscanw(int y, int x, char *fmt, ...);
int mvwscanw(WINDOW *win, int y, int x, char *fmt, ...);
int scanw(char *fmt, ...);
int wscanw(WINDOW *win, char *fmt, ...);
```

DESCRIPTION

These functions are similar to **scanf()**. Their effect is as though **mvwgetstr()** were called to get a multi-byte character string from the current or specified window at the current or specified cursor position, and then **sscanf()** were used to interpret and convert that string.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

SEE ALSO

getnstr(3X), printw(3X), fscanf() (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification), wcstombs() (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4

The entry is rewritten for clarity and its name is changed from **scanw()** to **mvscanw()**.

NAME
mvwin — move window

SYNOPSIS
```
#include <curses.h>
int mvwin(WINDOW *win, int y, int x);
```

DESCRIPTION
The **mvwin()** function moves the specified window so that its origin is at position (*y*, *x*). If the move would cause any portion of the window to extend past any edge of the screen, the function fails and the window is not moved.

RETURN VALUE
Upon successful completion, **mvwin()** returns OK. Otherwise, it returns ERR.

ERRORS
No errors are defined.

APPLICATION USAGE
The application should not move subwindows by calling **mvwin()**.

SEE ALSO
derwin(3X), doupdate(3X), is_linetouched(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The entry is rewritten for clarity.

NAME

nan() - string-to-NaN conversion function

SYNOPSIS

```
#include <math.h>

double nan(const char *tagp);
```

DESCRIPTION

The **nan()** function creates a quiet NaN from the specified string. The call **nan("***n-char-sequence***")** is equivalent to **strtod("NAN(***n-char-sequence***)", (char**) NULL)**. The call **nan("")** is equivalent to **strtod("NAN()", (char**) NULL)**. If **tagp** does not point to an *n-char-sequence* string, the result NaN's content is unspecified.

The ISO/ANSI C committee has approved the **nan()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

The **nan()** function returns a quiet NaN.

ERRORS

No errors are defined.

SEE ALSO

copysign(3M), nextafter(3M), math(5).

n

(ENHANCED CURSES)

NAME

napms — suspend the calling process

SYNOPSIS

```
#include <curses.h>
int napms(int ms);
```

DESCRIPTION

The **napms()** function takes at least *ms* milliseconds to return.

RETURN VALUE

The **napms()** function returns OK.

ERRORS

No errors are defined.

APPLICATION USAGE

A more reliable method of achieving a timed delay is the **usleep()** function.

SEE ALSO

delay_output(3X), usleep(2) (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

n

NAME

dbm_open, dbm_close, dbm_fetch, dbm_store, dbm_delete, dbm_firstkey, dbm_nextkey, dbm_error, dbm_clearerr - database subroutines

SYNOPSIS

```
#include <ndbm.h>
DBM *dbm_open(const char *file, int flags, mode_t mode);
void dbm_close(DBM *db);
datum dbm_fetch(DBM *db, datum key);
int dbm_store(DBM *db, datum key, datum content, int flags);
int dbm_delete(DBM *db, datum key);
datum dbm_firstkey(DBM *db);
datum dbm_nextkey(DBM *db);
int dbm_error(DBM *db);
int dbm_clearerr(DBM *db);
```

DESCRIPTION

These functions maintain key/content pairs in a database. They handle very large (a billion blocks (block = 1024 bytes)) databases and can access a keyed item in one or two file system accesses.

key and *content* parameters are described by the **datum** type. A **datum** specifies a string of *dsize* bytes pointed to by *dptr*. Arbitrary binary data, as well as normal ASCII strings, are allowed. The database is stored in two files. One file is a directory containing a bit map of keys and has **.dir** as its suffix. The second file contains all data and has **.pag** as its suffix.

Before a database can be accessed, it must be opened by **dbm_open**. This will open and/or create the files *file***.dir** and *file***.pag** depending on the *flags* parameter (see *open*(2)).

Once open, the data stored under a key is accessed by **dbm_fetch** and data is placed under a key by **dbm_store**. The *flags* field can be either **DBM_INSERT** or **DBM_REPLACE**. **DBM_INSERT** can only insert new entries into the database, and cannot change an existing entry having the same key. **DBM_REPLACE** replaces an existing entry if it has the same key. A key (and its associated contents) is deleted by **dbm_delete**. A linear pass through all keys in a database can be made in (apparently) random order by use of **dbm_firstkey** and **dbm_nextkey**. **dbm_firstkey** returns the first key in the database. **dbm_nextkey** returns the next key in the database, The following code can be used to traverse the database:

```
for (key = dbm_firstkey(db); key.dptr != NULL; key = dbm_nextkey(db))
```

dbm_error returns non-zero when an error has occurred reading or writing the database. **dbm_clearerr** resets the error condition on the named database.

DIAGNOSTICS

All functions that return an **int** indicate errors with negative values and success with zero. Functions that return a **datum** indicate errors with a null *dptr*. If **dbm_store** is called with a *flags* value of **DBM_INSERT** and finds an existing entry with the same key, a value of **1** is returned. If a call to **dbm_store** results in an internal block overflow, a value of –2 is returned.

APPLICATION USAGE

The interfaces **dbm_open()**, **dbm_close()**, **dbm_fetch()**, **dbm_store()**, **dbm_delete()**, **dbm_firstkey()**, **dbm_nextkey()**, **dbm_error()** and **dbm_clearerr** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing **dbm_open()**, **dbm_close()**, **dbm_fetch()**, **dbm_store()**, **dbm_delete()**, **dbm_firstkey()**, **dbm_nextkey()** or **dbm_error()**.

WARNINGS

The *ndbm* functions provided in this library should not be confused in any way with those of a general-purpose database management system. These functions *do not* provide for multiple search keys per entry, they *do not* protect against multi-user access (in other words they do not lock records or files), and they *do*

not provide the many other useful database functions that are found in more robust database management systems. Creating and updating databases by use of these functions is relatively slow because of data copies that occur upon hash collisions. These functions *are useful* for applications requiring fast lookup of relatively static information that is to be indexed by a single key.

The pointer to data that is returned from these functions are not aligned. This can cause problems if the block contains data that must be aligned to a specific boundry. If the block contains data that must be aligned, the block should be copied to an appropriately aligned area.

The **.pag** file will contain holes so that its apparent size is about four times its actual content. Some older UNIX systems create real file blocks for these holes when touched. These files cannot be copied by normal means (such as *cp*(1), *cat*(1), *tar*(1), or *ar*(1)) without expansion.

dptr pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 1024 bytes). Moreover, all key/content pairs that hash together must fit on a single block. **dbm_store** returns an error in the event that a disk block fills with inseparable data.

dbm_delete does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by **dbm_firstkey** and **dbm_nextkey** depends on a hashing function, not on anything interesting.

A **dbm_store** or **dbm_delete** during a pass through the keys by **dbm_firstkey** and **dbm_nextkey** may yield unexpected results.

AUTHOR

ndbm(3X) was developed by the University of California, Berkeley.

SEE ALSO

dbm(3X).

n

NAME

net_aton(), net_ntoa() - network station address string conversion routines

SYNOPSIS

```
#include <sys/netio.h>
char *net_aton(char *dstr, const char *sstr, int size);
char *net_ntoa(char *dstr, const char *sstr, int size);
```

MULTITHREAD USAGE

Thread Safe:	Yes
Cancel Safe:	Yes
Async-cancel Safe:	Yes
Async-signal Safe:	Yes

These functions can be called safely in a Multithreaded environment. They are not cancellation points.

DESCRIPTION

net_aton() and **net_ntoa()** translate station addresses between hexadecimal, octal or decimal, and binary formats:

net_aton() converts a hexadecimal, octal, or decimal address to a binary address.

net_ntoa() converts a binary address to an ASCII hexadecimal address.

Both routines are provided in the standard C library and are loaded automatically during compilation.

net_aton Parameters

The following parameters are used by **net_aton()**:

dstr Pointer to the binary address returned by the function.

sstr Pointer to a null-terminated ASCII form of a station address (Ethernet or IEEE 802.3). This address can be an octal, decimal, or hexadecimal number as used in the C language (in other words, a leading 0x or 0X implies hexadecimal; a leading 0 implies octal; otherwise, the number is interpreted as decimal).

size Length of the binary address to be returned in *dstr*. The length is 6 for Ethernet/IEEE 802.3 addresses.

net_ntoa Parameters

net_ntoa() converts a 48-bit binary station address to its ASCII hexadecimal equivalent. The following parameters are used by **net_ntoa()**:

dstr Pointer to the ASCII hexadecimal address returned by the function. *dstr* is null-terminated and padded with leading zeroes if necessary. *dstr* must be at least ($2 \times size + 3$) bytes long to accommodate the size of the converted address.

sstr Pointer to a station address in its binary form.

size Length of *sstr*.

RETURN VALUE

net_aton() and **net_ntoa()** return NULL if any error occurs.

EXAMPLES

```
#include <netio.h>
#define  destination_addr "0x00DD0002AD00"
     ...
struct fis arg;
char str[16];
     ...
(void) net_aton(arg.value.s, destination_addr, 6);
               /* arg.value.s = "<48-bit binary value>"  */
(void) net_ntoa(str, arg.value.s, 6);
               /* str = "0x00DD0002AD00" */
```

AUTHOR

net_aton() was developed by HP.

SEE ALSO

lan(7).

NAME

netdir(), netdir_getbyname(), netdir_getbyaddr(), netdir_free(), netdir_options(), taddr2uaddr(), uaddr2taddr(), netdir_perror(), netdir_sperror() - generic transport name-to-address translation

SYNOPSIS

```
#include <netdir.h>

int netdir_getbyname(const struct netconfig  *config,
    const struct nd_hostserv *service,
    struct nd_addrlist **addrs );

int netdir_getbyaddr(const struct netconfig  *config,
    struct nd_hostservlist **service,
    const struct netbuf  *netaddr );

void netdir_free(void *ptr,const int struct_type );

int netdir_options(const struct netconfig  *config,
    const int option,const int fildes,char  *point_to_args );

char *taddr2uaddr(const struct netconfig  *config,
    const struct netbuf  *addr );

struct netbuf *uaddr2taddr(const struct netconfig  *config,
    const char  *uaddr );

void netdir_perror(char *s );

char *netdir_sperror(void);
```

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

DESCRIPTION

These routines provide a generic interface for name-to-address mapping that will work with all transport protocols. This interface provides a generic way for programs to convert transport specific addresses into common structures and back again. The **netconfig** structure, described on the *netconfig*(4) manual page, identifies the transport.

The **netdir_getbyname()** routine maps the machine name and service name in the **nd_hostserv** structure to a collection of addresses of the type understood by the transport identified in the **netconfig** structure. This routine returns all addresses that are valid for that transport in the **nd_addrlist** structure. The **nd_hostserv** structure contains the following members:

```
char *h_host; /* host name */

char *h_serv; /* service name */
```

The **nd_addrlist** structure contains the following members:

```
int n_cnt; /* number of addresses */

struct netbuf *n_addrs;
```

netdir_getbyname() accepts some special-case host names. The host names are defined in **<netdir.h>**. The currently defined host names are:

HOST_SELF Represents the address to which local programs will bind their endpoints. HOST_SELF differs from the host name provided by **gethostname()** (see *gethostname*(2)), which represents the address to which *remote* programs will bind their

endpoints.

HOST_ANY Represents any host accessible by this transport provider. HOST_ANY allows applications to specify a required service without specifying a particular host name.

HOST_SELF_CONNECT
Represents the host address that can be used to connect to the local host.

HOST_BROADCAST
Represents the address for all hosts accessible by this transport provider. Network requests to this address will be received by all machines.

All fields of the **nd_hostserv** structure must be initialized.

To find the address of a given host and service on all available transports, call the **netdir_getbyname()** routine with each **struct netconfig** structure returned by **getnetconfig()**. (See *getnetconfig*(3N)).

The **netdir_getbyaddr()** routine maps addresses to service names. This routine returns *service*, a list of host and service pairs that would yield this address. If more than one tuple of host and service name is returned, then the first tuple contains the preferred host and service names:

```
struct nd_hostservlist {
     int *h_cnt; /* number of hostservs found */
     struct hostserv *h_hostservs;
};
```

The **netdir_free()** structure is used to free the structures allocated by the name to address translation routines. *ptr* points to the structure that has to be freed. The **struct_type** identifies the structure:

struct netbuf	**ND_ADDR**
struct nd_addrlist	**ND_ADDRLIST**
struct hostserv	**ND_HOSTSERV**
struct nd_hostservlist	**ND_HOSTSERVLIST**

The universal address returned by **taddr2uaddr()** should be freed by **free().**

The **netdir_options()** routine is used to do all transport-specific setups and option management. *fildes* is the associated file descriptor. *option*, *fildes*, and *pointer_to_args* are passed to the **netdir_options()** routine for the transport specified in **config**. Currently three values are defined for *option*:

ND_SET_BROADCAST
ND_SET_RESERVEDPORT
ND_CHECK_RESERVEDPORT

The **taddr2uaddr()** and **uaddr2taddr()** routines support translation between universal addresses and TLI type **netbufs**. The **taddr2uaddr()** routine takes a **struct netbuf** data structure and returns a pointer to a string that contains the universal address. It returns -1 **NULL** if the conversion is not possible. This is not a fatal condition as some transports may not suppose a universal address form.

uaddr2taddr() is the reverse of **taddr2uaddr()**. It returns the **struct netbuf** data structure for the given universal address.

If a transport provider does not support an option, **netdir_options()** returns -1 and the error message can be printed through **netdir_perror()** or **netdir_sperror()**.

The specific actions of each option follow.

ND_SET_BROADCAST Sets the transport provider up to allow broadcast, if the transport supports broadcast. *fildes* is a file descriptor into the transport, for example, the result of a **t_open** of **/dev/udp**). *pointer_to_args* is not used. If this completes, broadcast operations may be performed on file descriptor *fildes*.

ND_SET_RESERVEDPORT
Allows the application to bind to a reserved port, if that concept exists for the transport provider. *fildes* is an unbound file descriptor into the transport. If *pointer_to_args* is NULL, *fildes* will be bound to a reserved port. If *pointer_to_args* is a pointer to a *netbuf* structure, an attempt will be made to bind to any reserved port on the specified address.

ND_CHECK_RESERVEDPORT
: Used to verify that the address corresponds to a reserved port, if that concept exists for the transport provider. *fildes* is not used. *pointer_to_args* is a pointer to a **netbuf** structure that contains the address. This option returns 40 only if the address specified in *pointer_to_args* is reserved.

RETURN VALUES

The **netdir_perror()** routine prints an error message on the standard output stating why one of the name-to-address mapping routines failed. The error message is preceded by the string given as an argument.

The **netdir_sperror()** routine returns a string containing an error message stating why one of the name-to-address mapping routines failed.

netdir_sperror() returns a pointer to a buffer which contains the error message string. This buffer is overwritten on each call. In multithreaded applications, this buffer is implemented as thread-specific data.

SEE ALSO

gethostname(3C), getnetconfig(3N), getnetpath(3N), netconfig(4).

n

NAME

newpad, pnoutrefresh, prefresh — pad management functions

SYNOPSIS

```
#include <curses.h>

WINDOW *newpad(int nlines, int ncols);

int pnoutrefresh(WINDOW *pad, int pminrow, int pmincol, int sminrow,
                 int smincol, int smaxrow, int smaxcol);

int prefresh(WINDOW *pad, int pminrow, int pmincol, int sminrow,
             int smincol, int smaxrow, int smaxcol);
```

DESCRIPTION

The **newpad()** function creates a specialised **WINDOW** data structure representing a pad with *nlines* lines and *ncols* columns. A pad is like a window, except that it is not necessarily associated with a viewable part of the screen. Automatic refreshes of pads do not occur.

The **prefresh()** and **pnoutrefresh()** functions are analogous to **wrefresh()** and **wnoutrefresh()** except that they relate to pads instead of windows. The additional arguments indicate what part of the pad and screen are involved. The *pminrow* and *pmincol* arguments specify the origin of the rectangle to be displayed in the pad. The *sminrow*, *smincol*, *smaxrow* and *smaxcol* arguments specify the edges of the rectangle to be displayed on the screen. The lower right-hand corner of the rectangle to be displayed in the pad is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of *pminrow*, *pmincol*, *sminrow* or *smincol* are treated as if they were zero.

RETURN VALUE

Upon successful completion, the **newpad()** function returns a pointer to the pad data structure. Otherwise, it returns a null pointer.

Upon successful completion, **pnoutrefresh()** and **prefresh()** return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

To refresh a pad, call **prefresh()** or **pnoutrefresh()**, not **wrefresh()**. When porting code to use pads from WINDOWS, remember that these functions require additional arguments to specify the part of the pad to be displayed and the location on the screen to be used for the display.

Although a subwindow and its parent pad may share memory representing characters in the pad, they need not share status information about what has changed in the pad. Therefore, after modifying a subwindow within a pad, it may be necessary to call **touchwin()** or **touchline()** on the pad before calling **prefresh()**.

SEE ALSO

derwin(3X), doupdate(3X), is_linetouched(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4

The **pnoutrefresh()** and **prefresh()** functions are merged with this entry. In previous issues, they appeared in the entry for **prefresh()**.

NAME

newwin, subwin — window creation functions

SYNOPSIS

```
#include <curses.h>

WINDOW *newwin(int nlines, int ncols, int begin_y, int begin_x);

WINDOW *subwin(WINDOW *orig, int nlines, int ncols, int begin_y,
               int begin_x);
```

DESCRIPTION

The **newwin()** function creates a new window with *nlines* lines and *ncols* columns, positioned so that the origin is (*begin_y*, *begin_x*). If *nlines* is zero, it defaults to **LINES** – *begin_y*; if *ncols* is zero, it defaults to **COLS** – *begin_x*.

The **subwin()** function creates a new window with *nlines* lines and *ncols* columns, positioned so that the origin is at (*begin_y*, *begin_x*). (This position is an absolute screen position, not a position relative to the window *orig*.) If any part of the new window is outside *orig*, the function fails and the window is not created.

RETURN VALUE

Upon successful completion, these functions return a pointer to the new window. Otherwise, they return a null pointer.

ERRORS

No errors are defined.

APPLICATION USAGE

Before performing the first refresh of a subwindow, portable applications should call **touchwin()** or **touchline()** on the parent window.

Each window maintains internal descriptions of the screen image and status. The screen image is shared among all windows in the window hierarchy. Refresh operations rely on information on what has changed within a window, which is private to each window. Refreshing a window, when updates were made to a different window, may fail to perform needed updates because the windows do not share this information.

A new full-screen window is created by calling:

```
newwin(0, 0, 0, 0);
```

SEE ALSO

delwin(3X), is_linetouched(3X), doupdate(3X), <curses.h>.

CHANGE HISTORY

First released in Issue 2

X/Open Curses, Issue 4

The entry is rewritten for clarity.

NAME

nextafter(), nextafterf() - next representable floating-point value

SYNOPSIS

```
#include <math.h>
double nextafter(double x, double y);
float nextafterf(float x, float y);
```

DESCRIPTION

The **nextafter()** function computes the next representable double-precision floating-point value following *x* in the direction of *y*. Thus, if *y* is less than *x*, **nextafter()** returns the largest representable floating-point number less than *x*.

The **nextafter()** function is recommended by the IEEE-754 standard for floating-point arithmetic. The ISO/ANSI C committee has approved the **nextafter()** function for inclusion in the C9X draft standard.

nextafterf() is a **float** version of **nextafter()**; it takes **float** arguments and returns a **float** result.

To use these functions, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

The **nextafter()** and **nextafterf()** functions return the next representable double-precision or single-precision floating-point value following *x* in the direction of *y*.

If *x* or *y* is NaN, **nextafter()** returns NaN.

If *x* is finite and the correct function value would overflow, **nextafter()** returns ±**HUGE_VAL** (according to the sign of *x*) and sets **errno** to [ERANGE].

ERRORS

If **nextafter()** fails, **errno** is set to the following value.

[ERANGE] The correct value would overflow.

SEE ALSO

math(5), limits(5).

STANDARDS CONFORMANCE

nextafter(): SVID3, XPG4.2

NAME

nis_db, db_initialize, db_create_table, db_destroy_table, db_first_entry, db_next_entry, db_reset_next_entry, db_list_entries, db_remove_entry, db_add_entry, db_table_exists, db_unload_table, db_checkpoint, db_standby, db_free_result - NIS+ Database access functions

SYNOPSIS

cc [*flag*...] *file*... **-lnisdb -lnsl** [*library*...]

```
#include <rpcsvc/nis.h>
#include <rpcsvc/nis_db.h>

bool db_initialize(const char *dictionary_pathname);

db_status db_create_table(const char *table_name,
    const table_obj *table);

db_status db_destroy_table(const char *table_name);

db_result *db_first_entry(const char *table_name,
    const int numattrs,
    const nis_attr *attrs);

db_result *db_next_entry(const char *table_name,
    const db_next_desc *next_handle);

db_result *db_reset_next_entry(const char *table_name,
    const db_next_desc *next_handle);

db_result *db_list_entries(const char *table_name,
    const int numattrs,
    const nis_attr *attrs);

db_result *db_remove_entry(const char *table_name,
    const int numattrs,
    const nis_attr *attrs);

db_result *db_add_entry(const char *table_name,
    const int numattrs,
    const nis_attr *attrs,
    const entry_obj *entry);

db_status db_table_exists(const char *table_name);

db_status db_unload_table(const char *table_name);

db_status db_checkpoint(const char *table_name);

db_status db_standby(const char *table_name );

void db_free_result(db_result *);
```

DESCRIPTION

These functions describe the interface between the NIS+ server and the underlying database. They are defined in the shared library **/usr/lib/libnisdb.so**.

The interface is a simple subset of a complete relational database and provides just those items that are needed by the NIS+ server daemon. When you replace the database, your interface routines should match these exactly. Also note that the database is responsible for verifying that the objects passed do not exceed the internal limits of the database being used.

The database's performance will directly affect the performance of the server. The default information base that is provided with NIS+ is the Structured Storage Manager (SSM). This is a memory based database that has been tuned for NIS+.

These routines should not be invoked by any NIS+ client. NIS+ clients should use the NIS+ tables API described in *nis_tables*(3N).

These routines only use the **table_obj**, **entry_obj** and the **nis_attr** structures defined in **<rpcsvc/nis.h>**. The NIS+ directory is itself stored in a table by the service daemon. This table has two columns, one searchable with the name of the object in it, the other non-searchable with binary XDRed data in it. The NIS+ server converts directory lookup requests in the namespace into table searches. The table it searches in response to these requests will have the same name as the directory of the name it is

searching for.

The structure returned by the DB access routines is defined as:

```
enum db_status {DB_SUCCESS, DB_NOTFOUND, DB_NOTUNIQUE, DB_BADTABLE,
        DB_BADQUERY, DB_BADOBJECT, DB_MEMORY_LIMIT, DB_STORAGE_LIMIT,
        DB_INTERNAL_ERROR };

struct db_result {
        db_status       status;                 /* Result status */
        db_next_desc    nextinfo;               /* descriptor */
        struct {
                u_int        objects_len;
                entry_obj    *objects_val;
        } objects;              /* A variable list of objects */
        long    ticks;          /* execution time in microseconds */
};
```

For a complete description of NIS+ objects, see *nis_objects*(3N).

The structure **db_next_desc** should be used as an opaque handle for **db_next_entry( )** and **db_reset_next_entry( )**.

The **nis_attr** structure used in **db_first_entry** and other related functions is defined as follows:

```
struct nis_attr {
        char    *zattr_ndx;
        struct {
                u_int   zattr_val_len;
                char    *zattr_val_val;
        } zattr_val;
};
```

zattr_ndx is the name of the attribute. **zattr_val_len** is the value of the attribute **zattr_val_val**.

In **db_result**, the *objects* array contains objects if and only if the result returned in the *status* variable is **DB_SUCCESS**. A null pointer, instead of a pointer to a **db_result** structure, is returned if there is insufficient memory to create the structure.

db_initialize( ) is called prior to any interaction with the database. It takes as argument the pathname of the file that contains, or will contain, catalog information associated with the database.

db_create_table( ) creates a new table using the given table name and the table object. It returns TRUE if the table was successfully created; FALSE otherwise.

db_destroy_table( ) destroys the table of the given name. It returns TRUE if the destruction was successful; FALSE otherwise.

db_first_entry( ) returns a copy of the first entry in the specified table that satisfies the given attributes. If no attributes are supplied, a copy of the first entry in the table is returned. **attrs** is an array of **nis_attr** structure with *numattrs* number of elements. The returned structure, **db_result**, contains a structure, **db_next_desc**, to be used as an argument to **db_next_entry( )** or **db_reset_next_entry( )**. **db_next_desc** should be used only as an opaque handle. **db_free_result( )** can be used to free up the returned **db_result** structure.

db_next_entry( ) returns a copy of the next entry as indicated by the *next_handle*. An initial call to **db_first_entry( )**, followed by a sequence of calls to **db_next_entry( )**, can be used to successfully obtain entries of an entire table or entries that satisfy the attributes supplied to **db_first_entry( )**. **db_free_result( )** can be used to free up the returned **db_result** structure.

db_reset_next_entry( ) terminates the **db_first_entry( )**/**db_next_entry( )** sequence as indicated by *next_handle*, freeing any resources that have been used to maintain the sequence. After a call to **db_reset_next_entry( )**, a call to **db_next_entry( )** using the same *next_handle* would fail, returning a **DB_BADQUERY** reply. **db_free_result( )** can be used to free up the returned **db_result** structure.

db_list_entries( ) returns copies of entries that satisfy the given attributes. **db_free_result( )** can be used to free up the returned **db_result** structure. **attrs** is an array of

nis_attr structure with *numattrs* number of elements.

db_remove_entry() removes all entries that satisfy the given attributes. **db_free_result()** can be used to free up the returned **db_result** structure. **attrs** is an array of **nis_attr** structure with *numattrs* number of elements.

db_add_entry() adds a copy of the given object to the specified table, replacing the one identified by the given attributes. If the given attributes identify more than one object, **DB_NOTUNIQUE** is returned. If no object is identified by the given attributes, the object is added. **attrs** is an array of **nis_attr** structure with *numattrs* number of elements. **db_free_result()** can be used to free up the returned **db_result** structure.

db_table_exists() provides an efficient way for the NIS+ service to detect that a table exists. This increases response time to the client and lowers the load on the server.

db_unload_table() is used by the service to unload or deactivate tables that are not currently being used. The service internally keeps track of access patterns to tables and will unload those tables that have not been accessed for a while. By unloading infrequently accessed tables, the service can minimize the amount of system resources for efficient operation.

db_checkpoint() organizes the contents of the table in a more efficient manner. Checkpointing may mean different things to different types of databases. It does not affect the logical contents of the table — operations and queries should return the same result before and after a checkpoint. For example, in a log-based system, checkpointing may mean incorporating log entries of updates accumulated since the previous checkpoint into the table.

db_free_result() frees up the space allocated by various functions listed on this manual page that return a **db_result** structure.

db_standby() is an advisory call to the database manager. This call informs the database that activity has slowed down and it can free up unnecessary resources such as file descriptors.

MULTITHREAD USAGE

Thread Safe: **No**
Cancel Safe: **No**
Fork Safe: **No**
Async-cancel Safe: **No**
Async-signal Safe: **No**

These functions can not be called safely in a multithreaded environment.

PROGRAMMING

Most of the routines in this library use an NIS+ *name* to identify the object that the user desires. The name must be in canonical form before being passed to the database because one server may be serving several namespaces and discrimination of the requested objects is accomplished by comparing the domain names.

DIAGNOSTICS

DB_SUCCESS The query or operation completed successfully and returned status.

DB_NOTFOUND The name or entry that was named in the argument did not exist.

DB_NOTUNIQUE An attempt was made to remove an entry from a table that is not uniquely specified.

DB_BADQUERY The query that was submitted to the database was invalid (for example, it might name some nonexistent fields).

DB_BADTABLE The table was corrupted.

DB_BADOBJECT The fields of the object does not conform to the fields of the table to which it is being added.

DB_MEMORY_LIMIT
There is insufficient memory to complete the operation requested.

DB_STORAGE_LIMIT
There is insufficient file storage available to complete the operation requested.

DB_INTERNAL_ERROR
An internal error was encountered during the execution of the operation requested

(either a programming error or an unrecoverable exception).

SEE ALSO

rpc.nisd(1M), nis_objects(3N), nisfiles(4).

NAME

nis_error, nis_sperrno, nis_perror, nis_lerror, nis_sperror, nis_sperror_r - display NIS+ error messages

SYNOPSIS

```
cc [ flag... ] file... -lnsl [ library... ]
#include <rpcsvc/nis.h>
char *nis_sperrno(const nis_error status);
void nis_perror(const nis_error status,
    const char *label);
void nis_lerror(const nis_error status,
    const char *label);
char *nis_sperror_r(nis_error status,
    char *label,
    char * buf);
char *nis_sperror(const nis_error status,
    const char *label);
```

DESCRIPTION

These functions convert NIS+ status values into text strings.

nis_sperrno() simply returns a pointer to a string constant which is the error string.

nis_perror() prints the error message corresponding to *status* as "*label*: **error message**" on standard error.

nis_lerror() sends the error text to *syslog*(3C) at level LOG_ERR.

The function **nis_sperror_r()**, returns a pointer to a string that can be used or copied using the **strdup()** function (see *string*(3C).) The caller must supply a string buffer, **buf**, large enough to hold the error string (a buffer size of 128 bytes is guaranteed to be sufficiently large).

The last function, **nis_sperror()**, is similar to **nis_sperror_r()** except that the string is returned as a pointer to a buffer that is reused on each call. **nis_sperror_r()** is the preferred interface, since it is suitable for single-threaded and multithreaded programs.

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. These functions may be thread cancellation points because they invoke functions that are thread cancellation points.

In a multithreaded environment, these functions are not safe to be called by a child process after fork() and before exec(). These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

SEE ALSO

niserror(1), string(3C), syslog(3C).

NAME

nis_groups, nis_ismember, nis_addmember, nis_removemember, nis_creategroup, nis_destroygroup, nis_verifygroup, nis_print_group_entry, nis_map_group, __nis_map_group - NIS+ group manipulation functions

SYNOPSIS

cc [*flag*...] *file*... **-lnsl** [*library*...]

```
#include <rpcsvc/nis.h>
bool_t nis_ismember(const nis_name principal, const nis_name group);
nis_error nis_addmember(const nis_name member, const nis_name group);
nis_error nis_removemember(const nis_name member, const nis_name group);
nis_error nis_creategroup(const nis_name group, const u_long flags);
nis_error nis_destroygroup(const nis_name group);
void nis_print_group_entry(const nis_name group);
nis_error nis_verifygroup(const nis_name group);
```

DESCRIPTION

These functions manipulate NIS+ groups. They are used by NIS+ clients and servers, and are the interfaces to the group authorization object.

The names of NIS+ groups are syntactically similar to names of NIS+ objects but they occupy a separate namespace. A group named "a.b.c.d." is represented by a NIS+ group object named "a.groups_dir.b.c.d."; the functions described here all expect the name of the group, not the name of the corresponding group object.

There are three types of group members:

- An *explicit* member is just a NIS+ principal-name, for example "wickedwitch.west.oz."
- An *implicit* ("domain") member, written "*.west.oz.", means that all principals in the given domain belong to this member. No other forms of wildcarding are allowed: "wickedwitch.*.oz." is invalid, as is "wickedwitch.west.*.". Note that principals in subdomains of the given domain are *not* included.
- A *recursive* ("group") member, written "@cowards.oz.", refers to another group; all principals that belong to that group are considered to belong here.

Any member may be made *negative* by prefixing it with a minus sign ('-'). A group may thus contain explicit, implicit, recursive, negative explicit, negative implicit, and negative recursive members.

A principal is considered to belong to a group if it belongs to at least one non-negative group member of the group and belongs to no negative group members.

The **nis_ismember()** function returns TRUE if it can establish that *principal* belongs to *group;* otherwise it returns FALSE.

The **nis_addmember()** and **nis_removemember()** functions add or remove a member. They do not check whether the member is valid. The user must have read and modify rights for the group in question.

The **nis_creategroup()** and **nis_destroygroup()** functions create and destroy group objects. The user must have create or destroy rights, respectively, for the *groups_dir* directory in the appropriate domain. The parameter *flags* to **nis_creategroup()** is currently unused and should be set to zero.

The **nis_print_group_entry()** function lists a group's members on the standard output.

The **nis_verifygroup()** function returns **NIS_SUCCESS** if the given group exists, otherwise it returns an error code.

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**

Async-signal Safe: No

These functions can be called safely in a multithreaded environment. These functions may be thread cancellation points because they invoke functions that are thread cancellation points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

EXAMPLES

Simple Memberships

Given a group **sadsouls.oz.** with members **tinman.oz.**, **lion.oz.**, and **scarecrow.oz.**, the function call

```
bool_var = nis_ismember("lion.oz.", "sadsouls.oz.");
```

will return 1 (TRUE) and the function call

```
bool_var = nis_ismember("toto.oz.", "sadsouls.oz.");
```

will return 0 (FALSE).

Implicit Memberships

Given a group **baddies.oz.**, with members **wickedwitch.west.oz.** and ***.monkeys.west.oz.**, the function call

```
bool_var = nis_ismember("hulk.monkeys.west.oz.", "baddies.oz.");
```

will return 1 (TRUE) because any principal from the **monkeys.west.oz.** domain belongs to the implicit group ***.monkeys.west.oz.**, but the function call

```
bool_var = nis_ismember("hulk.big.monkeys.west.oz.", "baddies.oz.");
```

will return 0 (FALSE).

Recursive Memberships

Given a group **goodandbad.oz.**, with members **toto.kansas**, **@sadsouls.oz.**, and **@baddies.oz.**, and the groups **sadsouls.oz.** and **baddies.oz.** defined above, the function call

```
bool_var = nis_ismember("wickedwitch.west.oz.", "goodandbad.oz.");
```

will return 1 (TRUE), because **wickedwitch.west.oz.** is a member of the **baddies.oz.** group which is recursively included in the **goodandbad.oz.** group.

NOTES

These functions only accept fully-qualified NIS+ names.

A group is represented by a NIS+ object (see *nis_objects*(3N)) with a variant part that is defined in the **group_obj** structure. It contains the following fields:

```
u_long            gr_flags;       /* Interpretation Flags
                                     (currently unused) */
struct {
        u_int     gr_members_len;
        nis_name  *gr_members_val;
} gr_members;                     /* Array of members */
```

NIS+ servers and clients maintain a local cache of expanded groups to enhance their performance when checking for group membership. Should the membership of a group change, servers and clients with that group cached will not see the change until either the group cache has expired or it is explicitly flushed. A server's cache may be flushed programmatically by calling the **nis_servstate()** function with tag **TAG_GCACHE** and a value of 1.

There are currently no known methods for **nis_ismember()**, **nis_print_group_entry()**, and **nis_verifygroup()** to get their answers from only the master server.

SEE ALSO

nisgrpadm(1), nis_objects(3N).

NAME

nis_local_names, nis_local_directory, nis_local_host, nis_local_group, nis_local_principal - NIS+ local names

SYNOPSIS

cc [*flag*...] *file*... **-lnsl** [*library*...]

```
#include <rpcsvc/nis.h>
nis_name nis_local_directory(void);
nis_name nis_local_host(void);
nis_name nis_local_group(void);
nis_name nis_local_principal(void);
```

DESCRIPTION

These functions return several default NIS+ names associated with the current process.

nis_local_directory() returns the name of the NIS+ domain for this machine. This is currently the same as the Secure RPC domain returned by the **sysinfo()** system call.

nis_local_host() returns the NIS+ name of the current machine. This is the fully qualified name for the host and is either the value returned by the *gethostname*(2) function or, if the host name is only partially qualified, the concatenation of that value and the name of the NIS+ directory. Note that if a machine's name and address cannot be found in the local NIS+ directory, its hostname must be fully qualified.

nis_local_group() returns the name of the current NIS+ group name. This is currently set by setting the environment variable **NIS_GROUP** to the groupname.

nis_local_principal() returns the NIS+ principal name for the user associated with the effective UID of the calling process. This function maps the effective uid into a principal name by looking for a LOCAL type credential (see *nisaddcred*(1M)) in the table named *cred.org_dir* in the default domain.

Note: The result returned by these routines is a pointer to a data structure with the NIS+ library, and should be considered a read-only result and should not be modified.

n

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. These functions may be thread cancellation points because they invoke functions that are thread cancellation points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

ENVIRONMENT

NIS_GROUP This variable contains the name of the local NIS+ group. If the name is not fully qualified, the value returned by **nis_local_directory()** will be concatenated to it.

SEE ALSO

nisdefaults(1), nisaddcred(1M), gethostname(2), nis_names(3N), nis_objects(3N).

NAME

nis_names, nis_lookup, nis_add, nis_remove, nis_modify, nis_freeresult - NIS+ namespace functions

SYNOPSIS

cc [*flag*...] *file*... **-lnsl** [*library*...]

```
#include <rpcsvc/nis.h>
nis_result *nis_lookup(const nis_name name, const u_long flags);
nis_result *nis_add(const nis_name name, const nis_object *obj);
nis_result *nis_remove(const nis_name name, const nis_object *obj);
nis_result *nis_modify(const nis_name name, const nis_object *obj);
void nis_freeresult(nis_result *result);
```

DESCRIPTION

These functions are used to locate and manipulate all NIS+ objects (see *nis_objects*(3N)) except the NIS+ entry objects. To look up the NIS+ entry objects within a NIS+ table, refer to *nis_subr*(3N).

nis_lookup() resolves a NIS+ name and returns a copy of that object from a NIS+ server. **nis_add()** and **nis_remove()** add and remove objects to the NIS+ namespace, respectively. **nis_modify()** can change specific attributes of an object that already exists in the namespace.

These functions should be used only with names that refer to an NIS+ Directory, NIS+ Table, NIS+ Group, or NIS+ Private object. If a name refers to an NIS+ entry object, the functions listed in *nis_subr*(3N) should be used.

nis_freeresult() frees all memory associated with a **nis_result** structure. This function must be called to free the memory associated with a NIS+ result. **nis_lookup()**, **nis_add()**, **nis_remove()**, and **nis_modify()** all return a pointer to a **nis_result** structure which *must* be freed by calling **nis_freeresult()** when you have finished using it. If one or more of the objects returned in the structure need to be retained, they can be copied with *nis_clone_object*(3N) (see *nis_subr*(3N)).

nis_lookup() takes two parameters, the name of the object to be resolved in *name*, and a flags parameter, *flags*, which is defined below. The object name is expected to correspond to the syntax of a non-indexed NIS+ name (see *nis_tables*(3N)). The **nis_lookup()** function is the *only* function from this group that can use a non-fully qualified name. If the parameter *name* is not a fully qualified name, then the flag **EXPAND_NAME** *must* be specified in the call. If this flag is not specified, the function will fail with the error NIS_BADNAME.

The *flags* parameter is constructed by logically ORing zero or more flags from the following list.

FOLLOW_LINKS When specified, the client library will "follow" links by issuing another NIS+ lookup call for the object named by the link. If the linked object is itself a link, then this process will iterate until the either a object is found that is not a *LINK* type object, or the library has followed 16 links.

HARD_LOOKUP When specified, the client library will retry the lookup until it is answered by a server. Using this flag will cause the library to block until at least one NIS+ server is available. If the network connectivity is impaired, this can be a relatively long time.

NO_CACHE When specified, the client library will bypass any object caches and will get the object from either the master NIS+ server or one of its replicas.

MASTER_ONLY When specified, the client library will bypass any object caches and any domain replicas and fetch the object from the NIS+ master server for the object's domain. This insures that the object returned is up to date at the cost of a possible performance degradation and failure if the master server is unavailable or physically distant.

EXPAND_NAME When specified, the client library will attempt to expand a partially qualified name by calling the function **nis_getnames()** (see *nis_subr*(3N)) which uses the environment variable **NIS_PATH**.

The status value may be translated to ascii text using the function **nis_sperrno()** (see *nis_error*(3N)).

On return, the *objects* array in the result will contain one and possibly several objects that were resolved by the request. If the FOLLOW_LINKS flag was present, on success the function could return several entry objects if the link in question pointed within a table. If an error occurred when following a link, the objects

array will contain a copy of the link object itself.

The function **nis_add()** will take the object *obj* and add it to the NIS+ namespace with the name *name*. This operation will fail if the client making the request does not have the *create* access right for the domain in which this object will be added. The parameter *name* must contain a fully qualified NIS+ name. The object members *zo_name* and *zo_domain* will be constructed from this name. This operation will fail if the object already exists. This feature prevents the accidental addition of objects over another object that has been added by another process.

The function **nis_remove()** will remove the object with name *name* from the NIS+ namespace. The client making this request must have the *destroy* access right for the domain in which this object resides. If the named object is a link, the link is removed and *not* the object that it points to. If the parameter *obj* is not NULL, it is assumed to point to a copy of the object being removed. In this case, if the object on the server does not have the same object identifier as the object being passed, the operation will fail with the NIS_NOTSAMEOBJ error. This feature allows the client to insure that it is removing the desired object. The parameter *name* must contain a fully qualified NIS+ name.

The function **nis_modify()** will modify the object named by *name* to the field values in the object pointed to by *obj*. This object should contain a copy of the object from the name space that is being modified. This operation will fail with the error NIS_NOTSAMEOBJ if the object identifier of the passed object does not match that of the object being modified in the namespace.

Note: Normally the contents of the member *zo_name* in the *nis_object* structure would be constructed from the name passed in the *name* parameter. However, if it is non-NULL the client library will use the name in the *zo_name* member to perform a rename operation on the object. This name *must not* contain any unquoted '.'(dot) characters. If these conditions are not met the operation will fail and return the NIS_BADNAME error code.

Results

These functions return a pointer to a structure of type **nis_result**:

```
struct nis_result {
     nis_error status;
     struct {
               u_int objects_len;
               nis_object *objects_val;
     } objects;
     netobj    cookie;
     u_long    zticks;
     u_long    dticks;
     u_long    aticks;
     u_long    cticks;
};
```

The *status* member contains the error status of the the operation. A text message that describes the error can be obtained by calling the function **nis_sperrno()** (see *nis_error*(3N)).

The *objects* structure contains two members. *objects_val* is an array of *nis_object* structures; *objects_len* is the number of cells in the array. These objects will be freed by the call to **nis_freeresult()**. If you need to keep a copy of one or more objects, they can be copied with the function **nis_clone_object()** and freed with the function **nis_destroy_object()** (see *nis_server*(3N)). Refer to *nis_objects*(3N) for a description of the **nis_object** structure.

The various ticks contain details of where the time was taken during a request. They can be used to tune one's data organization for faster access and to compare different database implementations (see *nis_db*(3N)).

zticks The time spent in the NIS+ service itself. This count starts when the server receives the request and stops when it sends the reply.

dticks The time spent in the database backend. This time is measured from the time a database call starts, until the result is returned. If the request results in multiple calls to the database, this is the sum of all the time spent in those calls.

aticks The time spent in any "accelerators" or caches. This includes the time required to locate the server needed to resolve the request.

cticks The total time spent in the request. This clock starts when you enter the client library and stops when a result is returned. By subtracting the sum of the other ticks values from this value, you can obtain the local overhead of generating a NIS+ request.

Subtracting the value in *dticks* from the value in *zticks* will yield the time spent in the service code itself. Subtracting the sum of the values in *zticks* and *aticks* from the value in *cticks* will yield the time spent in the client library itself. Note: all of the tick times are measured in microseconds.

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. These functions may be thread cancellation points because they invoke functions that are thread cancellation points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

RETURN VALUES

The client library can return a variety of error returns and diagnostics. The more salient ones are documented below.

NIS_SUCCESS The request was successful.

NIS_S_SUCCESS The request was successful, however the object returned came from an object cache and not directly from the server. If you do not wish to see objects from object caches you must specify the flag **NO_CACHE** when you call the lookup function.

NIS_NOTFOUND The named object does not exist in the namespace.

NIS_CACHEEXPIRED The object returned came from an object cache that has *expired*. The time to live value has gone to zero and the object may have changed. If the flag **NO_CACHE** was passed to the lookup function then the lookup function will retry the operation to get an unexpired copy of the object.

NIS_NAMEUNREACHABLE
A server for the directory of the named object could not be reached. This can occur when there is a network partition or all servers have crashed. See the **HARD_LOOKUP** flag.

NIS_UNKNOWNOBJ The object returned is of an unknown type.

NIS_TRYAGAIN The server connected to was too busy to handle your request. For the *add*, *remove*, and *modify* operations this is returned when either the master server for a directory is unavailable or it is in the process of checkpointing its database. It can also be returned when the server is updating its internal state. And in the case of **nis_list()** if the client specifies a callback and the server does not have enough resources to handle the callback.

NIS_SYSTEMERROR A generic system error occurred while attempting the request. Most commonly the server has crashed or the database has become corrupted. Check the syslog record for error messages from the server.

NIS_NOT_ME A request was made to a server that does not serve the name in question. Normally this will not occur, however if you are not using the built in location mechanism for servers you may see this if your mechanism is broken.

NIS_NOMEMORY Generally a fatal result. It means that the service ran out of heap space.

NIS_NAMEEXISTS An attempt was made to add a name that already exists. To add the name, first remove the existing name and then add the new object or modify the existing named object.

NIS_NOTMASTER An attempt was made to update the database on a replica server.

NIS_INVALIDOBJ The object pointed to by *obj* is not a valid NIS+ object.

NIS_BADNAME The name passed to the function is not a legal NIS+ name.

NIS_LINKNAMEERROR The name passed resolved to a *LINK* type object and the contents of the link pointed to an invalid name.

NIS_NOTSAMEOBJ An attempt to remove an object from the namespace was aborted because the object that would have been removed was not the same object that was passed in the request.

NIS_NOSUCHNAME This hard error indicates that the named directory of the table object does not exist. This occurs when the server that should be the parent of the server that serves the table, does not know about the directory in which the table resides.

NIS_NOSUCHTABLE The named table does not exist.

NIS_MODFAIL The attempted modification failed.

NIS_FOREIGNNS The name could not be completely resolved. When the name passed to the function would resolve in a namespace that is outside the NIS+ name tree, this error is returned with a NIS+ object of type **DIRECTORY**, which contains the type of namespace and contact information for a server within that namespace.

NIS_RPCERROR This fatal error indicates the RPC subsystem failed in some way. Generally there will be a *syslog*(3C) message indicating why the RPC request failed.

ENVIRONMENT

NIS_PATH If the flag **EXPAND_NAME** is set, this variable is the search path used by **nis_lookup()**.

NOTES

You cannot modify the name of an object if that modification would cause the object to reside in a different domain.

You cannot modify the schema of a table object.

SEE ALSO

nis_error(3N), nis_objects(3N), nis_server(3N), nis_subr(3N), nis_tables(3N).

n

NAME

nis_objects - NIS+ object formats

SYNOPSIS

cc [*flag*...] *file*... **-lnsl** [*library*...]

/usr/include/rpcsvc/nis_objects.h

DESCRIPTION

Common Attributes

The NIS+ service uses a variant record structure to hold the contents of the objects that are used by the NIS+ service. These objects all share a common structure which defines a set of attributes that all objects possess. The **nis_object** structure contains the following members:

```
typedef char            *nis_name;
struct  nis_object {
        nis_oid         zo_oid;
        nis_name        zo_name;
        nis_name        zo_owner;
        nis_name        zo_group;
        nis_name        zo_domain;
        u_long          zo_access;
        u_long          zo_ttl;
        objdata         zo_data;
};
```

In this structure, the first member **zo_oid**, is a 64 bit number that uniquely identifies this instance of the object on this server. This member is filled in by the server when the object is created and changed by the server when the object is modified. When used in conjunction with the object's name and domain it uniquely identifies the object in the entire NIS+ namespace.

The second member, **zo_name**, contains the leaf name of the object. This name is *never* terminated with a '.' (dot). When an object is created or added to the namespace, the client library will automatically fill in this field and the domain name from the name that was passed to the function.

zo_domain contains the name of the NIS+ domain to which this object belongs. This information is useful when tracking the parentage of an object from a cache. When used in conjunction with the members **zo_name** and **zo_oid**, it uniquely identifies an object. This makes it possible to always reconstruct the name of an object by using the code fragment

```
sprintf(buf,"%s.%s", obj→zo_name, obj→zo_domain);
```

The **zo_owner** and **zo_group** members contain the NIS+ names of the object's principal owner and group owner, respectively. Both names *must be* NIS+ fully qualified names. However, neither name can be used directly to identify the object they represent. This stems from the condition that NIS+ uses itself to store information that it exports.

The **zo_owner** member contains a fully qualified NIS+ name of the form *principal.domain*. This name is called a NIS+ principal name and is used to identify authentication information in a credential table. When the server constructs a search query of the form

[cname=*principal***],cred.org_dir.***domain***.**

The query will return to the server credential information about *principal* for all flavors of RPC authentication that are in use by that principal. When an RPC request is made to the server, the authentication flavor is extracted from the request and is used to find out the NIS+ principal name of the client. For example, if the client is using the AUTH_DES authentication flavor, it will include in the authentication credentials the network name or *netname* of the user making the request. This netname will be of the form

unix.*UID@domain*

The NIS+ server will then construct a query on the credential database of the form

[auth_name=*netname***,auth_type=AUTH_DES],cred.org_dir.***domain***.**

This query will return an entry which contains a principal name in the first column. This NIS+ principal name is used to control access to NIS+ objects.

The group owner for the object is treated differently. The group owner member is optional (it should be the null string if not present) but must be fully qualified if present. A group name takes the form

group.domain.

which the server then maps into a name of the form

group **.groups_dir.** *domain.*

The purpose of this mapping is to prevent NIS+ group names from conflicting with user specified domain or table names. For example, if a domain was called *engineering.foo.com.*, then without the mapping a NIS+ group of the same name to represent members of engineering would not be possible. The contents of groups are lists of NIS+ principal names which are used exactly like the **zo_owner** name in the object. See *nis_groups*(3N) for more details.

The **zo_access** member contains the bitmask of access rights assigned to this object. There are four access rights defined, and four are reserved for future use and must be zero. This group of 8 access rights can be granted to four categories of client. These categories are the object's owner, the object's group owner, all authenticated clients (world), and all unauthenticated clients (nobody). Note that access granted to "nobody" is really access granted to everyone, authenticated and unauthenticated clients.

The **zo_ttl** member contains the number of seconds that the object can "live" in a cache before it is expired. This value is called the time to live (ttl) for this object. This number is particularly important on group and directory (domain) objects. When an object is cached, the current time is added to the value in **zo_ttl**. Then each time the cached object is used, the time in **zo_ttl** is compared with the current time. If the current time is later than the time in **zo_ttl** the object is said to have expired and the cached copy should not be used.

Setting the ttl is somewhat of an art. You can think of it as the "half life" of the object, or half the amount of time you believe will pass before the object changes. The benefit of setting the ttl to a large number is that the object will stay in a cache for long periods of time. The problem with setting it to a large value is that when the object changes it will take a long time for the caches to flush out old copies of that object. The problems and benefits are reversed for setting the time to a small value. Generally setting the value to 43200 (12 hrs) is reasonable for things that change day to day, and 3024000 is good for things that change week to week. Setting the value to 0 will prevent the object from ever being cached since it would expire immediately.

The **zo_data** member is a discriminated union with the following members:

```
zotypes zo_type;
union {
        struct directory_obj    di_data;
        struct group_obj        gr_data;
        struct table_obj        ta_data;
        struct entry_obj        en_data;
        struct link_obj         li_data;
        struct {
                  u_int         po_data_len;
                  char          *po_data_val;
        } po_data;
} objdata_u;
```

The union is discriminated based on the type value contained in **zo_type**. There six types of objects currently defined in the NIS+ service. These types are the directory, link, group, table, entry, and private types.

```
enum zotypes {
        BOGUS_OBJ       = 0,
        NO_OBJ          = 1,
        DIRECTORY_OBJ   = 2,
        GROUP_OBJ       = 3,
        TABLE_OBJ       = 4,
        ENTRY_OBJ       = 5,
        LINK_OBJ        = 6,
        PRIVATE_OBJ     = 7
};
typedef enum zotypes zotypes;
```

All object types define a structure that contains data specific to that type of object. The simplest are private objects which are defined to contain a variable length array of octets. Only the owner of the object is expected to understand the contents of a private object. The following section describe the other five object

types in more significant detail.

Directory Objects

The first type of object is the *directory* object. This object's variant part is defined as follows:

```
enum nstype {
        UNKNOWN = 0,
        NIS     = 1,
        SUNYP   = 2,
        DNS     = 4,
        X500    = 5,
        DNANS   = 6,
        XCHS    = 7,
}
typedef enum nstype nstype;

struct oar_mask {
      u_long      oa_rights;
      zotypes     oa_otype;
}
typedef struct oar_mask oar_mask;

struct endpoint {
      char  *uaddr;
      char  *family;
      char  *proto;
}
typedef struct endpoint endpoint;

struct nis_server {
      nis_name    name;
      struct {
                  u_int           ep_len;
                  endpoint        *ep_val;
      } ep;
      u_long      key_type;
      netobj      pkey;
}
typedef struct nis_server nis_server;

struct directory_obj {
          nis_name        do_name;
          nstype          do_type;
          struct {
                  u_int           do_servers_len;
                  nis_server      *do_servers_val;
      } do_servers;
      u_long      do_ttl;
      struct {
                  u_int           do_armask_len;
                  oar_mask        *do_armask_val;
          } do_armask;
}
typedef struct directory_obj directory_obj;
```

The main structure contains five primary members: **do_name**, **do_type**, **do_servers**, **do_ttl**, and **do_armask**. The information in the **do_servers** structure is sufficient for the client library to create a network connection with the named server for the directory.

The **do_name** member contains the name of the directory or domain represented in a format that is understandable by the type of nameservice serving that domain. In the case of NIS+ domains, this is the same as the name that can be composed using the **zo_name** and **zo_domain** members. For other name services, this name will be a name that they understand. For example, if this were a directory object describing an X.500 namespace that is "under" the NIS+ directory *eng.hp.com.*, this name might contain "/C=US, /O=Hewlett-Packard, /OU=Engineering/". The type of nameservice that is being described is determined by the value of the member **do_type**.

The **do_servers** structure contains two members. **do_servers_val** is an array of *nis_server* structures; **do_servers_len** is the number of cells in the array. The *nis_server* structure is designed to contain enough information such that machines on the network providing name services can be contacted without having to use a name service. In the case of NIS+ servers, this information is the name of the machine in *name*, its public key for authentication in *pkey*, and a variable length array of endpoints, each of which describes the network endpoint for the **rpcbind** daemon on the named machine. The client library uses the addresses to contact the server using a transport that both the client and server can communicate on and then queries the **rpcbind** daemon to get the actual transport address that the server is using.

Note that the first server in the *do_servers* list is always the master server for the directory.

The *key_type* field describes the type of key stored in the *pkey* netobj (see **/usr/include/rpc/xdr.h** for a definition of the network object structure). Currently supported types are **NIS_PK_NONE** for no public key and **NIS_PK_DH** for a Diffie-Hellman type public key.

The **do_ttl** member contains a copy of the **zo_ttl** member from the common attributes. This is duplicated because the cache manager only caches the variant part of the directory object.

The **do_armask** structure contains two members. **do_armask_val** is an array of **oar_mask** structures; **do_armask_len** is the number of cells in the array. The **oar_mask** structure contains two members: **oa_rights** specifies the access rights allowed for objects of type **oa_otype**. These access rights are used for objects of the given type in the directory when they are present in this array.

The granting of access rights for objects contained within a directory is actually two-tiered. If the directory object itself grants a given access right (using the **zo_access** member in the **nis_object** structure representing the directory), then all objects within the directory are allowed that access. Otherwise, the **do_armask** structure is examined to see if the access is allowed specifically for that type of structure. This allows the administrator of a namespace to set separate policies for different object types, for example, one policy for the creation of tables and another policy for the creation of other directories. See *nis+*(1) for more details.

Link Objects

Link objects provide a means of providing *aliases* or symbolic links within the namespace. Their variant part is defined as follows.

```
struct link_obj {
        zotypes li_rtype;
        struct {
                u_int           li_attrs_len;
                nis_attr        *li_attrs_val;
      } li_attrs;
      nis_name li_name;
}
```

The **li_rtype** member contains the object type of the object pointed to by the link. This is only a hint, since the object which the link points to may have changed or been removed. The fully qualified name of the object (table or otherwise) is specified in the member **li_name**.

NIS+ links can point to either other objects within the NIS+ namespace, or to entries within a NIS+ table. If the object pointed to by the link is a table and the member **li_attrs** has a nonzero number of attributes (index name/value pairs) specified, the table is searched when this link is followed. All entries that match the specified search pattern are returned. Note that unless the flag **FOLLOW_LINKS** is specified, the *nis_lookup*(3N) function will always return non-entry objects.

Group Objects

Group objects contain a membership list of NIS+ principals. The group objects' variant part is defined as follows.

```
struct group_obj {
        u_long  gr_flags;
        struct {
            u_int           gr_members_len;
            nis_name        *gr_members_val;
      } gr_members;
}
```

The **gr_flags** member contains flags that are currently unused. The **gr_members** structure contains the list of principals. For a complete description of how group objects are manipulated see *nis_groups*(3N).

Table Objects

The NIS+ table object is analogous to a YP map. The differences stem from the access controls, and the variable schemas that NIS+ allows. The table objects data structure is defined as follows:

```
#define TA_BINARY       1
#define TA_CRYPT        2
#define TA_XDR          4
#define TA_SEARCHABLE   8
#define TA_CASE        16

struct table_col {
        char    *tc_name;
        u_long  tc_flags;
        u_long  tc_rights;
}
typedef struct table_col table_col;

struct table_obj {
        char    *ta_type;
        u_int   ta_maxcol;
        u_char  ta_sep;
        struct {
                u_int           ta_cols_len;
                table_col       *ta_cols_val;
        } ta_cols;
        char    *ta_path;
}
```

The **ta_type** member contains a string that identifies the type of entries in this table. NIS+ does not enforce any policies as to the contents of this string. However, when entries are added to the table, the NIS+ service will check to see that they have the same "type" as the table specified by this member.

The structure **ta_cols** contains two members. **ta_cols_val** is an array of **table_col** structures. The length of the array depends on the number of columns in the table; it is defined when the table is created and is stored in **ta_cols_len**. **ta_maxcol** also contains the number of columns in the table and always has the same value as **ta_cols_len**. Once the table is created, this length field cannot be changed.

The **ta_sep** character is used by client applications that wish to print out an entry from the table. Typically this is either space (" ") or colon (":").

The **ta_path** string defines a concatenation path for tables. This string contains an ordered list of fully qualified table names, separated by colons, that are to be searched if a search on this table fails to match any entries. This path is only used with the flag **FOLLOW_PATH** with a **nis_list()** call. See *nis_tables*(3N) for information on these flags.

In addition to checking the type, the service will check that the number of columns in an entry is the same as those in the table before allowing that entry to be added.

Each column has associated with it a name in **tc_name**, a set of flags in **tc_flags**, and a set of access rights in **tc_rights**. The name should be indicative of the contents of that column.

The **TA_BINARY** flag indicates that data in the column is binary (rather than text). Columns that are searchable cannot contain binary data. The **TA_CRYPT** flag specifies that the information in this column should be encrypted prior to sending it over the network. This flag has no effect in the export version of NIS+. The **TA_XDR** flag is used to tell the client application that the data in this column is encoded using the XDR protocol. The **TA_BINARY** flag must be specified with the XDR flag. Further, by convention, the name of a column that has the **TA_XDR** flag set is the name of the XDR function that will decode the data in that column.

The **TA_SEARCHABLE** flag specifies that values in this column can be searched. Searchable columns must contain textual data and must have a name associated with them. The flag **TA_CASE** specifies that searches involving this column ignore the case of the value in the column. At least one of the columns in the table should be searchable. Also, the combination of all searchable column values should uniquely select an entry within the table.

Entry Objects

Entry objects are stored in tables. The structure used to define the entry data is as follows.

```
#define EN_BINARY       1
#define EN_CRYPT        2
#define EN_XDR          4
#define EN_MODIFIED     8

struct entry_col {
        u_long  ec_flags;
        struct {
                u_int   ec_value_len;
                char    *ec_value_val;
        } ec_value;
}
typedef struct entry_col entry_col;

struct entry_obj {
        char    *en_type;
        struct {
                u_int           en_cols_len;
                entry_col       *en_cols_val;
        } en_cols;
}
```

The **en_type** member contains a string that specifies the type of data this entry represents. The NIS+ server will compare this string to the type string specified in the table object and disallow any updates or modifications if they differ.

The **en_cols** structure contains two members: **en_cols_len** and **en_cols_val**. **en_cols_val** is an array of **entry_col** structures. **en_cols_len** contains a count of the number of cells in the **en_cols_val** array and reflects the number of columns in the table -- it always contains the same value as the **table_obj.ta_cols.ta_cols_len** member from the table which contains the entry.

The **entry_col** structure contains information about the entry's per-column values. **ec_value** contains information about a particular value. It has two members: **ec_value_val**, which is the value itself, and **ec_value_len**, which is the length (in bytes) of the value. **entry_col** also contains the member **ec_flags**, which contains a set of flags for the entry.

The flags in **ec_flags** are primarily used when adding or modifying entries in a table. All columns that have the flag **EN_CRYPT** set will be encrypted prior to sending them over the network. Columns with **EN_BINARY** set are presumed to contain binary data. The server will ensure that the column in the table object specifies binary data prior to allowing the entry to be added. When modifying entries in a table, only those columns that have changed need be sent to the server. Those columns should each have the **EN_MODIFIED** flag set to indicate this to the server.

SEE ALSO

nis+(1), nis_groups(3N), nis_names(3N), nis_server(3N), nis_subr(3N), nis_tables(3N).

NAME

nis_ping, nis_checkpoint - misc NIS+ log administration functions

SYNOPSIS

cc [*flag*...] *file*... **-lnsl** [*library*...]

```
#include <rpcsvc/nis.h>

void nis_ping(const nis_name dirname,
    const u_long utime,
    const nis_object *dirobj);

nis_result *nis_checkpoint(const nis_name dirname);
```

DESCRIPTION

nis_ping() is called by the master server for a directory when a change has occurred within that directory. The parameter *dirname* identifies the directory with the change. If the parameter *dirobj* is **NULL**, this function looks up the directory object for *dirname* and uses the list of replicas it contains. The parameter *utime* contains the timestamp of the last change made to the directory. This timestamp is used by the replicas when retrieving updates made to the directory.

The effect of calling **nis_ping()** is to schedule an update on the replica. A short time after a ping is received, typically about two minutes, the replica compares the last update time for its databases to the timestamp sent by the ping. If the ping timestamp is later, the replica establishes a connection with the master server and requests all changes from the log that occurred after the last update that it had recorded in its local log.

nis_checkpoint() is used to force the service to checkpoint information that has been entered in the log but has not been checkpointed to disk. When called, this function checkpoints the database for each table in the directory, the database containing the directory and the transaction log. Care should be used in calling this function since directories that have seen a lot of changes may take several minutes to checkpoint. During the checkpointing process, the service will be unavailable for updates for all directories that are served by this machine as master.

nis_checkpoint() returns a pointer to a *nis_result* structure (described in *nis_tables*(3N)). This structure should be freed with **nis_freeresult()** (see *nis_names*(3N)). The only items of interest in the returned result are the status value and the statistics.

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. These functions may be thread cancellation points because they invoke functions that are thread cancellation points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

SEE ALSO

nislog(1M), nis_names(3N), nis_tables(3N), nisfiles(4).

NAME

nis_server, nis_mkdir, nis_rmdir, nis_servstate, nis_stats, nis_getservlist, nis_freeservlist, nis_freetags - miscellaneous NIS+ functions

SYNOPSIS

cc [*flag*...] *file*... **-lnsl** [*library*...]

```
#include <rpcsvc/nis.h>

nis_error nis_mkdir(const nis_name dirname,
    const nis_server *machine);

nis_error nis_rmdir(const nis_name dirname,
    const nis_server *machine);

nis_error nis_servstate(const nis_server *machine,
    const nis_tag *tags,
    const int numtags,
    nis_tag **result);

nis_error nis_stats(const nis_server *machine,
    const nis_tag *tags,
    const int numtags,
    nis_tag **result);

void nis_freetags(nis_tag *tags,
    const int numtags);

nis_server **nis_getservlist(const nis_name dirname);

void nis_freeservlist(nis_server **machines);
```

DESCRIPTION

These functions provide a variety of services for NIS+ applications.

nis_mkdir() is used to create the necessary databases to support NIS+ service for a directory, *dirname*, on a server, *machine*. If this operation is successful, it means that the directory object describing *dirname* has been updated to reflect that server *machine* is serving the named directory. For a description of the **nis_server** structure, refer to *nis_objects*(3N).

nis_rmdir() is used to delete the directory, *dirname*, from the specified machine. The *machine* parameter cannot be **NULL**. For a description of the **nis_server** structure, refer to *nis_objects*(3N).

nis_servstate() is used to set and read the various state variables of the NIS+ servers. In particular the internal debugging state of the servers may be set and queried.

The **nis_stats()** function is used to retrieve statistics about how the server is operating. Tracking these statistics can help administrators determine when they need to add additional replicas or to break up a domain into two or more subdomains. For more information on reading statistics, see *nisstat*(1M).

nis_servstate() and **nis_stats()** use the tag list. This tag list is a variable length array of *nis_tag* structures whose length is passed to the function in the *numtags* parameter. The set of legal tags are defined in the file **<rpcsvc/nis_tags.h>** which is included in **<rpcsvc/nis.h>**. Because these tags can and do vary between implementations of the NIS+ service, it is best to consult this file for the supported list. Passing unrecognized tags to a server will result in their *tag_value* member being set to the string "unknown." Both of these functions return their results in malloced tag structure, **result*. If there is an error, **result* is set to **NULL**. The *tag_value* pointers points to allocated string memory which contains the results. Use **nis_freetags()** to free the tag structure.

nis_getservlist() returns a null terminated list of *nis_server* structures that represent the list of servers that serve the domain named *dirname*. Servers from this list can be used when calling functions that require the name of a NIS+ server. For a description of the **nis_server** structure, refer to *nis_objects*(3N). **nis_freeservlist()** frees the list of servers returned by **nis_getservlist()**. Note that this is the only legal way to free that list.

MULTITHREAD USAGE

Thread Safe: **Yes**
Cancel Safe: **Yes**

Fork Safe: **No**
Async-cancel Safe: **No**
Async-signal Safe: **No**

These functions can be called safely in a multithreaded environment. These functions may be thread cancellation points because they invoke functions that are thread cancellation points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

SEE ALSO
nisstat(1M), nis_names(3N), nis_objects(3N), nis_subr(3N).

n

NAME

nis_subr, nis_leaf_of, nis_name_of, nis_domain_of, nis_getnames, nis_freenames, nis_dir_cmp, nis_clone_object, nis_destroy_object, nis_print_object - NIS+ subroutines

SYNOPSIS

cc [*flag*...] *file*... **-lnsl** [*library*...]

```
#include <rpcsvc/nis.h>
nis_name nis_leaf_of(const nis_name name);
nis_name nis_name_of(const nis_name name);
nis_name nis_domain_of(const nis_name name);
nis_name *nis_getnames(const nis_name name);
void nis_freenames(nis_name *namelist);
name_pos nis_dir_cmp(const nis_name n1, const nis_name n2);
nis_object *nis_clone_object(const nis_object *src, nis_object *dest);
void nis_destroy_object(nis_object *obj);
void nis_print_object(const nis_object *obj);
```

DESCRIPTION

These subroutines are provided to assist in the development of NIS+ applications. They provide several useful operations on both NIS+ names and objects.

The first group, **nis_leaf_of()**, **nis_domain_of()**, and **nis_name_of()** provide the functions for parsing NIS+ names. **nis_leaf_of()** will return the first label in an NIS+ name. It takes into account the double quote character '"' which can be used to protect embedded '.' (dot) characters in object names. Note that the name returned will never have a trailing dot character. If passed the global root directory name ".", it will return the null string.

nis_domain_of() returns the name of the NIS+ domain in which an object resides. This name will always be a fully qualified NIS+ name and ends with a dot. By iteratively calling **nis_leaf_of()** and **nis_domain_of()** it is possible to break a NIS+ name into its individual components.

nis_name_of() is used to extract the unique part of a NIS+ name. This function removes from the tail portion of the name all labels that are in common with the local domain. Thus if a machine were in domain **foo.bar.baz.** and **nis_name_of()** were passed a name **bob.friends.foo.bar.baz**, then **nis_name_of()** would return the unique part, **bob.friends**. If the name passed to this function is not in either the local domain or one of its children, this function will return null.

nis_getnames() will return a list of candidate names for the name passed in as *name*. If this name is not fully qualified, **nis_getnames()** will generate a list of names using the default NIS+ directory search path, or the environment variable **NIS_PATH** if it is set. The returned array of pointers is terminated by a NULL pointer, and the memory associated with this array should be freed by calling **nis_freenames()**.

Though **nis_dir_cmp()** can be used to compare any two NIS+ names, it is used primarily to compare domain names. This comparison is done in a case independent fashion, and the results are an enum of type **name_pos**. When the names passed to this function are identical, the function returns a value of SAME_NAME. If the name *n1* is a direct ancestor of name *n2*, then this function returns the result HIGHER_NAME. Similarly, if the name *n1* is a direct descendant of name *n2*, then this function returns the result LOWER_NAME. When the name *n1* is neither a direct ancestor nor a direct descendant of *n2*, as it would be if the two names were siblings in separate portions of the namespace, then this function returns the result NOT_SEQUENTIAL. Finally, if either name cannot be parsed as a legitimate name then this function returns the value BAD_NAME.

The second set of functions, consisting of **nis_clone_object()** and **nis_destroy_object()**, are used for manipulating objects. **nis_clone_object()** creates an exact duplicate of the NIS+ object *src*. If the value of *dest* is non-null, it creates the clone of the object into this object structure and allocates the necessary memory for the variable length arrays. If this parameter is null, a pointer to the cloned object is returned. Refer to *nis_objects*(3N) for a description of the **nis_object** structure.

nis_destroy_object() can be used to destroy an object created by **nis_clone_object()**. This will free up all memory associated with the object and free the pointer passed. If the object was cloned into

an array (using the *dest* parameter to **nis_clone_object()**) then the object *cannot* be freed with this function. Instead, the function **xdr_free(xdr_nis_object,** *dest***)** must be used.

nis_print_object() prints out the contents of a NIS+ object structure on the standard output. Its primary use is for debugging NIS+ programs.

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. These functions may be thread cancellation points because they invoke functions that are thread cancellation points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

ENVIRONMENT

NIS_PATH This variable overrides the default NIS+ directory search path used by **nis_getnames()**. It contains an ordered list of directories separated by ':' (colon) characters. The '$' (dollar sign) character is treated specially. Directory names that end in '$' have the default domain appended to them, and a '$' by itself is replaced by the list of directories between the default domain and the global root that are at least two levels deep. The default NIS+ directory search path is '$'.

NOTES

nis_leaf_of(), **nis_name_of()** and **nis_clone_object()** return their results as thread-specific data in multithreaded applications.

SEE ALSO

nis_names(3N), nis_objects(3N), nis_tables(3N).

NAME

nis_tables, nis_list, nis_add_entry, nis_remove_entry, nis_modify_entry, nis_first_entry, nis_next_entry - NIS+ table functions

SYNOPSIS

cc [*flag*...] *file*... **-lnsl** [*library*...]

```
#include <rpcsvc/nis.h>

nis_result *nis_list(const nis_name name,
    const u_long flags,
    int (*callback)(const nis_name table_name,
    const nis_object *object,
    const void *userdata),
    const void *userdata));

nis_result *nis_add_entry(const nis_name table_name,
    const nis_object *object,
    const u_long flags);

nis_result *nis_remove_entry(const nis_name name,
    const nis_object *object,
    const u_long flags);

nis_result *nis_modify_entry(const nis_name name,
    const nis_object *object,
    const u_long flags);

nis_result *nis_first_entry(const nis_name table_name);

nis_result *nis_next_entry(const nis_name table_name,
    const netobj *cookie);

void nis_freeresult(nis_result *result);
```

DESCRIPTION

These functions are used to search and modify NIS+ tables. **nis_list()** is used to search a table in the NIS+ namespace. **nis_first_entry()** and **nis_next_entry()** are used to enumerate a table one entry at a time. **nis_add_entry()**, **nis_remove_entry()**, and **nis_modify_entry()** are used to change the information stored in a table. **nis_freeresult()** is used to free the memory associated with the **nis_result** structure.

Entries within a table are named by NIS+ indexed names. An indexed name is a compound name that is composed of a search criteria and a simple NIS+ name that identifies a table object. A search criteria is a series of column names and their associated values enclosed in bracket '[]' characters. Indexed names have the following form:

[*colname*=*value*, **...** **],** *tablename*

The list function, **nis_list()**, takes an indexed name as the value for the *name* parameter. Here, the tablename should be a fully qualified NIS+ name unless the **EXPAND_NAME** flag (described below) is set. The second parameter, *flags*, defines how the function will respond to various conditions. The value for this parameter is created by logically **OR**ing together one or more flags from the following list.

FOLLOW_LINKS
: If the table specified in *name* resolves to be a **LINK** type object (see *nis_objects*(3N)), this flag specifies that the client library follow that link and do the search at that object. If this flag is not set and the name resolves to a link, the error **NIS_NOTSEARCHABLE** will be returned.

FOLLOW_PATH
: This flag specifies that if the entry is not found within this table, the list operation should follow the path specified in the table object. When used in conjunction with the **ALL_RESULTS** flag below, it specifies that the path should be followed regardless of the result of the search. When used in conjunction with the **FOLLOW_LINKS** flag above, named tables in the path that resolve to links will be followed until the table they point to is located. If a table in the path is not reachable because no server that serves it is available, the result of the operation will be either a soft success or a soft failure to indicate that not all tables in the path could be searched. If a name in the path names is either an

invalid or non-existent object then it is silently ignored.

HARD_LOOKUP
This flag specifies that the operation should continue trying to contact a server of the named table until a definitive result is returned (such as NIS_NOTFOUND).

ALL_RESULTS
This flag can only be used in conjunction with **FOLLOW_PATH** and a callback function. When specified, it forces all of the tables in the path to be searched. If *name* does not specify a search criteria (implying that all entries are to be returned), then this flag will cause all of the entries in all of the tables in the path to be returned.

NO_CACHE This flag specifies that the client library should bypass any client object caches and get its information directly from either the master server or a replica server for the named table.

MASTER_ONLY
This flag is even stronger than **NO_CACHE** in that it specifies that the client library should *only* get its information from the master server for a particular table. This guarantees that the information will be up to date. However, there may be severe performance penalties associated with contacting the master server directly on large networks. When used in conjunction with the **HARD_LOOKUP** flag, this will block the list operation until the master server is up and available.

EXPAND_NAME
When specified, the client library will attempt to expand a partially qualified name by calling **nis_getnames()** (see *nis_local_names*(3N)) which uses the environment variable **NIS_PATH**.

RETURN_RESULT
This flag is used to specify that a copy of the returning object be returned in the **nis_result** structure if the operation was successful.

The third parameter to **nis_list()**, *callback*, is an optional pointer to a function that will process the **ENTRY** type objects that are returned from the search. If this pointer is **NULL**, then all entries that match the search criteria are returned in the *nis_result* structure, otherwise this function will be called once for each entry returned. When called, this function should return **0** when additional objects are desired and **1** when it no longer wishes to see any more objects. The fourth parameter, *userdata*, is simply passed to callback function along with the returned entry object. The client can use this pointer to pass state information or other relevant data that the callback function might need to process the entries.

nis_add_entry() will add the NIS+ object to the NIS+ *table_name*. The *flags* parameter is used to specify the failure semantics for the add operation. The default (*flags* equal 0) is to fail if the entry being added already exists in the table. The **ADD_OVERWRITE** flag may be used to specify that the existing object is to be overwritten if it exists, (a modify operation) or added if it does not exist. With the **ADD_OVERWRITE** flag, this function will fail with the error **NIS_PERMISSION** if the existing object does not allow modify privileges to the client.

If the flag **RETURN_RESULT** has been specified, the server will return a copy of the resulting object if the operation was successful.

nis_remove_entry() removes the identified entry from the table or a set of entries identified by *table_name*. If the parameter *object* is non-null, it is presumed to point to a cached copy of the entry. When the removal is attempted, and the object that would be removed is not the same as the cached object pointed to by *object* then the operation will fail with an **NIS_NOTSAMEOBJ** error. If an object is passed with this function, the search criteria in name is optional as it can be constructed from the values within the entry. However, if no object is present, the search criteria must be included in the *name* parameter. If the flags variable is null, and the search criteria does not uniquely identify an entry, the **NIS_NOTUNIQUE** error is returned and the operation is aborted. If the flag parameter **REM_MULTIPLE** is passed, and if remove permission is allowed for each of these objects, then all objects that match the search criteria will be removed. Note that a null search criteria and the **REM_MULTIPLE** flag will remove all entries in a table.

nis_modify_entry() modifies an object identified by *name*. The parameter *object* should point to an entry with the **EN_MODIFIED** flag set in each column that contains new information. These columns will replace their counterparts in the entry that is stored in the table. The entry passed must have the same number of columns, same type, and valid data in the modified columns for this operation to succeed.

If the flags parameter contains the flag **MOD_SAMEOBJ** then the object pointed to by *object* is assumed to be a cached copy of the original object. If the OID of the object passed is different than the OID of the object the server fetches, then the operation fails with the **NIS_NOTSAMEOBJ** error. This can be used to implement a simple read-modify-write protocol which will fail if the object is modified before the client can write the object back.

If the flag **RETURN_RESULT** has been specified, the server will return a copy of the resulting object if the operation was successful.

nis_first_entry() fetches entries from a table one at a time. This mode of operation is extremely inefficient and callbacks should be used instead wherever possible. The table containing the entries of interest is identified by *name*. If a search criteria is present in *name* it is ignored. The value of *cookie* within the **nis_result** structure must be copied by the caller into local storage and passed as an argument to **nis_next_entry()**.

nis_next_entry() retrieves the next entry from a table specified by *table_name*. The order in which entries are returned is not guaranteed. Further, should an update occur in the table between client calls to **nis_next_entry()** there is no guarantee that an entry that is added or modified will be seen by the client. Should an entry be removed from the table that would have been the next entry returned, the error **NIS_CHAINBROKEN** is returned instead.

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. These functions may be thread cancellation points because they invoke functions that are thread cancellation points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

n

RETURN VALUES

These functions return a pointer to a structure of type **nis_result**:

```
struct nis_result {
          nis_error         status;
          struct {
                  u_int           objects_len;
                  nis_object      *objects_val;
        } objects;
        netobj      cookie;
        u_long      zticks;
        u_long      dticks;
        u_long      aticks;
        u_long      cticks;
};
```

The *status* member contains the error status of the the operation. A text message that describes the error can be obtained by calling the function **nis_sperrno()** (see *nis_error*(3N)).

The **objects** structure contains two members. *objects_val* is an array of *nis_object* structures; *objects_len* is the number of cells in the array. These objects will be freed by a call to **nis_freeresult()** (see *nis_names*(3N)). If you need to keep a copy of one or more objects, they can be copied with the function **nis_clone_object()** and freed with the function **nis_destroy_object()** (see *nis_server*(3N)).

The various ticks contain details of where the time (in microseconds) was taken during a request. They can be used to tune one's data organization for faster access and to compare different database implementations (see *nis_db*(3N)).

zticks

The time spent in the NIS+ service itself, this count starts when the server receives the request and stops when it sends the reply.

dticks
The time spent in the database backend, this time is measured from the time a database call starts, until a result is returned. If the request results in multiple calls to the database, this is the sum of all the time spent in those calls.

aticks
The time spent in any "accelerators" or caches. This includes the time required to locate the server needed to resolve the request.

cticks
The total time spent in the request, this clock starts when you enter the client library and stops when a result is returned. By subtracting the sum of the other ticks values from this value you can obtain the local overhead of generating a NIS+ request.

Subtracting the value in *dticks* from the value in *zticks* will yield the time spent in the service code itself. Subtracting the sum of the values in *zticks* and *aticks* from the value in *cticks* will yield the time spent in the client library itself. Note: all of the tick times are measured in microseconds.

ERRORS

The client library can return a variety of error returns and diagnostics. The more salient ones are documented below.

NIS_BADATTRIBUTE
The name of an attribute did not match up with a named column in the table, or the attribute did not have an associated value.

NIS_BADNAME The name passed to the function is not a legal NIS+ name.

NIS_BADREQUEST
A problem was detected in the request structure passed to the client library.

NIS_CACHEEXPIRED
The entry returned came from an object cache that has *expired*. This means that the time to live value has gone to zero and the entry may have changed. If the flag NO_CACHE was passed to the lookup function then the lookup function will retry the operation to get an unexpired copy of the object.

NIS_CBERROR An RPC error occurred on the server while it was calling back to the client. The transaction was aborted at that time and any unsent data was discarded.

NIS_CBRESULTS Even though the request was successful, all of the entries have been sent to your callback function and are thus not included in this result.

NIS_FOREIGNNS The name could not be completely resolved. When the name passed to the function would resolve in a namespace that is outside the NIS+ name tree, this error is returned with a NIS+ object of type **DIRECTORY**. The returned object contains the type of namespace and contact information for a server within that namespace.

NIS_INVALIDOBJ
The object pointed to by *object* is not a valid NIS+ entry object for the given table. This could occur if it had a mismatched number of columns, or a different data type (for example, binary or text) than the associated column in the table.

NIS_LINKNAMEERROR
The name passed resolved to a **LINK** type object and the contents of the object pointed to an invalid name.

NIS_MODFAIL The attempted modification failed for some reason.

NIS_NAMEEXISTS
An attempt was made to add a name that already exists. To add the name, first remove the existing name and then add the new name or modify the existing named object.

NIS_NAMEUNREACHABLE
This soft error indicates that a server for the desired directory of the named table object could not be reached. This can occur when there is a network partition or the server has crashed. Attempting the operation again may succeed. See the **HARD_LOOKUP** flag.

NIS_NOCALLBACK
The server was unable to contact the callback service on your machine. This results in no data being returned.

NIS_NOMEMORY Generally a fatal result. It means that the service ran out of heap space.

NIS_NOSUCHNAME
This hard error indicates that the named directory of the table object does not exist. This occurs when the server that should be the parent of the server that serves the table, does not know about the directory in which the table resides.

NIS_NOSUCHTABLE
The named table does not exist.

NIS_NOT_ME A request was made to a server that does not serve the given name. Normally this will not occur, however if you are not using the built in location mechanism for servers, you may see this if your mechanism is broken.

NIS_NOTFOUND No entries in the table matched the search criteria. If the search criteria was null (return all entries) then this result means that the table is empty and may safely be removed by calling the **nis_remove()**.

If the FOLLOW_PATH flag was set, this error indicates that none of the tables in the path contain entries that match the search criteria.

NIS_NOTMASTER A change request was made to a server that serves the name, but it is not the master server. This can occur when a directory object changes and it specifies a new master server. Clients that have cached copies of the directory object in the **/var/nis/NIS_SHARED_DIRCACHE** file will need to have their cache managers restarted (use **nis_cachemgr -i**) to flush this cache.

NIS_NOTSAMEOBJ
An attempt to remove an object from the namespace was aborted because the object that would have been removed was not the same object that was passed in the request.

NIS_NOTSEARCHABLE
The table name resolved to a NIS+ object that was not searchable.

NIS_PARTIAL This result is similar to **NIS_NOTFOUND** except that it means the request succeeded but resolved to zero entries. When this occurs, the server returns a copy of the table object instead of an entry so that the client may then process the path or implement some other local policy.

NIS_RPCERROR This fatal error indicates the RPC subsystem failed in some way. Generally there will be a *syslog*(3C) message indicating why the RPC request failed.

NIS_S_NOTFOUND
The named entry does not exist in the table, however not all tables in the path could be searched, so the entry may exist in one of those tables.

NIS_S_SUCCESS Even though the request was successful, a table in the search path was not able to be searched, so the result may not be the same as the one you would have received if that table had been accessible.

NIS_SUCCESS The request was successful.

NIS_SYSTEMERROR
Some form of generic system error occurred while attempting the request. Check the *syslog*(3C) record for error messages from the server.

NIS_TOOMANYATTRS
The search criteria passed to the server had more attributes than the table had searchable columns.

NIS_TRYAGAIN The server connected to was too busy to handle your request. **add_entry()**, **remove_entry()**, and **modify_entry()** return this error when the master server is currently updating its internal state. It can be returned to **nis_list()** when the function specifies a callback and the server does not have the resources to handle callbacks.

NIS_TYPEMISMATCH
An attempt was made to add or modify an entry in a table, and the entry passed was of a different type than the table.

ENVIRONMENT

NIS_PATH When set, this variable is the search path used by **nis_list()** if the flag **EXPAND_NAME** is set.

WARNINGS

Use the flag **HARD_LOOKUP** carefully since it can cause the application to block indefinitely during a network partition.

NOTES

The path used when the flag **FOLLOW_PATH** is specified, is the one present in the *first* table searched. The path values in tables that are subsequently searched are ignored.

It is legal to call functions that would access the nameservice from within a list callback. However, calling a function that would itself use a callback, or calling **nis_list()** with a callback from within a list callback function is not currently supported.

There are currently no known methods for **nis_first_entry()** and **nis_next_entry()** to get their answers from only the master server.

SEE ALSO

niscat(1), niserror(1), nismatch(1), nis_cachemgr(1M), nis_error(3N), nis_local_names(3N), nis_names(3N), nis_objects(3N), syslog(3C),

n

NAME
nl, nonl — enable/disable newline translation

SYNOPSIS
```
#include <curses.h>
int nl(void);
int nonl(void);
```

DESCRIPTION
The **nl()** function enables a mode in which carriage return is translated to newline on input. The **nonl()** function disables the above translation. Initially, the above translation is enabled.

RETURN VALUE
Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS
No errors are defined.

APPLICATION USAGE
The default translation adapts the terminal to environments in which newline is the line termination character. However, by disabling the translation with **nonl()**, the application can sense the pressing of the carriage return key.

SEE ALSO
<curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The entry is rewritten for clarity. The argument list for the **nl()** and **nonl()** functions is explicitly declared as **void**.

n

NAME

nl_langinfo( ) - language information

SYNOPSIS

```
#include <langinfo.h>
char *nl_langinfo(nl_item item);
```

DESCRIPTION

nl_langinfo() returns a pointer to a null-terminated string containing information relevant to a particular language or cultural area defined in the program's locale (see *setlocale*(3C)). The manifest constant names and values of **item** are defined in <**langinfo.h**>. For example:

```
nl_langinfo(ABDAY_1)
```

returns a pointer to the string "Dom" if the language identified by the current locale is pt_PT.iso88591, and "Su" if the identified language is fi_FI.iso88591.

If an invalid **item** is specified, a pointer to an empty string is returned. An empty string can also be returned for a valid **item** if that item is not applicable to the language or customs of the current locale. For example, a thousands separator is not used when writing numbers according to the customs associated with the Arabic language.

EXTERNAL INFLUENCES

Locale

The string returned for a particular **item** is determined by the locale category specified for that item in *langinfo*(5).

International Code Set Support

Single- and multi-byte character code sets are supported.

WARNINGS

nl_langinfo() returns a pointer to a static area that is overwritten on each call.

n

AUTHOR

nl_langinfo() was developed by OSF and HP.

SEE ALSO

localeconv(3C), setlocale(3C), hpnls(5), lang(5), langinfo(5).

STANDARDS CONFORMANCE

nl_langinfo(): AES, SVID3, XPG2, XPG3, XPG4

NAME

nl_tools_16: ADVANCE(), byte_status(), BYTE_STATUS(), CHARADV(), CHARAT(), c_colwidth(), C_COLWIDTH(), firstof2(), FIRSTof2(), secof2(), SECof2(), WCHAR(), WCHARADV() - tools to process 16-bit characters (OBSOLETED AT 10.30)

SYNOPSIS

```
#include <nl_ctype.h>
int firstof2(int c);
int secof2(int c);
int byte_status(int c, int laststatus);
int c_colwidth(int c);
int FIRSTof2(int c);
int SECof2(int c);
int BYTE_STATUS(int c, int laststatus);
int C_COLWIDTH(int c);
int CHARAT(const char *p);
int ADVANCE(const char *p);
int CHARADV(const char *p);
int WCHAR(wchar_t wc, char *p);
int WCHARADV(wchar_t wc, char *p);
void PCHAR(int c, char *p);
void PCHARADV(int c, char *p);
```

Remarks

All interfaces listed above whose names begin with a capital letter are implemented as macros; the others are functions.

These macros and functions are obsolete. See WARNINGS.

n

DESCRIPTION

The following macros and functions perform their operations based upon the loaded NLS environment (see *setlocale*(3C)).

FIRSTof2() Takes a byte and returns a nonzero value if it can be the first byte of a two-byte character according to the NLS environment loaded, and zero if it cannot.

SECof2() Takes a byte and returns a nonzero value if it can be the second byte of a two-byte character according to the loaded NLS environment, and zero if it cannot.

BYTE_STATUS() Returns one of the following values based on the value of the current byte in *c* and the status of the previous byte interpreted in *laststatus* as returned by the last call to **BYTE_STATUS()**. These are the status values as defined in **<nl_ctype.h>**:

ONEBYTE	Single-byte character
SECOF2	Second byte of two-byte character
FIRSTOF2	First byte of two-byte character

To validate a two-byte character, both the first and second bytes must be valid. If the value of *laststatus* is **FIRSTOF2** but **SECof2(**c**)** returns false, **BYTE_STATUS(** *c*, *laststatus* **)** returns **ONEBYTE**.

C_COLWIDTH() Takes a byte which is assumed to be either a one-byte character or the first byte of a two-byte character, and returns the number of columns the character would occupy on a terminal display.

For the macros **BYTE_STATUS()**, **C_COLWIDTH()**, **FIRSTof2()**, and **SEC0of2()**, results are undefined for values of *c* less than –1 (EOF) or greater than 255.

CHARAT() Takes as an argument a pointer *p*, which is assumed to be pointing at either a one-byte character or the first byte of a two-byte character. In either case, **CHARAT()** returns the **wchar_t** value that corresponds to the character pointed to by *p*.

ADVANCE() Advances its pointer argument by the byte width of the character it is pointing at (either one or two bytes).

CHARADV() Combines the functions of **CHARAT()** and **ADVANCE()** in a single macro. It takes as an argument a pointer *p*, which is assumed to be pointing at either a one-byte character or the first byte of a two-byte character. In either case **CHARADV()** returns the **wchar_t** value that corresponds to the character pointed to by *p*, and advances *p* beyond the last byte of the character.

WCHAR() Converts the **wchar_t** value *wc* into the corresponding one or two byte character, and writes it at the location specified by *p*. **WCHAR()** returns the **wchar_t** value *wc*.

WCHARADV() Combines the functions of **WCHAR()** and **ADVANCE()** in a single macro. It converts the **wchar_t** value *wc* into the corresponding one or two byte character, and writes it at the location specified by *p*, then advances *p* past the last byte. **WCHARADV()** returns the **wchar_t** value *wc*.

firstof2()
secof2()
byte_status()
c_colwidth() Subroutine versions of the corresponding macros. These functions can be called from languages other than C.

EXTERNAL INFLUENCES

Locale

The **LC_CTYPE** category determines the interpretation of single and/or multibyte characters.

WARNINGS

The HP proprietary functions and macros described in this manual entry are *OBSOLETE*. They are not portable to other vendor's systems, and will not be provided in future HP-UX releases.

For maximum portability, use the routines documented in the *multibyte*(3C) manual entry for multibyte character processing.

Other macros listed in this manual entry cannot be used as the first argument to **WCHAR()** or **WCHARADV()**. For example,

```
*t++ = *f++
```

cannot be replaced by

```
WCHARADV(CHARADV (f),t)
```

Instead, use a method such as

```
int c; ... c = CHARADV (f), WCHARADV (c,t)
```

WCHAR() and **WCHARADV()** may produce a "null effect" warning from *lint*(1) if not used as part of another expression or as part of a statement. This does not affect the functionality of either macro.

Note that **WCHAR()** and **WCHARADV()** are not "replace_char" macros. They do not prevent the second byte of a two-byte character from being left dangling if **WCHAR()** or **WCHARADV()** overwrite the first byte of the two-byte character with a single-byte character.

CHARAT(), **ADVANCE()**, and **CHARADV()** do not examine the byte following the location pointed to by the argument to verify its validity as a **SECof2** byte.

AUTHOR

nl_tools_16() was developed by HP.

SEE ALSO

setlocale(3C), multibyte(3C), wconv(3X), wctype(3X), hpnls(5).

NAME

nlist(), nlist64() - get entries from name list

SYNOPSIS

```
#include <nlist.h>
int nlist(const char *file_name, struct nlist *nl);
int nlist64(const char* file_name, struct nlist64 *nl);
```

Remarks

The use of symbol table type and value information is inherently non-portable. Use of **nlist()** or **nlist64()** should reduce the effort required to port a program that uses such information, but complete portability across all HP-UX implementations cannot be expected.

DESCRIPTION

nlist() and **nlist64()** have basically the same functionality except **nlist64()** uses a new nlist structure, **nlist64**, and can process **Som** or **Elf** files. **nlist()** can only process Som files and uses the **nlist** structure.

The **nlist** function 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/nlist64** structures pointed to by *nl*. The array of **nlist/nlist64** structures initially contains only the names of variables. Once the **nlist** function has been called, the variable names are augmented with symbol information. The list is terminated by a null name, which consists of a null string in the variable-name position of the structure. The name list of the file is searched for each variable name. If the name is found, the symbol's type, scope, and value in the file is inserted into the name list structure. For **nlist64()**, if the file searched is an Elf file, the section index is also inserted. For **nlist()** and Som files, the subspace index is inserted. On wide mode systems, the symbol value is 64 bits. If the file searched is a Som file, then the value field is zero padded. If the name is not found, the fields in the name list structure are set to 0. The structures **nlist** and **nlist64** are defined in the include file <**nlist.h**>. See *a.out*(4) and *nlist*(4) for further description of the symbol table structure.

The file must have the organization and symbol table described for an **a.out** file in *a.out*(4). The information is extracted from the symbol table used by the loader, *ld*(1).

On machines that have such a file, this subroutine is useful for examining the system name list kept in file **/stand/vmunix**. In this way programs can obtain system addresses that are up to date.

RETURN VALUE

All nlist structure fields are set to 0 if the file cannot be found or if it is not a valid object file containing a linker symbol table.

nlist() returns -1 upon error; otherwise it returns 0.

WARNINGS

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

SEE ALSO

a.out(4), nlist(4).

STANDARDS CONFORMANCE

nlist(): SVID2, SVID3

NAME
nodelay — enable or disable block during read

SYNOPSIS
```
#include <curses.h>
int nodelay(WINDOW *win, bool bf);
```

DESCRIPTION
The **nodelay()** function specifies whether Delay Mode or No Delay Mode is in effect for the screen associated with the specified window. If *bf* is TRUE, this screen is set to No Delay Mode. If *bf* is FALSE, this screen is set to Delay Mode. The initial state is FALSE.

RETURN VALUE
Upon successful completion, **nodelay()** returns OK. Otherwise, it returns ERR.

ERRORS
No errors are defined.

SEE ALSO
Input Processing in curses_intro(3X), getch(3X), halfdelay(3X), <curses.h>, *X/Open System Interface Definitions, Issue 4, Version 2* specification, Section 9.2, *Parameters That Can Be Set*.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The entry is rewritten for clarity.

n

(ENHANCED CURSES)

NAME

noqiflush, qiflush — enable/disable queue flushing

SYNOPSIS

```
#include <curses.h>
void noqiflush(void);
void qiflush(void);
```

DESCRIPTION

The **qiflush()** function causes all output in the display driver queue to be flushed whenever an interrupt key (interrupt, suspend, or quit) is pressed. The **noqiflush()** causes no such flushing to occur. The default for the option is inherited from the display driver settings.

RETURN VALUE

These functions do not return a value.

ERRORS

No errors are defined.

APPLICATION USAGE

Calling **qiflush()** provides faster response to interrupts, but causes Curses to have the wrong idea of what is on the screen. The same effect is achieved outside Curses using the NOFLSH local mode flag specified in the *X/Open System Interface Definitions, Issue 4, Version 2* specification (*General Terminal Interface*).

SEE ALSO

Input Processing in curses_intro(3X), intrflush(3X), <curses.h>, *X/Open System Interface Definitions, Issue 4, Version 2* specification, Section 9.2, *Parameters That Can Be Set* (NOFLSH flag).

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

n

NAME

notimeout, timeout, wtimeout — control blocking on input

SYNOPSIS

```
#include <curses.h>
int notimeout(WINDOW *win, bool bf);
void timeout(int delay);
void wtimeout(WINDOW *win, int delay);
```

DESCRIPTION

The **notimeout()** function specifies whether Timeout Mode or No Timeout Mode is in effect for the screen associated with the specified window. If *bf* is TRUE, this screen is set to No Timeout Mode. If *bf* is FALSE, this screen is set to Timeout Mode. The initial state is FALSE.

The **timeout()** and **wtimeout()** functions set blocking or non-blocking read for the current or specified window based on the value of *delay*:

delay **< 0** One or more blocking reads (indefinite waits for input) are used.

delay **= 0** One or more non-blocking reads are used. Any Curses input function will fail if every character of the requested string is not immediately available.

delay **> 0** Any Curses input function blocks for *delay* milliseconds and fails if there is still no input.

RETURN VALUE

Upon successful completion, the **notimeout()** function returns OK. Otherwise, it returns ERR.

The **timeout()** and **wtimeout()** functions do not return a value.

ERRORS

No errors are defined.

n

SEE ALSO

Input Processing in curses_intro(3X), getch(3X), halfdelay(3X), nodelay(3X), <curses.h>, *X/Open System Interface Definitions, Issue 4, Version 2* specification, Section 9.2, *Parameters That Can Be Set*.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME
overlay, overwrite — copy overlapped windows

SYNOPSIS
```
#include <curses.h>
int overlay(const WINDOW *srcwin, WINDOW *dstwin);
int overwrite(const WINDOW *srcwin, WINDOW *dstwin);
```

DESCRIPTION
The **overlay()** and **overwrite()** functions overlay *srcwin* on top of *dstwin*. The *scrwin* and *dstwin* arguments need not be the same size; only text where the two windows overlap is copied.

The **overwrite()** function copies characters as though a sequence of **win_wch()** and **wadd_wch()** were performed with the destination window's attributes and background attributes cleared.

The **overlay()** function does the same thing, except that, whenever a character to be copied is the background character of the source window, **overlay()** does not copy the character but merely moves the destination cursor the width of the source background character.

If any portion of the overlaying window border is not the first column of a multi-column character then all the column positions will be replaced with the background character and rendition before the overlay is done. If the default background character is a multi-column character when this occurs, then these functions fail.

RETURN VALUE
Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS
No errors are defined.

SEE ALSO
copywin(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The entry is rewritten for clarity. The type of argument **srcwin()** is changed from **WINDOW *** to **WINDOW *CONST**.

NAME

PAM - Pluggable Authentication Module

SYNOPSIS

#include <security/pam_appl.h>

cc [*flag* ...] *file* ... **-lpam** [*library* ...]

DESCRIPTION

PAM gives system administrators the flexibility of choosing any authentication service available on the system to perform authentication. The framework also allows new authentication service modules to be plugged in and made available without modifying the applications.

The PAM framework, **libpam**, consists of an interface library and multiple authentication service modules. The PAM interface library is the layer implementing the Application Programming Interface (API). The authentication service modules are a set of dynamically loadable objects invoked by the PAM API to provide a particular type of user authentication.

Interface Overview

The PAM library interface consists of functions which can be grouped into five categories. The names for all the authentication library functions start with **pam_**.

The first category contains functions for establishing and terminating an authentication activity (*pam_start*(3) and *pam_end*(3)), functions to maintain module specific data (*pam_[sg]et_data*(3)), functions to maintain state information (*pam_[sg]et_item*(3)), and a function to return error status information (*pam_strerror*(3)).

The second category contains functions to authenticate an individual user (*pam_authenticate*(3)) and to set the credentials of the user (*pam_setcred*(3)).

The third category contains functions to do account management (*pam_acct_mgmt*(3)). This includes checking for password aging and access-hour restrictions.

The fourth category contains functions to perform session management (*pam_open_session*(3) and *pam_close_session*(3)) after access to the system has been granted.

The fifth category consists of functions to change authentication tokens *pam_chauthtok*(3). An authentication token is the object used to verify the identity of the user. In UNIX, an authentication token is a user's password, even when using a smart card, because the PAM Framework retrieves the password from the smart card.

All the **pam_*()** interfaces are implemented through the library **libpam**. For each of the categories listed above, excluding the first category **pam_start()**, **pam_end()**, **pam_[sg]et_data()**, **pam_[sg]et_item()**, and **pam_strerror()**) there exists a dynamically loadable shared module that provides the appropriate service layer functionality upon demand. The functional entry points in the service layer start with the **pam_sm_** prefix. The only difference between the **pam_sm_*()** interfaces and their corresponding **pam_** interfaces is that all the **pam_sm_*()** interfaces require extra parameters to pass service specific options to the shared modules. Please refer to *pam_sm*(3) for an overview of the PAM service module APIs.

Stateful Interface

A sequence of calls sharing a common set of state information is referred to as an authentication transaction. An authentication transaction begins with a call to **pam_start()**. **pam_start()** allocates space, performs various initialization activities, and assigns a PAM authentication handle to be used for subsequent calls to the library.

After initiating an authentication transaction, applications can invoke **pam_authenticate()** to authenticate a particular user, and **pam_acct_mgmt()** to perform system entry management (the application may want to determine if the user's password has expired).

If the user has been successfully authenticated, applications call **pam_setcred()** to set any user credentials associated with the authentication service. Within one authentication transaction (between **pam_start()** and **pam_end()**), all calls to the PAM interface should be made with the same authentication handle returned by **pam_start()**. This is necessary because certain service modules may store module-specific data in the handle that is intended for use by other modules. For example, during the call to **pam_authenticate()**, service modules may store data in the handle that is intended for use by **pam_setcred()**.

To perform session management, applications call **pam_open_session()**. For example, the system may want to store the total time for the session. The function **pam_close_session()** closes the current session.

When necessary, applications can call **pam_get_item()** and **pam_set_item()** to access and update specific authentication information. Such information may include the current username.

To terminate an authentication transaction, the application simply calls **pam_end()**, which frees previously allocated space used to store authentication information.

Application - Authentication Service Interactive Interface

The authentication service in PAM does not communicate directly with the user; instead it relies on the application to perform all such interactions. The application passes a pointer to the function, **conv()**, along with any associated application data pointers, through a **pam_conv** structure to the authentication service when it initiates an authentication transaction (via a call to **pam_start()**). The service will then use the function, **conv()**, to prompt the user for data, output error messages, and display text information. Refer to *pam_start*(3) for more information.

Stacking Multiple Schemes

The PAM architecture enables authentication by multiple authentication services through *stacking*. System entry applications, such as *login*(1), stack multiple service modules to authenticate users with multiple authentication services. The order in which authentication service modules are stacked is specified in the configuration file, *pam.conf*(4). A system administrator determines this ordering, and also determines whether the same password can be used for all authentication services.

Administrative Interface

The authentication library, **/usr/lib/libpam.1**, implements the framework interface. Various authentication services are implemented by their own loadable modules whose paths are specified through the **pam.conf**(4) file.

User configuration

The system administrator can determine a policy by user. These are specified in the configuration files: *pam.conf*(4), *pam_user.conf*(4).

RETURN VALUES

The PAM functions may return one of the following generic values, or one of the values defined in the specific man pages:

PAM_SUCCESS	Successful function return.
PAM_OPEN_ERR	**shl_load()** failure when dynamically loading a service module.
PAM_SYMBOL_ERR	Symbol not found.
PAM_SERVICE_ERR	Error in service module.
PAM_SYSTEM_ERR	System error.
PAM_BUF_ERR	Memory buffer error.
PAM_CONV_ERR	Conversation failure.
PAM_PERM_DENIED	Permission denied.

WARNINGS

Please note that all the PAM APIs and the data structures are subject to change without notice.

SEE ALSO

pam_authenticate(3), pam_open_session(3), pam_chauthtok(3), pam_set_item(3), pam_setcred(3), pam_sm(3), pam_start(3), pam_strerror(3), pam.conf(4), pam_user.conf(4).

NAME

pam_acct_mgmt - perform PAM account validation procedures

SYNOPSIS

cc [*flag* ...] *file* ... **-lpam** [*library* ...]

```
#include <security/pam_appl.h>
int pam_acct_mgmt(pam_handle_t *pamh, int flags);
```

DESCRIPTION

The function **pam_acct_mgmt()** is called to determine if the current user's account is valid. This includes checking for password and account expiration, as well as verifying access hour restrictions and terminal access restrictions for trusted mode. This function is typically called after the user has been authenticated with *pam_authenticate*(3).

The *pamh* argument is an authentication handle obtained by a prior call to **pam_start()**. The following flags may be set in the *flags* field:

PAM_SILENT
The account management service should not generate any messages

PAM_DISALLOW_NULL_AUTHTOK
The account management service should return **PAM_AUTHTOKEN_REQD** if the user has a null authentication token.

RETURN VALUES

Upon successful completion, **PAM_SUCCESS** is **returned.** In addition to the error return values described in *pam*(3), the following values may be returned:

PAM_USER_UNKNOWN	User not known to underlying account management module.
PAM_AUTH_ERR	Authentication failure.
PAM_AUTHTOKEN_REQD	New authentication token required. This is normally returned if the machine security policies require that the password should be changed because the password is NULL or it has aged.
PAM_ACCT_EXPIRED	User account has expired.
PAM_ACCT_DISABLED	User account has been disabled (trusted mode only).
PAM_TERM_DISABLED	Terminal has been disabled (trusted mode only).
PAM_NOT_AUTHORIZED	User is not authorized for terminal access (trusted mode only).
PAM_NOT_RTIME	Wrong time to login (trusted mode only).

SEE ALSO

pam(3), pam_start(3), pam_authenticate(3).

NAME

pam_authenticate - perform authentication within the PAM framework

SYNOPSIS

cc [*flag* ...] *file* ... **-lpam** [*library* ...]

```
#include <security/pam_appl.h>
int pam_authenticate(pam_handle_t *pamh, int flags);
```

DESCRIPTION

pam_authenticate() is called to authenticate the current user. The user is usually required to enter a password or similar authentication token depending upon the authentication service configured within the system. In the case of smart card authentication this token would be a **PIN** (Personal Identification Number). The user in question should have been specified by a prior call to **pam_start()** or **pam_set_item()**. The following flags may be set in the *flags* field:

PAM_SILENT
Authentication service should not generate any messages

PAM_DISALLOW_NULL_AUTHTOK
The authentication service should return **PAM_AUTH_ERROR** if the user has a null authentication token

NOTES

In the case of authentication failures due to an incorrect username or password, it is the responsibility of the application to retry **pam_authenticate()** and to maintain the retry count. An authentication service module may implement an internal retry count and return an error **PAM_MAXTRIES** if the module does not want the application to retry.

If the PAM framework can not load the authentication module, then it will return **PAM_ABORT**. This indicates a serious failure and that the application should not attempt to retry the authentication.

For security reasons, the location of authentication failures is hidden from the user. Thus, if several authentication services are stacked and a single service fails, **pam_authenticate()** requires that the user re-authenticate to all the services.

A null authentication token in the authentication database will result in successful authentication unless **PAM_DISALLOW_NULL_AUTHTOK** was specified. In such cases, there will not be any prompting for the user to enter an authentication token.

The authentication can be done through a smart card. In this case the user plugs their smart card in the smart card reader and is required to enter their smart card PIN.

RETURN VALUES

Upon successful completion, PAM_SUCCESS is returned. In addition to the error return values described in *pam*(3), the following values may be returned:

PAM_AUTH_ERR	Authentication failure.
PAM_CRED_INSUFFICIENT	Can not access authentication data due to insufficient credentials.
PAM_AUTHINFO_UNAVAIL	Underlying authentication service can not retrieve authentication information.
PAM_USER_UNKNOWN	User not known to the underlying authentication module.
PAM_MAXTRIES	An authentication service has maintained a retry count which has been reached. No further retries should be attempted.

SEE ALSO

pam(3), pam_start(3), pam_open_session(3), pam_setcred(3).

NAME

pam_chauthtok - perform password related functions within the PAM framework

SYNOPSIS

cc [*flag* ...] *file* ... **-lpam** [*library* ...]

```
#include <security/pam_appl.h>

int pam_chauthtok(pam_handle_t *pamh, int flags, int argc,
    const char **argv);
```

DESCRIPTION

pam_chauthtok() is called to change the authentication token associated with a particular user referenced by the authentication handle, *pamh*.

The following flag may be passed in to **pam_chauthtok()**:

PAM_SILENT
The password service should not generate any messages.

PAM_CHANGE_EXPIRED_AUTHTOK
The password service should only update those passwords that have aged. If this flag is not passed, all password services should update their passwords.

Upon successful completion of the call, the authentication token of the user will be changed in accordance with the password service configured in the system through *pam.conf*(4).

NOTES

The flag **PAM_CHANGE_EXPIRED_AUTHTOK** is typically used by a **login** application which has determined that the user's password has aged or expired. Before allowing the user to login, the **login** application may invoke **pam_chauthtok()** with this flag to allow the user to update the password. Typically applications such as *passwd*(1) should not use this flag.

pam_chauthtok() performs a preliminary check before attempting to update passwords. This check is performed for each password module in the stack as listed in *pam.conf*(4). The check may include pinging remote name services to determine if they are available. If **pam_chauthtok()** returns **PAM_TRY_AGAIN**, then the check has failed, and passwords are not updated.

RETURN VALUES

Upon successful completion, **PAM_SUCCESS** is returned. In addition to the error return values described in *pam*(3), the following values may be returned:

PAM_PERM_DENIED	No permission.
PAM_AUTHTOK_ERR	Authentication token manipulation error.
PAM_AUTHTOK_RECOVERY_ERR	Authentication information cannot be recovered.
PAM_AUTHTOK_LOCK_BUSY	Authentication token lock busy.
PAM_AUTHTOK_DISABLE_AGING	Authentication token aging disabled.
PAM_USER_UNKNOWN	User unknown to password service.
PAM_TRY_AGAIN	Preliminary check by password service failed.

SEE ALSO

pam(3), pam_start(3), pam_authenticate(3).

NAME

pam_get_user - PAM routine to retrieve user name.

SYNOPSIS

cc [*flag* ...] *file* ... **-lpam** [*library* ...]

```
#include <security/pam_appl.h>

int pam_get_user(pam_handle_t *pamh, char **user, const char *prompt);
```

DESCRIPTION

pam_get_user() is used by PAM service modules to retrieve the current user name from the PAM handle. If the user name has not been set, via **pam_start()** or **pam_set_item()**, then the PAM conversation function will be used to prompt the user for the user name with the string "prompt". If *prompt* is NULL, then **pam_get_item()** is called and the value of **PAM_USER_PROMPT** is used for prompting. If the value of **PAM_USER_PROMPT** is NULL, the following default prompt is used:

```
Please enter user name:
```

After the user name is gathered by the conversation function, **pam_set_item()** is called to set the value of **PAM_USER**.

By convention, applications that need to prompt for a user name should call **pam_set_item()** and set the value of **PAM_USER_PROMPT** before calling **pam_authenticate()**. The service module's **pam_sm_authenticate()** function will then call **pam_get_user()** to prompt for the user name. Note that certain PAM service modules (such as a smart card module) may override the value of **PAM_USER_PROMPT** and pass in their own prompt.

Applications that call **pam_authenticate()** multiple times should set the value of **PAM_USER** to NULL with **pam_set_item()** before calling **pam_authenticate()** if they want the user to be prompted for a new user name each time.

The value of *user* retrieved by **pam_get_user()** should not be modified or freed. The item will be released by **pam_end()**.

RETURN VALUES

Upon success, **pam_get_user()** returns **PAM_SUCCESS**; otherwise it returns an error code. Refer to *pam*(3) for information on error related return values.

SEE ALSO

pam(3), pam_start(3), pam_authenticate(3), pam_get_item(3), pam_set_item(3), pam_sm(3), pam_sm_authenticate(3), pam_end(3).

p

NAME

pam_open_session, pam_close_session - perform PAM session creation and termination operations

SYNOPSIS

cc [*flag* ...] *file* ... **-lpam** [*library* ...]

```
#include <security/pam_appl.h>
int pam_open_session(pam_handle_t *pamh, int flags);
int pam_close_session(pam_handle_t *pamh, int flags);
```

DESCRIPTION

pam_open_session() is called after a user has been successfully authenticated (refer to *pam_authenticate*(3) and *pam_acct_mgmt*(3)) and is used to notify the session modules that a new session has been initiated. All programs that use the *pam*(3) library should invoke **pam_open_session()** when beginning a new session. Upon termination of this activity, **pam_close_session()** should be invoked to inform *pam*(3) that the session has terminated.

The *pamh* argument is an authentication handle obtained by a prior call to **pam_start()**. The following flag may be set in the *flags* field for **pam_open_session()** and **pam_close_session()**:

PAM_SILENT The session service should not generate any messages.

RETURN VALUES

Upon successful completion, **PAM_SUCCESS** is returned. In addition to the return values defined in *pam*(3), the following value may be returned on error:

PAM_SESSION_ERR Can not make/remove an entry for the specified session.

SEE ALSO

pam(3), pam_start(3), pam_authenticate(3), pam_acct_mgmt(3).

p

NAME

pam_set_data, pam_get_data - PAM routines to maintain module specific state

SYNOPSIS

cc [*flag* ...] *file* ... **-lpam** [*library* ...]

```
#include <security/pam_appl.h>

int pam_set_data(pam_handle_t *pamh, const char *module_data_name,
    const void *data, void (*cleanup) (pam_handle_t *pamh, void *data,
    int pam_end_status));

int pam_get_data(const pam_handle_t *pamh, const char *module_data_name,
    void **data);
```

DESCRIPTION

pam_set_data() and **pam_get_data()** allow PAM service modules to access and update module specific information as needed. These functions should not be used by applications.

pam_set_data() stores module specific data within the PAM handle, *pamh*. The *module_data_name* argument uniquely identifies the data, and the *data* argument represents the actual data. *module_data_name* should be unique across all services (UNIX, etc).

The *cleanup* function is used to free any memory used by the *data* after it is no longer needed, and is invoked by **pam_end()**. The *cleanup* function takes as its arguments a pointer to the PAM handle, *pamh*, a pointer to the actual data, *data*, and a status code, *pam_end_status*. The status code determines exactly what state information needs to be purged, and is therefore specific to each module.

If **pam_set_data()** is called and module data already exists under the same *module_data_name* (from a prior call to **pam_set_data()**), then the existing *data* is replaced by the new *data*, and the existing *cleanup* function is replaced by the new *cleanup* function.

pam_get_data() retrieves module specific data stored in the PAM handle, *pamh*, identified by the unique name, *module_data_name*. The *data* argument is assigned the address of the requested data.

RETURN VALUES

In addition to the return values listed in *pam*(3), the following value may also be returned:

PAM_NO_MODULE_DATA No module specific data is present.

SEE ALSO

pam(3), pam_start(3).

NAME

pam_set_item, pam_get_item - authentication information routines for PAM

SYNOPSIS

cc [*flag* ...] *file* ... **-lpam** [*library* ...]

```
#include <security/pam_appl.h>
int pam_set_item(pam_handle_t *pamh, int item_type, const void *item);
int pam_get_item(const pam_handle_t *pamh, int item_type, void **item);
```

DESCRIPTION

pam_get_item() and **pam_set_item()** allow applications and PAM service modules to access and update PAM information as needed. The information is specified by *item_type,* and can be one of the following:

PAM_SERVICE	The service name.
PAM_USER	The user name.
PAM_AUTHTOK	The user authentication token.
PAM_OLDAUTHTOK	The old user authentication token.
PAM_TTY	The tty name.
PAM_RHOST	The remote host name.
PAM_RUSER	The remote user name.
PAM_CONV	The pam_conv structure.
PAM_USER_PROMPT	The default prompt used by **pam_get_user()**.

The *item_type* **PAM_AUTHTOK** and **PAM_OLDAUTHTOK** are available only to the module providers for security reasons. The authentication module, account module, and session management module should treat **PAM_AUTHTOK** as the current authentication token, and should ignore **PAM_OLDAUTHTOK**. The password management module should treat **PAM_OLDAUTHTOK** as the current authentication token and **PAM_AUTHTOK** as the new authentication token.

pam_set_item() is passed the authentication handle, *pamh,* returned by **pam_start()**, a pointer to the object, *item,* and its type, *item_type.* If successful, **pam_set_item()** copies the item to an internal storage area allocated by the authentication module and returns **PAM_SUCCESS**. An item that had been previously set will be overwritten by the new value.

pam_get_item() is passed the authentication handle, *pamh,* returned by **pam_start()**, an *item_type*, and the address of the pointer, *item*, which is assigned the address of the requested object. The object data is valid until modified by a subsequent call to **pam_set_item()** for the same *item_type*, or unless it is modified by any of the underlying service modules. If the item has not been previously set, **pam_get_item()** returns a NULL pointer. An *item* retrieved by **pam_get_item()** should not be modified or freed. The item will be released by **pam_end()**.

RETURN VALUES

Upon success, **pam_get_item()** returns **PAM_SUCCESS**; otherwise it returns an error code. Refer to *pam*(3) for information on error related return values.

SEE ALSO

pam_start(3), pam_authenticate(3), pam_acct_mgmt(3), pam_open_session(3), pam_setcred(3), pam_chauthtok(3), pam_get_user(3), pam(3).

NAME

pam_setcred - modify/delete user credentials for an authentication service

SYNOPSIS

cc [*flag* ...] *file* ... **-lpam** [*library* ...]

```
#include <security/pam_appl.h>
int pam_setcred(pam_handle_t *pamh, int flags);
```

DESCRIPTION

pam_setcred() is used to establish, modify, or delete user credentials. **pam_setcred()** is typically called after the user has been authenticated and after a session has been opened (refer to *pam_authenticate*(3), *pam_acct_mgmt*(3), and *pam_open_session*(3)).

The user is specified by a prior call to **pam_start()** or **pam_set_item()**, and is referenced by the authentication handle, *pamh*. The following flags may be set in the *flags* field. Note that the first four flags are mutually exclusive:

PAM_ESTABLISH_CRED Set user credentials for an authentication service.

PAM_DELETE_CRED Delete user credentials associated with an authentication service.

PAM_REINITIALIZE_CRED
Reinitialize user credentials.

PAM_REFRESH_CRED Extend lifetime of user credentials.

PAM_SILENT Authentication service should not generate any messages.

If none of the flags are set, **PAM_ESTABLISH_CRED** is used as the default.

RETURN VALUES

Upon success, **pam_setcred()** returns **PAM_SUCCESS**. In addition to the error return values described in *pam*(3), the following values may be returned upon error:

PAM_CRED_UNAVAIL Underlying authentication service can not retrieve user credentials unavailable.

PAM_CRED_EXPIRED User credentials expired.

PAM_USER_UNKNOWN User unknown to underlying authentication service.

PAM_CRED_ERR Failure setting user credentials.

SEE ALSO

pam(3), pam_start(3), pam_authenticate(3), pam_acct_mgmt(3), pam_open_session(3).

NAME

PAM - PAM Service Module APIs

SYNOPSIS

```
#include <security/pam_appl.h>
#include <security/pam_modules.h>
cc [ flag ... ] file ... -lpam [ library ... ]
```

DESCRIPTION

PAM gives system administrators the flexibility of choosing any authentication service available on the system to perform authentication. The framework also allows new authentication service modules to be plugged in and made available without modifying the applications.

The PAM framework, **libpam**, consists of an interface library and multiple authentication service modules. The PAM interface library is the layer implementing the Application Programming Interface (API). The authentication service modules are a set of dynamically loadable objects invoked by the PAM API to provide a particular type of user authentication.

This manual page gives an overview of the PAM APIs for the service modules.

Interface Overview

The PAM service module interface consists of functions which can be grouped into four categories. The names for all the authentication library functions start with **pam_sm**. The only difference between the **pam_*()** interfaces and their corresponding **pam_sm_*()** interfaces is that all the **pam_sm_*()** interfaces require extra parameters to pass service specific options to the shared modules. They are otherwise identical.

The first category contains functions to authenticate an individual user (*pam_sm_authenticate*(3)) and to set the credentials of the user (*pam_sm_setcred*(3)). These back-end functions implement the functionality of *pam_authenticate*(3) and *pam_setcred*(3), respectively.

The second category contains functions to do account management (*pam_sm_acct_mgmt*(3)). This includes checking for password aging and access-hour restrictions. This back-end function implements the functionality of *pam_acct_mgmt*(3).

The third category contains functions to perform session management (*pam_sm_open_session*(3) and *pam_sm_close_session*(3)) after access to the system has been granted. These back-end functions implement the functionality of *pam_open_session*(3) and *pam_close_session*(3), respectively.

The fourth category consists a function to change authentication tokens (*pam_sm_chauthtok*(3)). This back-end function implements the functionality of *pam_chauthtok*(3).

Stateful Interface

A sequence of calls sharing a common set of state information is referred to as an authentication transaction. An authentication transaction begins with a call to **pam_start()**. **pam_start()** allocates space, performs various initialization activities, and assigns an authentication handle to be used for subsequent calls to the library. Note that the service modules do not get called or initialized when **pam_start()** is called. The modules are loaded and the symbols resolved upon first use of that function.

The PAM handle keeps certain information about the transaction that can be accessed through the **pam_get_item()** API. Though the modules can also use **pam_set_item()** to change any of the item information, it is recommended that nothing be changed except **PAM_AUTHTOK** and **PAM_OLDAUTHTOK**.

If the modules want to store any module specific state information then they can use the *pam_set_data*(3) function to store that information with the PAM handle. The data should be stored with a name which is unique across all modules and module types. Some modules use this technique to share data across two different module types.

For example, during the call to **pam_authenticate()**, the UNIX module may store the authentication status (success or reason for failure) in the handle, using a unique name. This information is intended for use by **pam_setcred()**.

During the call to **pam_acct_mgmt()**, the account modules may store data in the handle to indicate which passwords have aged. This information is intended for use by **pam_chauthtok()**.

The module can also store a cleanup function associated with the data. The PAM framework calls this cleanup function, when the application calls **pam_end()** to close the transaction.

Interaction With the User

The PAM service modules do not communicate directly with the user; instead they rely on the application to perform all such interactions. The application passes a pointer to the function, **conv( )**, along with any associated application data pointers, through the **pam_conv** structure when it initiates an authentication transaction (via a call to **pam_start( )**). The service module will then use the function, **conv( )**, to prompt the user for data, output error messages, and display text information. Refer to *pam_start*(3) for more information. The modules are responsible for the localization of all messages to the user.

Conventions

By convention, applications that need to prompt for a user name should call **pam_set_item( )** and set the value of **PAM_USER_PROMPT** before calling **pam_authenticate( )**. The service module's **pam_sm_authenticate( )** function will then call **pam_get_user( )** to prompt for the user name. Note that certain PAM service modules (such as a smart card module) may override the value of **PAM_USER_PROMPT** and pass in their own prompt.

Though the PAM framework enforces no rules about the module's names, location, options and such, there are certain conventions that all module providers are expected to follow.

By convention, the modules should be located in the **/usr/lib/security** directory.

The modules are named **libpam**_*service_name*.**1** (for example, **libpam_unix.1** module).

For every such module, there should be a corresponding manual page in section 5 which should describe the *service_name* it supports, the functionality of the module, along with the options it supports. The dependencies should be clearly identified to the system administrator. For example, it should be made clear whether this module is a stand-alone module or depends upon the presence of some other module. One should also specify whether this module should come before or after some other module in the stack.

By convention, the modules should support the following options:

debug Syslog debugging information at LOG_DEBUG level. Be careful as to not log any sensitive information such as passwords.

nowarn Turn off warning messages such as "password is about to expire"

In addition, it is recommended that the auth and the password module support the following options:

use_first_pass
Instead of prompting the user for the password, use the user's initial password (entered when the user was authenticated to the first authentication module in the stack) for authentication. If the passwords do not match, or if no password has been entered, return failure and do not prompt the user for a password. Support for this scheme allows the user to type only one password for multiple schemes.

try_first_pass
Instead of prompting the user for the password, use the user's initial password (entered when the user was authenticated to the first authentication module in the stack) for authentication. If the passwords do not match, or if no password has been entered, prompt the user for a password after identifying which type of password (ie. UNIX, etc.) is being requested. Support for this scheme allows the user to try to use only one password for multiple schemes, and type multiple passwords only if necessary.

use_psd Instead of prompting the user for the password, prompt for the user's PIN (Personal Idenfitication Number) associated with their smart card. This allows the smart card to be accessed, from which the password can be retrieved. With this option the user must plug their smart card in the smart card reader connected to their system..

If an unsupported option is passed to the modules, it should syslog the error at LOG_ERR level.

The permission bits on the service module should be set such that it is not writable by either "group" or "other". The PAM framework will not load the module if the above permission rules are not followed.

ERRORS

If there are any errors, the modules should log them using *syslog*(3C) at the LOG_ERR level.

RETURN VALUES

The PAM service module functions may return any of the PAM error numbers specified in the specific man pages. It can also return a PAM_IGNORE error number to mean that the PAM framework should ignore this module regardless of whether it is required, optional or sufficient. This error number is normally returned when the module does not want to deal with the given user at all.

SEE ALSO

pam(3), pam_start(3), pam_set_item(3), pam_get_user(3), pam_authenticate(3), pam_open_session(3), pam_setcred(3), pam_chauthtok(3), pam_strerror(3), pam_sm_authenticate(3), pam_sm_open_session(3), pam_sm_setcred(3), pam_sm_chauthtok(3), pam.conf(4), pam_user.conf(4).

NAME

pam_sm_acct_mgmt - Service provider implementation for pam_acct_mgmt

SYNOPSIS

cc [*flag* ...] *file* ... **-lpam** [*library* ...]

```
#include <security/pam_appl.h>
#include <security/pam_modules.h>
int pam_sm_acct_mgmt(pam_handle_t *pamh, int flags, int argc,
    const char **argv);
```

DESCRIPTION

In response to a call to *pam_acct_mgmt*(3), the PAM framework calls **pam_sm_acct_mgmt()** from the modules listed in the *pam.conf*(4) file. The account management provider supplies the back-end functionality for this interface function. The applications should not call this API directly.

The function, **pam_sm_acct_mgmt()**, determines whether the current user's account and password are valid. This includes checking for password and account expiration, valid log-in times, etc. The user in question is specified by a prior call to **pam_start()**, and is referenced by the authentication handle, *pamh*, which is passed as the first argument to **pam_sm_acct_mgmt()**. The following flags may be set in the *flags* field:

PAM_SILENT
The account management service should not generate any messages.

PAM_DISALLOW_NULL_AUTHTOK
The account management service should return **PAM_AUTHTOKEN_REQD** if the user has a null authentication token.

The *argc* argument represents the number of module options passed in from the configuration file *pam.conf*(4). *argv* specifies the module options, which are interpreted and processed by the account management service. Please refer to the specific module man pages for the various available *options*. If an unknown option is passed to the module, an error should be logged through *syslog*(3C) and the option ignored.

If an account management module determines that the user password has aged or expired, it should save this information as state in the authentication handle, *pamh*, using **pam_set_data()**. **pam_chauthtok()** uses this information to determine which passwords have expired.

RETURN VALUES

If there are no restrictions to logging in, **PAM_SUCCESS** is returned. The following error values may also be returned upon error:

PAM_USER_UNKNOWN	User not known to underlying authentication module.
PAM_AUTHTOKEN_REQD	New authentication token required.
PAM_ACCT_EXPIRED	User account has expired.
PAM_PERM_DENIED	User denied access to account at this time.
PAM_IGNORE	Ignore underlying account module regardless of whether the control flag is **required**, **optional** or **sufficient**
PAM_ACCT_DISABLED	User account has been disabled (trusted mode only).
PAM_TERM_DISABLED	Terminal has been disabled (trusted mode only).
PAM_NOT_AUTHORIZED	User is not authorized for terminal access (trusted mode only).
PAM_NOT_RTIME	Wrong time to login (trusted mode only).

SEE ALSO

pam(3), pam_acct_mgmt(3), syslog(3C), pam.conf(4).

NAME

pam_sm_authenticate - Service provider implementation for pam_authenticate

SYNOPSIS

cc [*flag* ...] *file* ... **-lpam** [*library* ...]

```
#include <security/pam_appl.h>
#include <security/pam_modules.h>
int pam_sm_authenticate(pam_handle_t *pamh, int flags, int argc,
    const char **argv);
```

DESCRIPTION

In response to a call to *pam_authenticate*(3), the PAM framework calls **pam_sm_authenticate()** from the modules listed in the *pam.conf*(4) file. The authentication provider supplies the back-end functionality for this interface function.

The function, **pam_sm_authenticate()**, is called to verify the identity of the current user. The user is usually required to enter a password or similar authentication token depending upon the authentication scheme configured within the system. The user in question is specified by a prior call to **pam_start()**, and is referenced by the authentication handle, *pamh*.

If the user is unknown to the authentication service, the service module should mask this error and continue to prompt the user for a password. It should then return the error, **PAM_USER_UNKNOWN**.

The following flag may be passed in to **pam_sm_authenticate()**:

PAM_SILENT
The authentication service should not generate any messages.

PAM_DISALLOW_NULL_AUTHTOK
The authentication service should return **PAM_AUTH_ERROR** if the user has a null authentication token.

The *argc* argument represents the number of module options passed in from the configuration file *pam.conf*(4). *argv* specifies the module options, which are interpreted and processed by the authentication service. Please refer to the specific module manual pages for the various available *options.* If any unknown option is passed in, the module should log the error and ignore the option.

Before returning, **pam_sm_authenticate()** should call **pam_get_item()** and retrieve **PAM_AUTHTOK**. If it has not been set before (ie. the value is NULL), **pam_sm_authenticate()** should set it to the password entered by the user using **pam_set_item()**.

An authentication module may save the authentication status (success or reason for failure) as state in the authentication handle using **pam_set_data()**. This information is intended for use by **pam_setcred()**.

NOTES

Modules should not retry the authentication in the event of a failure. Applications handle authentication retries and maintain the retry count. To limit the number of retries, the module can return a **PAM_MAXTRIES** error.

RETURN VALUES

Upon successful completion, **PAM_SUCCESS** must be returned. In addition, the following values may be returned:

PAM_MAXTRIES	Maximum number of authentication attempts exceeded.
PAM_AUTH_ERR	Authentication failure.
PAM_CRED_INSUFFICIENT	Can not access authentication data due to insufficient credentials.
PAM_AUTHINFO_UNAVAIL	Underlying authentication service can not retrieve authentication information.
PAM_USER_UNKNOWN	User not known to underlying authentication module.
PAM_IGNORE	Ignore underlying authentication module regardless of whether the control flag is **required**, **optional** or **sufficient**.

SEE ALSO
pam(3), pam_authenticate(3), pam.conf(4), pam_user.conf(4).

NAME

pam_sm_chauthtok - Service provider implementation for pam_chauthtok

SYNOPSIS

cc [*flag* ...] *file* ... **-lpam** [*library* ...]

```
#include <security/pam_appl.h>
#include <security/pam_modules.h>
int pam_sm_chauthtok(pam_handle_t *pamh, int flags, int argc,
    const char **argv);
```

DESCRIPTION

In response to a call to **pam_chauthtok()** the PAM framework calls **pam_sm_chauthtok()** from the modules listed in the *pam.conf*(4) file. The password management provider supplies the back-end functionality for this interface function.

pam_sm_chauthtok() changes the authentication token associated with a particular user referenced by the authentication handle, *pamh*.

The following flag may be passed in to **pam_chauthtok()**:

PAM_SILENT The password service should not generate any messages.

PAM_CHANGE_EXPIRED_AUTHTOK
The password service should only update those passwords that have aged. If this flag is not passed, the password service should update all passwords.

PAM_PRELIM_CHECK The password service should only perform preliminary checks. No passwords should be updated.

PAM_UPDATE_AUTHTOK The password service should update passwords.

Note that **PAM_PRELIM_CHECK** and **PAM_UPDATE_AUTHTOK** can not be set at the same time.

Upon successful completion of the call, the authentication token of the user will be ready for change or will be changed (depending upon the flag) in accordance with the authentication scheme configured within the system.

The *argc* argument represents the number of module options passed in from the configuration file *pam.conf*(4). *argv* specifies the module options, which are interpreted and processed by the password management service. Please refer to the specific module man pages for the various available *options*.

It is the responsibility of **pam_sm_chauthtok()** to determine if the new password meets certain strength requirements. **pam_sm_chauthtok()** may continue to re-prompt the user (for a limited number of times) for a new password until the password entered meets the strength requirements.

Before returning, **pam_sm_chauthtok()** should call **pam_get_item()** and retrieve both **PAM_AUTHTOK** and **PAM_OLDAUTHTOK**. If both are NULL, **pam_sm_chauthtok()** should set them to the new and old passwords as entered by the user.

NOTES

The PAM framework invokes the password services twice. The first time the modules are invoked with the flag, **PAM_PRELIM_CHECK**. During this stage, the password modules should only perform preliminary checks (ping remote name services to see if they are ready for updates, for example). If a password module detects a transient error (remote name service temporarily down, for example) it should return **PAM_TRY_AGAIN** to the PAM framework, which will immediately return the error back to the application. If all password modules pass the preliminary check, the PAM framework invokes the password services again with the flag, **PAM_UPDATE_AUTHTOK**. During this stage, each password module should proceed to update the appropriate password. Any error will again be reported back to application.

If a service module receives the flag, **PAM_CHANGE_EXPIRED_AUTHTOK**, it should check whether the password has aged or expired. If the password has aged or expired, then the service module should proceed to update the password. If the status indicates that the password has not yet aged/expired, then the password module should return **PAM_IGNORE**.

If a user's password has aged or expired, a PAM account module could save this information as state in the authentication handle, pamh, using **pam_set_data()**. The related password management module could retrieve this information using **pam_get_data()** to determine whether or not it should prompt the user to update the password for this particular module.

RETURN VALUES

Upon successful completion, **PAM_SUCCESS** must be returned. The following values may also be returned:

PAM_PERM_DENIED	No permission.
PAM_AUTHTOK_ERR	Authentication token manipulation error.
PAM_AUTHTOK_RECOVERY_ERR	Old authentication token cannot be recovered.
PAM_AUTHTOK_LOCK_BUSY	Authentication token lock busy.
PAM_AUTHTOK_DISABLE_AGING	Authentication token aging disabled.
PAM_USER_UNKNOWN	User unknown to password service.
PAM_TRY_AGAIN	Preliminary check by password service failed.

SEE ALSO

pam(3), pam_chauthtok(3), pam.conf(4).

NAME

pam_sm_open_session, pam_sm_close_session - Service provider implementation for pam_open_session and pam_close_session respectively

SYNOPSIS

cc [*flag* ...] *file* ... **-lpam** [*library* ...]

```
#include <security/pam_appl.h>

#include <security/pam_modules.h>

int pam_sm_open_session(pam_handle_t *pamh, int flags, int argc,
    const char **argv);

int pam_sm_close_session(pam_handle_t *pamh, int flags, int argc,
    const char **argv);
```

DESCRIPTION

In response to a call to **pam_open_session()** and **pam_close_session()**, the PAM framework calls **pam_sm_open_session()** and **pam_sm_close_session()**, respectively from the modules listed in the *pam.conf*(4) file. The session management provider supplies the back-end functionality for this interface function.

pam_sm_open_session() is called to initiate session management. **pam_sm_close_session()** is invoked when a session has terminated. The argument *pamh* is an authentication handle. The following flag may be set in the *flags* field:

PAM_SILENT Session service should not generate any messages.

The *argc* argument represents the number of module options passed in from the configuration file *pam.conf*(4). *argv* specifies the module options, which are interpreted and processed by the session management service. If an unknown option is passed in, an error should be logged through *syslog*(3C) and the option ignored.

RETURN VALUES

Upon successful completion, **PAM_SUCCESS** should be returned. The following values may also be returned upon error:

PAM_SESSION_ERR Can not make/remove an entry for the specified session.

PAM_IGNORE Ignore underlying session module regardless of whether the control flag is **required**, **optional** or **sufficient**

SEE ALSO

pam(3), pam_open_session(3), syslog(3C), pam.conf(4).

NAME

pam_sm_setcred - Service provider implementation for pam_setcred

SYNOPSIS

cc [*flag* ...] *file* ... **-lpam** [*library* ...]

```
#include <security/pam_appl.h>
#include <security/pam_modules.h>
int pam_sm_setcred(pam_handle_t *pamh, int flags, int argc,
    const char **argv);
```

DESCRIPTION

In response to a call to **pam_setcred()**, the PAM framework calls **pam_sm_setcred()** from the modules listed in the *pam.conf*(4) file. The authentication provider supplies the back-end functionality for this interface function.

pam_sm_setcred() is called to set the credentials of the current user associated with the authentication handle, *pamh*. The following flags may be set in the *flags* field. Note that the first four flags are mutually exclusive:

PAM_CRED_ESTABLISH	Set user credentials for the authentication service.
PAM_CRED_DELETE	Delete user credentials associated with the authentication service.
PAM_CRED_REINITIALIZE	Reinitialize user credentials.
PAM_CRED_REFRESH	Extend lifetime of user credentials.
PAM_SILENT	Authentication service should not generate messages.

If none of these flags are set, **PAM_CRED_ESTABLISH** is used as the default.

The *argc* argument represents the number of module options passed in from the configuration file *pam.conf*(4). *argv* specifies the module options, which are interpreted and processed by the authentication service. If an unknown option is passed to the module, an error should be logged and the option ignored.

If the **PAM_SILENT** flag is not set, then **pam_sm_setcred()** should print any failure status from the corresponding **pam_sm_authenticate()** function using the conversation function.

The authentication status (success or reason for failure) is saved as module-specific state in the authentication handle by the authentication module. The status should be retrieved using **pam_get_data()**, and used to determine if user credentials should be set.

NOTES

pam_sm_setcred() is passed the same module options that are used by **pam_sm_authenticate()**.

RETURN VALUES

Upon successful completion, **PAM_SUCCESS** should be returned. The following values may also be returned upon error:

PAM_CRED_UNAVAIL	Underlying authentication service can not retrieve user credentials.
PAM_CRED_EXPIRED	User credentials have expired.
PAM_USER_UNKNOWN	User unknown to the authentication service.
PAM_CRED_ERR	Failure in setting user credentials.
PAM_IGNORE	Ignore underlying authentication module regardless of whether the control flag is **required**, **optional** or **sufficient.**

SEE ALSO

pam(3), pam_authenticate(3), pam_setcred(3), pam_sm_authenticate(3), pam.conf(4).

NAME

pam_start, pam_end - authentication transaction routines for PAM

SYNOPSIS

cc [*flag* ...] *file* ... **-lpam** [*library* ...]

```
#include <security/pam_appl.h>

int pam_start(const char *service, const char *user,
    const struct pam_conv *pam_conv, pam_handle_t **pamh);

int pam_end(pam_handle_t *pamh, int status);
```

DESCRIPTION

pam_start() is called to initiate an authentication transaction. **pam_start()** takes as arguments the name of the current service, *service*, the name of the user to be authenticated, *user*, the address of the conversation structure, *pam_conv*, and the address of a variable to be assigned the authentication handle, *pamh*.

Upon successful completion, *pamh* will refer to a PAM handle for use with subsequent calls to the authentication library.

The pam_conv structure, *pam_conv*, contains the address of the conversation function provided by the application. The underlying PAM service module invokes this function to output information to and retrieve input from the user. The *pam_conv* structure has the following entries:

```
struct pam_conv {
        int     (*conv)();          /* Conversation function */
        void    *appdata_ptr;       /* Application data */
};
```

where **conv** is:

```
int conv(int num_msg,
        const struct pam_message **msg, struct pam_response **resp,
        void *appdata_ptr);
```

The function **conv()** is called by a service module to hold a PAM conversation with the application or user. For window applications, the application can create a new pop-up window to be used by the interaction.

The parameter *num_msg* is the number of messages associated with the call. The parameter *msg* is a pointer to an array of length *num_msg* of the *pam_message* structure.

The structure *pam_message* is used to pass prompt, error message, or any text information from the authentication service to the application or user. It is the responsibility of the PAM service modules to localize the messages. The memory used by *pam_message* has to be allocated and freed by the PAM modules. The *pam_message* structure has the following entries:

```
struct pam_message{
        int     msg_style;
        char    *msg;
};
```

The message style, *msg_style*, can be set to one of the following values:

PAM_PROMPT_ECHO_OFF Prompt user, disabling echoing of response.

PAM_PROMPT_ECHO_ON Prompt user, enabling echoing of response.

PAM_ERROR_MSG Print error message.

PAM_TEXT_INFO Print general text information.

The maximum size of the message and the response string is **PAM_MAX_MSG_SIZE** defined in <**security/pam_appl.h**>.

The structure *pam_response* is used by the authentication service to get the user's response back from the application or user. The storage used by *pam_response* has to be allocated by the application and freed by the PAM modules. The *pam_response* structure has the following entries:

```
struct pam_response{
        char    *resp;
        int     resp_retcode; /* currently not used, should be set to 0 */
};
```

It is the responsibility of the conversation function to strip off newline characters for **PAM_PROMPT_ECHO_OFF** and **PAM_PROMPT_ECHO_ON** message styles, and to add newline characters (if appropriate) for **PAM_ERROR_MSG** and **PAM_TEXT_INFO** message styles.

appdata_ptr is an application data pointer which is passed by the application to the PAM service modules. Since the PAM modules pass it back through the conversation function, the applications can use this pointer to point to any application-specific data.

pam_end() is called to terminate the authentication transaction identified by *pamh* and to free any storage area allocated by the authentication module. The argument, *status*, is passed to the **cleanup()** function stored within the pam handle, and is used to determine what module specific state must be purged. A cleanup function is attached to the handle by the underlying PAM modules through a call to *pam_set_item*(3) to free module specific data.

RETURN VALUES

Refer to *pam*(3) for information on error related return values.

SEE ALSO

pam_authenticate(3), pam_set_item(3), pam_acct_mgmt(3), pam_open_session(3), pam_setcred(3), pam_chauthtok(3), pam_strerror(3), pam(3).

NAME

pam_strerror - get PAM error message string

SYNOPSIS

cc [*flag* ...] *file* ... **-lpam** [*library* ...]

```
#include <security/pam_appl.h>
const char *pam_strerror(pam_handle_t *pamh, int errnum);
```

DESCRIPTION

pam_strerror() maps the PAM error number in *errnum* to a PAM error message string, and returns a pointer to that string. The application should not free or modify the string returned.

The *pamh* argument is the PAM handle obtained by a prior call to **pam_start()**. If **pam_start()** returns an error, a NULL PAM handle should be passed.

ERRORS

pam_strerror() returns NULL if *errnum* is out-of-range.

SEE ALSO

pam(3), pam_start(3).

p

NAME

pathfind() - search for named file in named directories

SYNOPSIS

```
#include <libgen.h>
char *pathfind (const char *path, const char *name, const char *mode);
```

DESCRIPTION

pathfind searches the directories named in *path* for the file *name*. The directories named in *path* are separated by colons. *mode* is a string of option letters chosen from the set **rwxfbcdpugks**:

Letter	Meaning
r	readable
w	writable
x	executable
f	normal file
b	block special
c	character special
d	directory
p	FIFO (pipe)
u	set user ID bit
g	set group ID bit
k	sticky bit
s	size nonzero

Options read, write, and execute are checked relative to the real (not the effective) user ID and group ID of the current process.

If the file *name*, with all the characteristics specified by *mode*, is found in any of the directories specified by *path*, then **pathfind** returns a pointer to a string containing the member of *path*, followed by a slash character (**/**), followed by *name*.

If *name* begins with a slash, it is treated as an absolute path name, and *path* is ignored.

An empty *path* member is treated as the current directory. **.** is not prepended at the occurrence of the first match; rather, the unadorned *name* is returned.

RETURN VALUE

If no match is found, **pathname** returns a null pointer, **((char *) 0)**.

APPLICATION USAGE

pathfind is thread-safe and async-cancel-safe.

EXAMPLES

To find the **ls** command using the **PATH** environment variable:

```
pathfind (getenv ("PATH"), "ls", "rx")
```

WARNINGS

The string pointed to by the returned pointer is stored in a static area that is reused on subsequent calls to **pathfind**.

SEE ALSO

access(2), getenv(3C), mknod(2), sh(1), stat(2), test(1).

NAME

pechochar, pecho_wchar — write a character and rendition and immediately refresh the pad

SYNOPSIS

```
#include <curses.h>
int pechochar(WINDOW *pad, chtype ch);
int pecho_wchar(WINDOW *pad, const cchar_t *wch);
```

DESCRIPTION

The **pechochar()** and **pecho_wchar()** functions output one character to a pad and immediately refresh the pad. They are equivalent to a call to **waddch()** or **wadd_wch()**, respectively, followed by a call to **prefresh()**. The last location of the pad on the screen is reused for the arguments to **prefresh()**.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

The **pechochar()** function is only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the A_ prefix.

SEE ALSO

echochar(3X), newpad(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

p

NAME

perror(), strerror(), errno, sys_errlist, sys_nerr - system error messages

SYNOPSIS

```
#include <stdio.h>
void perror(const char *s);
#include <string.h>
char *strerror(int errnum);
#include <errno.h>
extern char *sys_errlist[];
extern int sys_nerr;
```

Obsolescent Interface

```
#include <string.h>
int strerror_r(int errnum, char *buffer, int buflen);
```

DESCRIPTION

perror() writes a language-dependent message to 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, followed by a colon, a blank, the message, and a new-line. To be most useful, the argument string should include the name of the program that incurred the error. The error number is taken from the symbol **errno**, which is set when errors occur but not cleared when non-erroneous calls are made. The contents of the message is identical to those returned by the **strerror()** function with **errno** as the argument. If given a NULL string, the **perror()** function prints only the message and a new-line.

To simplify variant formatting of messages, the **strerror()** function and the **sys_errlist** array of message strings are provided. The **strerror()** function maps the error number in *errnum* to a language-dependent error message string and returns a pointer to the string. The message string is returned without a new-line. **errno** can be used as an index into **sys_errlist** to get an untranslated 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 might be added to the system before they are added to the table. **strerror()** must be used to retrieve messages when translations are desired.

Obsolescent Interface

strerror_r() handles system error messages.

APPLICATION USAGE

perror() and **strerror()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing **perror()** or **strerror()**.

EXTERNAL INFLUENCES

Environment Variables

The language of the message returned by **strerror()** and printed by **perror()** is specified by the **LANG** environment variable. If the language-dependent message is not available, or if **LANG** is not set or is set to the empty string, the default version of the message associated with the "C" language (see *lang*(5)) is used.

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

perror() returns no value.

If the *errnum* message number is valid, **strerror()** returns a pointer to a language-dependent message string. The array pointed to should not be modified by the program, and might be overwritten by a subsequent call to the function. If a valid *errnum* message number does not have a corresponding language-dependent message, **strerror()** uses *errnum* as an index into **sys_errlist** to get the message string. If the *errnum* message number is invalid, **strerror()** returns a pointer to a NULL string.

WARNINGS

The return value for **strerror()** points to data whose content is overwritten by subsequent calls to **strerror()** from the same thread.

strerror_r() is an obsolescent interface supported only for compatibility with existing DCE applications. New multithreaded applications should use **strerror()**.

SEE ALSO

errno(2), lang(5), environ(5).

STANDARDS CONFORMANCE

perror(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

strerror(): AES, SVID3, XPG3, XPG4, ANSI C

sys_errlist(): SVID2, SVID3, XPG2

sys_nerr(): SVID2, SVID3, XPG2

p

NAME

pfmt(), vpfmt() - display message in standard format

SYNOPSIS

```
#include <pfmt.h>
int pfmt(FILE *stream, long flags, char *fmt, /* [arg, ] */ ...);

#include <stdarg.h>
#include <pfmt.h>
int vpfmt(FILE *stream, long flags, char *fmt, va_list ap);
```

DESCRIPTION

The **pfmt()** system call can be used to write a message in standard format to *stream*. It can also be used to write a localized string to *stream*. The arguments to **pfmt()** are formatted using **printf()** style formatting. **vpfmt()** is similar to **pfmt()** except that the arguments are passed in an argument list (see *stdarg*(5)).

The standard format displayed to *stream* has the following fields:

label:*severity*:*text*

The *label* string is defined through *setlabel*(3C). If no label is defined, this field is not used. The *severity* string is controlled by the *severity* group of *flags*. The *text* string is the formatted user string.

The *flags* control how formatting is done. The control information is separated into several different groups. Only one flag from each group should be set.

Output Format

MM_NOSTD Do not use the standard format. Treat *fmt* as a **printf()** format string. In this mode, only flags related to catalog access can be set.

MM_STD Format using the standard format (**MM_STD** is the default).

Catalog Access

MM_NOGET Do not access the localized message catalog. Use the *def_str* (a field in *fmt)* as the format string.

MM_GET Access the localized message catalog (**MM_GET** is the default).

Severity

MM_HALT Display a localized **HALT** string.

MM_ERROR Display a localized **ERROR** string (**MM_ERROR** is the default).

MM_WARNING Display a localized **WARNING** string.

MM_INFO Display a localized **INFO** string.

Besides these reserved severities, additional severity strings may be defined by the user (see *addsev*(3C)). To specify a user defined severity, *flags* should be a logical-or of the numeric value of the user defined severity and flags from other control groups.

Action

MM_ACTION This flag generates the localized string of **TO FIX:** in the *severity* field of the standard format message. If this flag and flags from the severity control group are set, this flag has precedence and the **TO FIX:** string will be displayed.

The *fmt* string has the following fields:

catalog:*msg_number*:*def_str*

The *catalog* is the message catalog created by *mkmsgs*(1) where the localized message is to be retrieved. The *msg_number* is a positive index number identifying the string to be retrieved from the message catalog (begins at 1). The *def_str* is the default string to use if **pfmt()** fails to retrieve the message from *catalog* from either the current locale or the default locale. The failure may occur if the message catalog does not exist or if the *msg_number* is out of bound.

If *catalog* is not specified, **pfmt()** uses the message catalog defined by *setcat*(3C). If **MM_NOGET** is set in *flags*, only *def_str* must be specified.

The **pfmt()** system call displays **Message not found!!** under the following conditions:

- No message catalog is specified in *fmt* and no catalog is defined via *setcat*(3C).
- *msg_number* is not positive.
- No message could be retrieved and *def_str* is not specified.

APPLICATION USAGE

pfmt() and **vpfmt()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing **pfmt()** or **vpfmt()**.

RETURN VALUE

If successful, **pfmt()** and **vpfmt()** return the number of bytes written. Otherwise they return a negative value.

EXAMPLES

Example 1

```
setlabel("UX:my_appl");
pfmt(stderr, MM_INFO,"MY_cat:1:file is writable");
```

generates the message:

```
UX:my_appl: INFO: file is writable
```

Example 2

```
setlabel("");
setcat("MY_cat");
pfmt(stderr, MM_ERROR,":1:%s is writable", "my_file");
```

generates the message:

```
ERROR: my_file is writable
```

Example 3

```
setlabel("");
setcat("MY_cat");
pfmt(stderr, MM_NOSTD,":1:%s is writable", "my_file");
```

generates the message:

```
my_file is writable
```

Example 4

```
#define MM_USER 10
setlabel("");
addsev(MM_USER, "MY_NOTE");
pfmt(stderr, MM_USER|MM_GET,"MY_cat:1:%s is writable", "my_file");
```

generates the message:

```
MY_NOTE: my_file is writable
```

SEE ALSO

mkmsgs(1), addsev(3C), gettxt(3C), setcat(3C), setlabel(3C), setlocale(3C), printf(3S), stdarg(5).

STANDARDS COMPLIANCE

pfmt(): SVID3

vpfmt(): SVID3

NAME

popen(), pclose() - initiate pipe I/O to/from a process

SYNOPSIS

```
#include <stdio.h>

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

int pclose(FILE *stream);
```

DESCRIPTION

popen() creates a pipe between the calling program and a command to be executed by the POSIX shell, **/usr/bin/sh** (see *sh-posix*(1)).

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() returns 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 can be used as an input filter and a type **w** command as an output filter.

APPLICATION USAGE

popen() and **pclose()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing **popen()** or **pclose()**.

RETURN VALUE

popen() returns a NULL pointer if files or processes cannot be created. The success of the command execution can be checked by examining the return value of **pclose()**.

pclose() returns **-1** if *stream* is not associated with a **popen()**ed command, or 127 if **/usr/bin/sh** could not be executed for some reason.

WARNINGS

If the original and **popen()**ed processes concurrently read or write a common file, neither should use buffered I/O because the buffering will not work properly. Problems with an output filter can be forestalled by careful buffer flushing, e.g., with **fflush()**; see *fclose*(3S).

SEE ALSO

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

STANDARDS CONFORMANCE

popen(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, POSIX.2

pclose(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, POSIX.2

NAME

pow(), powf() - power functions

SYNOPSIS

```
#include <math.h>
double pow(double x, double y);
float powf(float x, float y);
```

DESCRIPTION

The **pow()** function returns x^y. If *x* is negative, *y* must be an integer value.

The **powf()** function is a **float** version of **pow()**; it takes **float** arguments and returns a **float** result. To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options.

powf() is not specified by any standard, but it is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, make sure your program includes <**math.h**>, and link in the math library by specifying **-lm** on the compiler or linker command line.

Millicode versions of the **pow()** function are available. Millicode versions of math library functions are usually faster than their counterparts in the standard library. To use these versions, compile your program with the **+Olibcalls** or the **+Oaggressive** optimization option.

If an error occurs, the millicode versions return the value described in the *RETURN VALUE* section, but do not set **errno**.

For more information, see the *HP-UX Floating-Point Guide*.

RETURN VALUE

If *x* and *y* are both zero, **pow()** returns 1.0.

If *x* or *y* is NaN, **pow()** returns NaN.

If the correct value after rounding would be smaller in magnitude than **MINDOUBLE**, **pow()** returns zero.

If *x* is zero and *y* is less than zero, **pow()** returns −**HUGE_VAL** and sets **errno** to [EDOM].

If *x* is less than zero and *y* is not an integer, **pow()** returns NaN and sets **errno** to [EDOM].

If the correct value would overflow, **pow()** returns ±**HUGE_VAL** and sets **errno** to [ERANGE].

ERRORS

If **pow()** fails, **errno** is set to one of the following values.

[EDOM]	*x* is zero and *y* is less than zero.
[EDOM]	*x* is less than zero and *y* is not an integer.
[ERANGE]	The correct value would overflow.

SEE ALSO

cbrt(3M), exp(3M), log(3M), sqrt(3M), math(5), values(5).

STANDARDS CONFORMANCE

pow(): SVID3, XPG4.2, ANSI C

NAME

prcmd - return streams to parallel remote commands

SYNOPSIS

```
#include <prcmd.h>

void prcmd_init (
     struct prc_host *hostp,
     int num_hosts,
     int caller_status,
     char *command,
     time_t timeout
);

int prcmd (
     struct prc_host *hostp,
     int num_hosts
);
```

Remarks

These functions reside in libdc, and are linked using the **-ldc** option to the **ld** or **cc** command.

DESCRIPTION

prcmd() is an interface similar to **rcmd()** that allows a client-side calling program to use an existing **rcmd()** server side daemon. **prcmd()**, however, performs in parallel across multiple systems, the time consuming steps associated with executing commands remotely (connection, command invocation, and command execution/interaction). With **rcmd()**, only command execution and interaction is under the control of the calling program, and it requires the calling program to perform its own **select()** calls in order to manage multiple (parallel) connections.

Features of the existing server-side remote-shell daemon (**remshd**) include: well known service, access control, and shell processing of the command line.

The calling program passes **prcmd()** a pointer (*hostp*) to the first node in a list of hosts for which to initiate, or continue, remote command processing via Internet socket connections. It also supplies the number of hosts in the list (*num_hosts*).

The calling program makes repeated calls to **prcmd()** to advance the status of the host connections. Each call to **prcmd()** causes a single call of **select()** to check the status of all connections that require checking (if there are any). Once a connection (to a remote command) is ready for reading, the calling program can read data from it, optionally write data to it, and then wait for more data to be ready to read back from it.

Header File

The **<prcmd.h>** header file defines a **prc_host** structure containing the following fields of interest:

```
char     *prc_hostid;         /* host name or IP address      */
int      prc_prev_status;     /* previous connection status   */
int      prc_conn_status;     /* connection status            */
time_t   prc_conn_time;       /* time of connection           */
int      prc_errno;           /* for failed connections       */
char     *prc_errmsg;         /* string from remshd           */
FILE     *prc_fp;             /* file ptr to stdin/stdout     */
FILE     *prc_fp2;            /* file pointer to stderr       */
int      prc_conn_close;      /* flag: close connection       */
int      prc_caller_status;   /* caller's info about conn     */
```

The calling program should change only the **prc_hostid**, **prc_conn_close**, **prc_caller_status**, and possibly **prc_conn_time** fields, as explained below. The other fields should be considered read-only.

The header file also defines the following macros of interest:

```
PRC_OK              /* function succeeded, check entries          */
PRC_ERR_NETWORKING  /* no networking on calling program's system  */
PRC_ERR_NOFILE      /* cannot get even one file descriptor        */
PRC_ERR_RCMD        /* cannot get ''shell'' service num/etc.      */
```

```
PRC_ERR_SELECT      /* select() failed                          */

PRC_CSBIT_ERR       /* connection has errored out               */

PRC_CONN_NONE       /* needs connection                         */
PRC_CONN1_WAIT      /* waiting for stdio connect()              */
PRC_CONN2_WAIT      /* waiting for stderr connect()             */
PRC_CONN3_WAIT      /* waiting for remshd reply                 */
PRC_READ_WAIT       /* waiting for data                         */
PRC_READ_READY      /* data is ready to read                    */
PRC_CONN_DONE       /* connection closed                        */
PRC_CONN_NO_IPS     /* can't get IP addresses                   */
PRC_CONN_FAILED     /* various causes                           */
PRC_CONN_REFUSED    /* by remote system                         */
PRC_CONN_TIMEOUT    /* during connection attempt                */

typedef struct prc_host *prcp_t;

#define PRC_NULLP   ((prcp_t) 0)
#define PRC_SIZE    (sizeof (struct prc_host))
```

Data Initialization

The program calling **prcmd()** should create an array of **prc_host** structures with one node for each target system. It should then call **prcmd_init()** using that array (*hostp*) and its size (*num_hosts*), a desired initial value (*caller_status*) for each host's **prc_caller_status** field, a *command* to be executed when initiating a remote connection to a host, and a *timeout* value to apply to each host during setup of its connection. This call sets the values of most fields for each host. The calling program can change each host's value for **prc_caller_status**, but normally they are all initialized to the same value.

Finally, initialize the following field in each array element before the first call to **prcmd()**:

prc_hostid

Either a system name (with or without domain suffix) or an Internet address in dot notation. This value differs for each host; it's easier for the calling program to set the values directly than to pass a list into **prcmd_init()**.

This value is a pointer to memory allocated by the calling program. Avoid destroying the underlying data until **prcmd()** establishes a connection to the host. After that, it no longer needs **prc_hostid**.

Like **rcmd()**, **prcmd()** does name lookup on host names to get internet addresses; but unlike **rcmd()**, it does not modify the calling program's **prc_hostid** values.

Common Error Handling

Each time **prcmd()** is called, it scans the host list looking for actions to take based on each host's **prc_conn_close** field and its **prc_conn_status** field, as described below. Errors are generally handled as follows:

Failure — If any system call fails while working on a given host's connection, except as noted below, **prcmd()** sets that host's **prc_conn_status** field to **PRC_CONN_FAILED**, sets its **prc_errno** to the returned **errno** value from the failed call, and closes all file (socket) descriptors, and stream pointers (if any), that are associated with the host.

Refusal — If any failure occurs on the remote (**remshd**) side of a connection, as described below, or if an invalid attempt is made to connect back to a host's standard error port, **prcmd()** sets **prc_conn_status** to **PRC_CONN_REFUSED** and closes the files specified by the host's **prc_fp** and **prc_fp2** fields. If any message is available from the remote **remshd** (up to a limit of **BUFSIZ**-1 characters) and **malloc()** succeeds, **prcmd()** saves the message text in a location pointed to by the **prc_errmsg** field; otherwise that field remains a null pointer. If instead the refusal occurs for "local" reasons and an **errno** value is available, **prcmd()** sets **prc_errno** to that value, otherwise to 0.

Timeout — A connection times out when it is in **PRC_CONN1_WAIT**, **PRC_CONN2_WAIT**, or **PRC_CONN3_WAIT** state (see below), **select()** reports that the host is not ready for I/O, and at least *timeout* seconds have elapsed since the host entered that state. The test is low-precision, to the second, using **time()**. In this case, **prcmd()** sets the **prc_conn_status** field for

the host to **PRC_CONN_TIMEOUT**, and closes the files specified by the host's **prc_fp** and **prc_fp2** fields.

A host specified by name can be listed with multiple IP addresses in the hosts database. In that case, when a connection to one of that host's IP addresses fails, is refused, or times out before reaching the **PRC_READ_WAIT** state, **prcmd()** attempts to open a new connection on the host's next IP address (up to a limit of five IP addresses per host). A failure is not returned for a given host until all of its IP addresses have been tried.

Host/Connection Status

See "**Usage Notes**" for additional information on how calling programs should handle the following connection status conditions.

PRC_CONN_NONE **prcmd()** maps the **prc_hostid** value from a name to one or more IP addresses (if it does not appear to already be an address), maps the IP addresses to binary, gets an Internet socket bound to a reserved port number for communicating with the host, maps the socket descriptor to a stream pointer using **fdopen()**, and performs a non-blocking **connect()** of the socket descriptor to the remote host's "shell" service. The socket's send and receive buffer sizes are the default; the calling program can change the sizes with **setsockopt (fileno (hostp → prc_fp))** at any time, subject to the restrictions documented in *setsockopt*(2).

If **gethostbyname()** (see *gethostbyname*(3N)) or **inet_addr()** (see *inet_addr*(3N)) fails, **prcmd()** sets the **prc_conn_status** field to **PRC_CONN_NO_IPS**.

Information needed for an **rcmd()** connection includes the port number for the "shell" service and the user name of the effective user ID as the local and remote user names. **prcmd()** looks up this information the first time it makes a connection. If **prcmd()** cannot obtain this information it returns **PRC_ERR_RCMD** (and tries again each time it's called).

If **socket()** fails with any of the following local-host errors: **EHOSTDOWN**, **EAFNOSUPPORT**, **ESOCKTNOSUPPORT**, **EPROTONOSUPPORT**, **EPROTOTYPE**, or **EINVAL**, **prcmd()** returns **PRC_ERR_NETWORKING** and the status of the host list is unspecified.

If **socket()** fails with **EMFILE** or **ENFILE**, which means the process or the system is out of file descriptors, or if there are no available reserved ports, the **prc_conn_status** field is left as **PRC_CONN_NONE** for later retries. After checking all host entries, if **prcmd()** has no open files (sockets) for any host, and it didn't close any files during this call, it returns **PRC_ERR_NOFILE** — that is, all file descriptors are in use for this process, but none by **prcmd()**.

If **connect()** succeeds, or if it fails with **EINPROGRESS**, which is normal for a non-blocking connect request, **prcmd()** sets **prc_conn_status** to **PRC_CONN1_WAIT**, **prc_conn_time** to the current system clock in seconds, and **prc_fp** to the stream pointer for this socket. This host is then included in the **select()** list per **PRC_CONN1_WAIT** (see below).

PRC_CONN1_WAIT

prcmd() includes this host in the **select()** list for write-readiness. If the connection is not ready for writing, the **prc_conn_status** field is left unchanged.

When the connection is ready for writing, **prcmd()** returns the socket to "blocking", starts a connection for standard error back from the remote command, sets **prc_fp2** to the file pointer for this connection, writes standard remsh connection information (see *remshd*(1M) and also above) to the remote process with **SIGPIPE** temporarily ignored using **sigvector()**, changes **prc_conn_status** to **PRC_CONN2_WAIT**, and sets **prc_conn_time** to the current system clock in seconds.

If a connection is refused, **prcmd()** gets **ECONNREFUSED** or **EPIPE** on attempt to write remsh connection information, and handles the error as previously described in the section called "**Common Error Handling**".

PRC_CONN2_WAIT

prcmd() includes this host in the **select()** list for read-readiness — for both

prc_fp (in case of error) and **prc_fp2** (in case of successful connection back to the standard error port). When neither connection is ready for reading, the **prc_conn_status** field is left unchanged.

When the standard error connection is ready for reading, **prcmd()** does an **accept()** on it, closes the old **prc_fp2** and revises the value to a new file pointer for the new socket/fd, changes **prc_conn_status** to **PRC_CONN3_WAIT**, and sets **prc_conn_time** to the current system clock in seconds.

If only the stdout connection is ready for reading, this indicates a **remshd** failure. **prcmd()** treats this as a refused connection, as described earlier under "**Common Error Handling**".

Note that a host is "held" in **PRC_CONN1_WAIT** or **PRC_CONN2_WAIT** status if there are no available file descriptors or reserved ports when one is needed.

PRC_CONN3_WAIT

prcmd() includes this host in the **select()** list for read-readiness. If the connection is not ready for reading, the **prc_conn_status** field is left unchanged.

When the connection is ready for reading, **prcmd()** consumes the null byte on stdout from **remshd**, sets **prc_conn_status** to **PRC_READ_WAIT**, and resets **prc_conn_time** to the current system clock in seconds for use by the calling program during the ensuing conversation. If anything other than a null byte arrives, this indicates a **remshd** failure. **prcmd()** treats this as a refused connection, as described previously.

PRC_READ_WAIT **prcmd()** includes this host in the **select()** list for read-readiness. When the connection is not ready for reading, the **prc_conn_status** field is left unchanged. Note, the remote command might be waiting for the calling program to write data to it.

When the connection is ready for reading, **prcmd()** sets **prc_conn_status** to **PRC_READ_READY**.

PRC_READ_READY

prcmd() treats a host in this status the same as **PRC_READ_WAIT**. If the connection is not ready for reading, **prcmd()** sets **prc_conn_status** back to **PRC_READ_WAIT**.

PRC_CONN_DONE **prcmd()** ignores this host's entry.

prcmd() enters this state from any other state when the **prc_conn_close** field (set by the calling program) is non-zero. It closes the files specified by **prc_fp** and **prc_fp2** (if not already null), sets those fields to null, and resets the **prc_conn_close** field to 0.

PRC_CONN_NO_IPS
PRC_CONN_FAILED
PRC_CONN_REFUSED
PRC_CONN_TIMEOUT

With any of these failure status values, **prcmd()** ignores this host's entry, except if **prc_conn_close** is set, the entry's **prc_conn_status** field is set to **PRC_CONN_DONE**.

Changes in Host Status

After each **prcmd()** call, the **prc_prev_status** field for each host's entry is set to the previous value of its **prc_conn_status** field. The calling program can use **(prc_prev_status != prc_conn_status)** to quickly check for a new status for any host. This is useful for once-only logging or display of status values except **PRC_READ_WAIT** and **PRC_READ_READY** — each of these states can be entered many times during a conversation.

Note that a host can leave and then re-enter **PRC_CONN_NONE** status if it has multiple IP addresses, a connection to one IP fails, and there are no files or ports available to try the host's next IP address.

The calling program can freely use the **prc_caller_status** field to record additional information about the status of each connection.

Usage Notes

Once a connection is made, and a host is in one of the **PRC_READ_*** states, control of the "conversation" between the calling program and remote host belongs to the calling program. Conversations are of two types:

1. calling program communicates first
2. remote command communicates first

If the calling program communicates first, it must use the **prc_caller_status** field, or other means, to remember whether or not it has sent initial data to the command; and it must send that data upon the host entering the **PRC_READ_WAIT** state the first time. If the command communicates first, or in any conversation after the calling program sends initial data, the calling program must "track" the conversation to know when to send more data, if any, and when to close the conversation by setting **prc_conn_close**.

Before the calling program can send the next input, or close the connection, it might have to wait through a series of **PRC_READ_WAIT** and **PRC_READ_READY** states (until all bytes of a single response from a command are received).

After each **prcmd()** call, the calling program should check the return value for a serious error. If the return value is **PRC_OK**, the calling program should next scan the *hostp* array and check **prc_prev_status** (if desired) and **prc_conn_status** fields for each host. The calling program only needs to take action for certain status values (marked with "•" below).

PRC_CONN_NONE After the first **prcmd()** call: Indicates the process ran out of socket (file) descriptors, or reserved ports, after making at least one (still open) connection (in this call, or in a previous call). Connection to this host is attempted again on the next call. The calling program can increase the number of available file descriptors (by closing files or calling **rlimit()**), or wait for existing connections to finish their communications.

PRC_CONN1_WAIT
PRC_CONN2_WAIT
PRC_CONN3_WAIT

These are internal states of little interest to the calling program. No action is needed except to call **prcmd()** again later. If **(prc_prev_status != prc_conn_status)**, the calling program might log/display the host's new status.

PRC_READ_WAIT •

Command is running but has not yet produced output. If the calling program communicates first, meaning the command needs some initial input, write it to the file specified by **prc_fp** and remember that this was done, probably by using the **prc_caller_status** field. Otherwise no action is required, except maybe to log/display the connection's status.

PRC_READ_READY •

Command produced output that is ready for reading. After reading the output, if the command needs more input, write it now to the file specified by **prc_fp**. If instead the conversation is done, set the **prc_conn_close** field non-zero.

PRC_CONN_DONE Command is done; connection was closed in response to the calling program setting the **prc_conn_close** field. No action is required.

The calling program can use **(prc_conn_status & PRC_CSBIT_ERR)** to quickly detect any error status. It can use **(prc_prev_status != prc_conn_status)** to detect that any failure condition is new so it is only handled once, or use **prc_caller_status** to record that it was processed the first time it appears in a return from **prcmd()**. In each case the calling program should handle or report the error as desired.

PRC_CONN_NO_IPS •

The **prc_hostid** field appears to be a name, not an address, and **gethostbyname()** fails for it, or **inet_addr()** fails on a numeric **prc_hostid** value.

The remaining conditions only refer to the cause of failure on the last of the host's IP addresses, if it has more than one.

PRC_CONN_FAILED •

Connection could not be initiated for some reason (on any of the host's IP addresses)

due to failure on the local system. **prc_errno** is set to the **errno** value from a failed system call.

PRC_CONN_REFUSED •
Connection refused. Per *errno*(2): "This usually results from trying to connect to a service that is inactive on the foreign host." It can also be due to any other remote (**remshd**) failure, including access denial; and to an improper connection attempt on the standard error port.

The calling program should check the host's **prc_errmsg** field. If it is not a null pointer, message text is available from the remote **remshd** command; and the calling program must **free()** the pointer when done using the value.

PRC_CONN_TIMEOUT •
Connection timed out. The connection has remained in one of the **PRC_CONN1_WAIT**, **PRC_CONN2_WAIT**, or **PRC_CONN3_WAIT** states for more than *timeout* seconds without becoming ready for I/O.

If the calling program performs unbuffered I/O using socket (file) descriptors instead of streams, it can refer to **fileno(prc_fp)** (see *fileno*(3S)).

If the calling program needs to send a signal to a remote command, it can write the signal number to the file specified by **prc_fp2** (see *rcmd*(3N)).

There is no way to tell **prcmd()** to close down all connections, except by setting **prc_conn_close** for all hosts, and calling **prcmd()** once more. This causes **prc_conn_status** to be set to **PRC_CONN_DONE**.

Regarding Timeouts

prcmd() is designed to maximize the number of parallel connections, and to maximize control by the calling program. Therefore **prcmd()** always calls **select()** with a timeout value of zero (immediate polling). It is the responsibility of the calling program to avoid calling **prcmd()** more frequently than is necessary (therefore consuming excessive CPU time). For example, the calling program might call **sleep(1)** (see *sleep*(3C)) between **prcmd()** calls. Note: This is unnecessary if the calling program is itself periodically invoked by timer interrupts, or if it performs other (time-consuming) tasks between **prcmd()** calls.

A timeout will occur if the host's connection remains in one of the wait states, **PRC_CONN1_WAIT**, **PRC_CONN2_WAIT**, or **PRC_CONN3_WAIT**, for more than *timeout* seconds without becoming ready for I/O, and no untried IP addresses for this host remain. When a host's connection reaches **PRC_READ_WAIT** state the first time, **prcmd()** resets the host's **prc_conn_time** field but does not check the host again for timeout. The calling program can do this if desired. The calling program is also permitted to reset (update) the **prc_conn_time** field as the communications proceed (for example, each time it writes data to the remote command).

RETURN VALUE

PRC_OK Call succeeded; check the host list for **prc_prev_status** and **prc_conn_status** field values. If **select()** fails with **EINTR** (due to a signal arriving), **prcmd()** returns control to the calling program with **PRC_OK**. In this case some connections might have been started (entered the **PRC_CONN1_WAIT** state) or stopped (entered the **PRC_CONN_DONE** state), under the calling program's control, but none has advanced due to I/O status; call **prcmd()** again as usual.

PRC_ERR_NETWORKING
A **socket()** call failed due to any of a number of serious networking problems (see earlier list), indicating the local host's networking is not enabled, and **prcmd()** is useless; **errno** is set on return from **prcmd()**.

PRC_ERR_NOFILE A **socket()** call failed with **EMFILE** or **ENFILE**, or there were no more available reserved ports, at a time when **prcmd()** had no open connections (sockets). This means further attempts to open connections are futile until the calling program, or the system, frees some file descriptors, or reserved ports.

PRC_ERR_RCMD Unable to get the "shell" service port number or the username for the effective user ID. It's not useful to call **prcmd()** again unless this condition is corrected.

PRC_ERR_SELECT A **select()** call failed (other than with **EINTR**); **errno** is set on return from **prcmd()**. The data in the host list is valid but no hosts are read-ready or write-ready, even if marked **PRC_READ_READY** from a previous successful call.

DIAGNOSTICS

Unlike **rcmd()**, **prcmd()** does not copy messages from **remshd** to the local standard error when **remshd** fails. It just puts the host in **PRC_CONN_REFUSED** state.

EXAMPLES

The following code fragment illustrates how to allocate and initialize an array of hosts, and call **prcmd()** once for the entire list of hosts:

```
int index;
struct prc_host prc_host [MAXHOSTS];

prcmd_init (prc_host, argc, 0, REMCMD, TIMEOUT);

for (index = 0; index < argc; ++index)
   (prc_host[index].prc_hostid) = argv [index];

if ((rc = prcmd (prc_host, argc)) != PRC_OK)
   ...
```

WARNINGS

prcmd() is unsafe in multi-thread applications.

Like **rcmd()**, **prcmd()** should only be called by a privileged process. Otherwise it puts every host in **PRC_CONN_FAILED** status with **errno** = 13 (**EACCES**) because it cannot **bind()** to any reserved port.

At this time **select()** calls are done on files specified by **prc_fp** only, not **prc_fp2**. **prcmd()** assumes that the remote command will never write to standard error and then block, but will instead terminate or otherwise close stdout. When stdout is read-ready, the calling program can safely do a blocking **read()** on **prc_fp**, and upon EOF or error, check **prc_fp2** with a blocking **read()**.

(Exception: For status **PRC_CONN2_WAIT** both files are selected since either might come ready depending on success or failure, as described earlier. However, this exception is handled internal to **prcmd()** and does not affect the calling program.)

Note that **prcmd_init()** saves a pointer to the value of *command*, not the value itself, so the calling program must preserve the value through all calls of **prcmd()**.

connect() times out after about two minutes, even with non-blocking I/O. If this happens to a host's connection, the result is indistinguishable from **PRC_CONN_REFUSED** since **write()** returns **EPIPE** in either case. This is not a problem if the calling program allows less than two minutes for a connection, that is, sends a *timeout* value less than 120, and calls **prcmd()** frequently enough that the latter catches the timeout itself before **connect()** does.

Be careful when using buffered I/O. Avoid calls to **fgets()**, **fread()**, and similar functions in circumstances where they might block indefinitely if less data is available than requested. Also, be sure to read all available data from a host via the file specified by **prc_fp** before waiting again for a connection to be read-ready. The connection might appear not-ready even though buffered, pending data was already read by the stdio library, and is available. Avoiding these problems might require the use of non-blocking I/O; see *fcntl*(2).

Be careful about writing to a socket that might be closed if the remote command terminates, or if the connection is lost. This can cause the calling process to receive **SIGPIPE**. The calling program must handle this signal if appropriate.

When handling a host whose connection failed with **PRC_CONN_REFUSED**, remember to **free()** the host's **prc_errmsg** pointer if it is non-null. This can be skipped if regaining malloc'd memory is not required.

Performance Issues

It is faster to call **prcmd()** with host IDs specified as internet addresses rather than names, because **prcmd()** will not need to look up each host in the hosts database. However, when doing so with hosts

that have multiple internet addresses, only the specified internet address is tried.

There is a limit of about 512 reserved ports on each system, and a soft limit for the number of available file descriptors for each process. Each specified host requires two reserved ports and two file descriptors for the connection to check its status. **prcmd()** defers (serializes) additional connections once all ports or file descriptors are in use, so additional hosts are not ignored, but the time required for all connections to complete increases accordingly. To improve performance, it might be necessary to increase the limit on open files if there are a lot of hosts; see *setrlimit*(2).

AUTHOR

prcmd() was developed by Hewlett-Packard.

SEE ALSO

remshd(1M), accept(2), bind(2), connect(2), errno(2), fcntl(2), read(2), select(2), setsockopt(2), sigvector(2), socket(2), time(2), fdopen(3S), fileno(3S), gethostbyname(3N), inet_addr(3N), malloc(3C), rcmd(3N), sleep(3C).

p

NAME

printf(), fprintf(), sprintf(), snprintf() - print formatted output

SYNOPSIS

```
#include <stdio.h>
int printf(const char *format, /* [arg,] */ ...);
int fprintf(FILE *stream, const char *format, /* [arg,] */ ...);
int sprintf(char *s, const char *format, /* [arg,] */ ...);
int snprintf(char *s, size_t maxsize, const char *format,
     /* [arg,] */ ...);
```

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.

snprintf() behaves like **sprintf()**, except that it limits the number of characters written to the destination buffer to *maxsize*, including the terminating null character.

Each function converts, formats, and prints its *arg*s under control of the *format*. *format* is a character string containing two types of objects: plain characters that are copied to the output stream, and conversion specifications, each of which results in fetching 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, excess *arg*s are ignored.

Each conversion specification is introduced by the character **%** or **%***n***$**, where *n* is a decimal integer in the range 1 through **{NL_ARGMAX}** (**NL_ARGMAX** is defined in **<limits.h>**). The **%***n***$** construction indicates that this conversion should be applied to the *n*th argument, rather than to the next unused one.

An argument can be referenced by a **%***n***$** specification more than once. The two forms of introducing a conversion specification, **%** and **%***n***$**, cannot be mixed within a single *format* string. When numbered argument specifications are used, specifying the Nth argument requires that all the leading arguments, from the first to the (N-1)th, are specified in the format string. Improper use of **%***n***$** in a format string results in a negative return value.

After the **%** or **%***n***$**, the following appear in sequence:

1. Zero or more *flags*, which modify the meaning of the conversion specification.
2. An optional string of decimal digits to specify a minimum *field width* in bytes. If the converted value has fewer characters than the field width, it is 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 is preceded by a zero, the string is right adjusted with zero-padding on the left (see the leading-zero flag, **0**, described below).
3. A *precision* that gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, or **X** conversions, the number of digits to appear after the radix character for the **e** and **f** conversions, the maximum number of significant digits for the **g** conversion, or the maximum number of bytes to be printed from a string in the **s** conversion. The *precision* takes the form of a period followed by a decimal digit string; a null digit string is treated as zero.
4. An optional **l** (the letter "ell"), specifying that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion character applies to a long integer *arg*; an optional **l** specifying that a following **n** conversion character applies to a pointer to a long integer *arg*; an optional **h**, specifying that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion character applies to a short integer *arg*; an optional **ll**, specifying that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion character applies to a long long integer arg; an optional **h** specifying that a following **n** conversion character applies to a pointer to a short integer *arg*; an optional **L** specifying that a following **e**, **E**, **f**, **g**, or **G** conversion character applies to a long double *arg*. An **l**, **h**, **ll** or **L** before any other conversion character is ignored.
5. A conversion character that indicates the type of conversion to be applied.

A field width or precision can 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, or both must appear in that order before the *arg*, if any, to be converted. A negative field width is taken as a **-** flag followed by a positive field width. A negative precision is taken as if the precision were omitted. Format strings containing **%***n***$** conversion specifications can also indicate a field width or precision by the sequence ******n***$**. The *n* indicates the position of an integer *arg* . With the ******n***$** sequence, the *arg*s specifying field width or precision can appear before or after the *arg* to be converted.

The flag characters and their meanings are:

' The integer portion of the result of a decimal conversion (%i, %d, %u, %f, %g, or %G) will be formatted with thousands' grouping characters. For other conversions the behavior is undefined. The nonmonetary grouping character is used.

- The resulting conversion is left-justified within the field.

+ The resulting signed conversion always begins with a sign (**+** or **-**).

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

This flag specifies that the value is converted to an "alternate form". For **c**, **d**, **i**, **s**, **n**, 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 nonzero result is prefixed by **0x** or **0X**. For a **p** conversion, a nonzero result is prefixed by **0x**. For **e**, **E**, **f**, **g**, and **G** conversions, the result always contains a radix character, even if no digits follow the radix. (Normally, a radix character appears in the resulting conversions only if followed by a digit). For **g** and **G** conversions, trailing zeros are *not* removed from the result (which they normally are).

0 Leading zeros (following any indication of sign or base) are used to pad to the field width for all conversion characters. No space padding is performed. If both the **0** and **-** appear, the **0** flag is ignored. For **d**, **i**, **o**, **u**, **p**, **x**, and **X**, conversions, if a precision is specified, the **0** flag is ignored.

The conversion characters and their meanings are:

d,**i**,**o**,**u**,**x**,**X** The integer *arg* is converted to signed decimal (**d** and **i** are identical), unsigned octal (**o**), decimal (**u**), 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 is expanded with leading zeros. (For compatibility with older versions, padding with leading zeros can alternatively be specified by inserting a zero in front of 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 double *arg* is converted to decimal notation in the style [**-**]**ddd***r***ddd**, where *r* is the radix character. The number of digits after the radix character is equal to the precision specification. If the precision is missing, six digits are output. If the precision is explicitly zero, no radix character appears.

e,**E** The double *arg* is converted in the style [**-**]**d***r***ddde±ddd**, where **r** is the radix character. There is one digit before the radix character 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 radix character appears. The **E** format code produces a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits.

g,**G** The 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** is used only if the exponent resulting from the conversion is less than **-4** or greater than or equal to the precision. Trailing zeros are removed from the fractional part of the result; a radix character appears only if it is followed by a digit.

c The integer *arg* is converted to an unsigned char, and the resulting character is printed.

c The **wchar_t** *arg* is converted to an array of bytes representing the single wide character according to the setting of **LC_CTYPE**. Resulting bytes are printed. If the precision is given, it is ignored. If the field width would otherwise cause the wide character to be split, the wide character is printed and the field width is adjusted upward.

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 bytes 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* yields undefined results.

S The *arg* is taken to be a pointer to a wide character string (**wchar_t ***). Wide characters from the string are converted to an array of bytes representing the string of wide characters according to the setting of **LC_CTYPE**. Resulting bytes are printed until a null wide character (**(wchar_t)0**) is encountered or the number of bytes indicated by the precision is reached. If the precision is missing, it is taken to be infinite, so all wide characters up to the first null wide character are printed. If the field width would otherwise cause the last multibyte character to be split, the last wide character is not printed. A NULL value for *arg* yields undefined results.

p The value of a pointer to void *arg* is printed as a sequence of unsigned hexadecimal numbers. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is a null string.

n A pointer to an integer *arg* is expected. This pointer is used to store the number of bytes printed on the output stream so far by this call to the function. No argument is converted.

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

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

The **e**, **f**, and **g** conversions will print **inf** for infinity and **nan** for both quiet and signaling **NaN** values.

The **E** and **G** conversions will print **INF** for infinity and **NAN** for both quiet and signaling **NaN** values. There will be a new **F** conversion specifier that will be identical to the **f** conversion specifier except that it will print infinity and NaN as respectively, **INF** and **NAN**.

Characters generated by **printf()** and **fprintf()** are printed as if **putc()** had been called (see *putc*(3S)).

p

APPLICATION USAGE

printf(), **fprintf()**, **sprintf()** and **snprintf()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing **printf()** or **fprintf()**.

EXTERNAL INFLUENCES

Locale

The **LC_CTYPE** category affects the following features:

- Plain characters within format strings are interpreted as single byte and/or multibyte characters.
- Field width is given in terms of bytes. As characters are placed on the output stream, they are interpreted as single byte or multibyte characters and the field width is decremented by the length of the character.
- Precision is given in terms of bytes. As characters are placed on the output stream, they are interpreted as single byte or multibyte characters and the precision is decremented by the length of the character.
- The return value is given in terms of bytes. As characters are placed on the output stream, they are interpreted as single byte or multibyte characters and the byte count that makes up the return value is incremented by the length of the character.

The **LC_NUMERIC** category determines the radix character used to print floating-point numbers, and the thousands' grouping characters if the grouping flag **'** is on.

International Code Set Support

Single byte character code sets are supported. Multibyte character code sets are also supported as described in the **LC_CTYPE** category above.

RETURN VALUE

Each function returns the number of bytes transmitted (excluding the **\0** in the case of **sprintf()** or a negative value if an output error was encountered. Improper use of **%*n*$** in a format string results in a negative return value.

ERRORS

printf(), and **fprintf()** fail if either the *stream* is unbuffered or *stream*'s buffer needed to be flushed causing an underlying **write()** call to be invoked (see *write*(2)), and:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.

[EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for writing.

[EFBIG] An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size (see *ulimit*(2)).

[EINTR] A signal was caught during the **write()** system call.

[EIO] The process is in a background process group and is attempting to write to its controlling terminal, **TOSTOP** is set, the process is neither ignoring nor blocking the **SIGTTOU** signal, and the process group of the process is orphaned.

[ENOSPC] There was no free space remaining on the device containing the file.

[EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal is also sent to the process.

Additional **errno** values can be set by the underlying **write()** function (see *write*(2)).

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));
```

To create a language independent date-and-time printing routine write:

```
printf(format,weekday,month,day,hour,min,2,2);
```

For American usage, *format* would point to the string:

```
"%1$s, %2$s %3$d, %4$*6$.*7$d:%5$*6$.*7$d"
```

and result in the output:

```
"Sunday, July 3, 10:02"
```

For German usage, the string:

```
"%1$s, %3$s %2$d, %4$*6$.*7$d:%5$*6$.*7$d"
```

results in the output:

```
Sonntag, 3 Juli 10:02
```

WARNINGS

Notice that with the **c** conversion character, an int *arg* is converted to an unsigned char. Hence, whole multibyte characters cannot be printed using a single **c** conversion character.

A precision with the **s** conversion character might result in the truncation of a multibyte character.

Use of a conversion character which doesn't match the type of the corresponding argument passed in will result in invalid data being returned. For example, the modifying **ll**, specifying that the following **d**, **i**, **o**, **u**, **x**, or **X** conversion character applies to a long long integer arg, is necessary for proper results when the argument is a long long integer.

AUTHOR

printf(), **fprintf()**, and **sprintf()** were developed by AT&T and HP.

SEE ALSO

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

STANDARDS CONFORMANCE

printf(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

fprintf(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

sprintf(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

(Pthread Library)

NAME

pthread - Introduction To POSIX.1c Threads

DESCRIPTION

The POSIX.1c library developed by HP enables the creation of processes that can exploit application and multi-processor platform parallelism. The pthread library *libpthread* consists of over 90 standardized interfaces for developing concurrent applications and synchronizing their actions within processes or between them. This manual page presents an overview of *libpthread* including terminology and how to compile and link programs which use threads.

COMPILATION SUMMARY

A multi-threaded application must define the appropriate POSIX revision level (199506) at compile time and link against the pthread library via -lpthread. For example:

```
cc -D_POSIX_C_SOURCE=199506L -o myapp myapp.c -lpthread
```

All program sources must also include the header file <**pthread.h**>.

Note: Some documentation will recommend the use of **-D_REENTRANT** for compilation. While this also functions properly, it is considered an obsolescent form.

THREAD OVERVIEW

A *thread* is an independent flow of control within a process, composed of a context (which includes a register set and a program counter) and a sequence of instructions to execute.

All processes consist of at least one thread. Multi-threaded processes contain several threads. All threads share the common address space allocated for the process. A program using the POSIX pthread APIs creates and manipulates what are called *user threads*. A *kernel thread* is a kernel-schedulable entity which may support one or more user threads. Currently, the HP-UX threads implementation supports only a one-to-one mapping between user and kernel threads.

Each thread is assigned a unique identifier of type *pthread_t* upon creation. The thread id is a process-private value and implementation-dependent. It is considered to be an opaque handle for the thread. Its value should not be used by the application.

NOTES ON INTERFACES

The HP-UX system provides some non-standard extensions to the pthread API. These will always have a distinguishing suffix of **_np** or **_NP** (non-portable).

The programmer should always consult the manpages for the functions being used. Some standard-specified functions are not available or may have no effect in some implementations.

THREAD CREATION/DESTRUCTION

A program creates a thread using the **pthread_create()** function. When the thread has completed its work, it may optionally call the **pthread_exit()** function, or simply return from its initial function. A thread can detect the completion of another by using the **pthread_join()** function.

pthread_create() Creates a thread and assigns a unique identifier, *pthread_t*. The caller provides a function which will be executed by the thread. Optionally, the call may explicitly specify some attributes for the thread (see **PTHREAD ATTRIBUTES** below).

pthread_exit() Called by a thread when it completes. This function does not return.

pthread_join() This is analogous to **wait()**, but for pthreads. Any thread may join any other thread in the process, there is no parent/child relationship. It returns when a specified thread terminates, and the thread resources have been reaped.

pthread_detach() Makes it unnecessary to "join" the thread. Thread resources are reaped by the system at the time the thread terminates.

PTHREAD ATTRIBUTES

A set of thread attributes may be provided to **pthread_create()**. Any changes from default values must be made to the attribute set before the call to **pthread_create()** is made. Subsequent changes to the attribute set do not affect the created thread. However, the attribute set may be used in multiple **pthread_create()** calls.

Note that only the "detachstate", "schedparam", "schedpolicy", and "processor" attributes of a thread may be effected subsequent to thread creation. However, this is done via the **pthread_detach()**, **pthread_setschedparam()**, and **pthread_processor_bind_np()** functions, respectively.

pthread_attr_init() Initializes an attribute set for use in the **pthread_create()** call.

pthread_attr_destroy() Destroys the content of an attribute set.

pthread_attr_getdetachstate(),
pthread_attr_getguardsize(),
pthread_attr_getinheritsched(),
pthread_attr_getprocessor_np(),
pthread_attr_getschedparam(),
pthread_attr_getschedpolicy(),
pthread_attr_getscope(),
pthread_attr_getstackaddr(),
pthread_attr_getstacksize(),
pthread_attr_setdetachstate(),
pthread_attr_setguardsize(),
pthread_attr_setinheritsched(),
pthread_attr_setprocessor_np(),
pthread_attr_setschedparam(),
pthread_attr_setschedpolicy(),
pthread_attr_setscope(),
pthread_attr_setstackaddr(),
pthread_attr_setstacksize()

These **pthread_attr_get/set<attribute>()** functions get/set the associated attribute in the attribute set. See the manpages for these functions for descriptions of the attributes.

pthread_default_stacksize_np()
This is used to set the default stacksize for threads created in subsequent attribute set initializations (calls to **pthread_attr_init()**) or in **pthread_create()** where no attributes are supplied.

CANCELLATION

Certain applications may desire to terminate a particular thread without causing the entire process to exit. A thread may be canceled by another thread in the same process while the cancellation target thread executes a system call or particular library routine.

When a thread issues a cancel request against another thread, the target thread can check to see if a request is pending against it via the **pthread_testcancel()** interface. When called with a request pending, the target thread terminates after executing any cleanup handlers which may have been installed. Cleanup handlers may be used to delete any dynamic storage allocated by the canceled thread, to unlock a mutex, or other operations.

Typically, the cancellation type for a thread is *deferred*. That is, cancellation requests are held pending until the thread reaches a *cancellation* point which is simply one of a list of library functions and system calls (see lists below).

The thread may set its cancellation type to *asynchronous*. In this case cancellation requests are acted upon at any time. This can be used effectively in compute-bound threads which do not call any functions that are cancellation points.

pthread_cancel() Cancel execution of a given thread.

pthread_testcancel() Called by a thread to process pending cancel requests.

pthread_setcancelstate(),
pthread_setcanceltype()

Set the characteristics of cancellation for the thread. Cancellation may be enabled or disabled, or it may be synchronous or deferred.

pthread_cleanup_pop(),
pthread_cleanup_push()

Register or remove cancellation cleanup handlers.

Cancellation points in the *pthread* library:

pthread_testcancel()
pthread_cond_timedwait()
pthread_cond_wait()
pthread_join()

System functions which are always cancellation points:

accept()	aio_suspend()	close()	connect()
creat()	dup2()	*fcntl()	fsync()
getdirentries()	getmsg()	getpmsg()	ioctl()
lockf()	lockf64()	lseek()	lseek64()
mq_receive()	mq_send()	msgrcv()	msgsnd()
msync()	nanosleep()	open()	pause()
poll()	putmsg()	putpmsg()	read()
readv()	recv()	recvfrom()	recvmsg()
rename()	select()	semop()	send()
sendmsg()	sendto()	sigsuspend()	sigtimedwait()
sigwait()	sigwaitinfo()	socket()	system()
wait()	wait3()	waitid()	waitpid()
write()	writev()		

* **`fcntl()`** is a cancellation point only with the **F_SETLK** command.

For the following *libc* functions, whether the thread is cancelled depends upon what action is performed while executing the function. If the thread blocks while inside the function, a cancellation point is created (i.e., the thread may be cancelled). Note: Other libraries may have cancellation points. Check the associated documentation for details.

blclose()	blget()	blopen()
blread()	blset()	catclose()
catgets()	catopen()	closedir()
closelog()	creat64()	ctermid()
cuserid()	dbm_close()	dbm_delete()
dbm_fetch()	dbm_firstkey()	dbm_nextkey()
dbm_open()	dbm_store()	dbmclose()
endexportent()	endfsent()	endgrent()
endhostent()	endnetent()	endnetgrent()
endprotoent()	endpwent()	endservent()
endutxent()	fclose()	fflush()
fgetc()	fgetgrent()	fgetpos()
fgetpos64()	fgetpwent()	fgets()
fgetwc()	fgetws()	fmtmsg()
fopen()	fopen64()	fprintf()
fputc()	fputs()	fputwc()
fputws()	fread()	freopen()
freopen64()	fsetpos()	fsetpos64()
ftell()	ftello()	ftello64()
ftw()	ftw64()	fwrite()
getc()	getc_unlocked()	getchar()
getchar_unlocked()	getcwd()	getdate()
getgrent()	getgrgid()	getgrgid_r()
getgrnam()	getgrnam_r()	gethostbyaddr()
gethostbyname()	gethostent()	getlogin()
getlogin_r()	getopt()	getpass()
getpw()	getpwent()	getpwnam()
getpwnam_r()	getpwuid()	getpwuid_r()
gets()	getservbyname()	getservbyport()
getservent()	gettxt()	getusershell()
getutent()	getutid()	getutline()
getutxent()	getutxid()	getutxline()
getw()	getwc()	getwchr()
getwd()	glob()	grantpt()
iconv_close()	iconv_open()	initgroups()
lckpwdf()	mkstemp()	msem_lock()

nftw()	nftw2()	nftw64()
nlist()	open64()	opendir()
openlog()	pclose()	perror()
pfmt()	popen()	prealloc()
prealloc64()	printf()	putc()
putc_unlocked()	putchar()	putchar_unlocked()
putpwent()	puts()	pututline()
pututxline()	putw()	putwc()
putwchar()	putws()	readdir()
readdir_r()	realpath()	remove()
rewind()	rewinddir()	scandir()
scanf()	seekdir()	setgrent()
sethostent()	setnetent()	setnetgrent()
setprotoent()	setpwent()	setservent()
setusershell()	setutent()	setutxent()
sigpause()	sleep()	strerror()
syslog()	tcdrain()	tell()
tmpfile()	tmpfile64()	tmpnam()
ttyname()	ttyname_r()	ttyslot()
ulckpwdf()	ungetc()	ungetwc()
usleep()	vfprintf()	vfscanf()
vprintf()	vscanf()	vsprintf()
vsscanf()	wordexp()	wordfree()

NOTE1: The above functions may not be fully supported or may be considered obsolete. Consult individual manpages for more info.

NOTE2: The list of cancellation points will vary from release to release. In general, if a function can return with an **EINTR** error, chances are that it is a cancellation point.

SCHEDULING

Threads may individually control their scheduling policy and priorities. Threads may also suspend their own execution, or that of other threads. Finally, threads are given some control over allocation of processor resources.

pthread_suspend() This function is used to temporarily stop the execution of a thread.

pthread_continue(),
pthread_resume_np()

These functions cause a previously suspended thread to continue execution.

pthread_num_processor_np(),
pthread_processor_bind_np(),
pthread_processor_id_np()

These functions are used to interrogate processor configuration and to bind a thread to a specific processor.

pthread_getconcurrency(),
pthread_setconcurrency()

These functions are used to control the actual concurrency for unbound threads.

pthread_getschedparam(),
pthread_setschedparam()

These functions are used to manipulate the scheduling policy and priority for a thread.

sched_get_priority_max(),
sched_get_priority_min()

These functions are used to interrogate the priority range for a given scheduling policy.

sched_yield This function is used by a thread to yield the processor to other threads of equal or greater priority.

COMMUNICATION & SYNCHRONIZATION

Multi-threaded applications concurrently execute instructions. Access to process-wide (or interprocess) shared resources (memory, file descriptors, etc.) requires mechanisms for coordination or synchronization among threads. The *libpthread* library offers synchronization primitives necessary to create a deterministic application. A multi-threaded application ensures determinism by forcing asynchronous thread contexts to synchronize, or serialize, access to data structures and resources managed and manipulated during run-time. These are mutual-exclusion (mutex) locks, condition variables, and read-write locks. The HP-UX operating system also provides POSIX semaphores (see next section).

Mutexes furnish the means to exclusively guard data structures from concurrent modification. Their protocol precludes more than one thread which has locked the mutex from changing the contents of the protected structure until the locker performs an analogous mutex unlock. A mutex can be initialized in two ways: by a call to **pthread_mutex_init()**; or by assignment of PTHREAD_MUTEX_INITIALIZER.

Condition Variables are used by a thread to wait for the occurrence of some event. A thread detecting or causing such an event can *signal* or *broadcast* that occurrence to the waiting thread or threads.

Read-Write locks permit concurrent read access by multiple threads to structures guarded by a read-write lock, but write access by only a single thread.

pthread_mutex_init(),
pthread_mutex_destroy()

Initialize/destroy contents of a mutex lock.

pthread_mutex_lock(),
pthread_mutex_trylock(),
pthread_mutex_unlock()

Lock/unlock a mutex.

pthread_mutex_getprioceiling(),
pthread_mutex_setprioceiling()

Manipulate mutex locking priorities.

pthread_mutexattr_init(),
pthread_mutexattr_destroy(),
pthread_mutexattr_getprioceiling(),
pthread_mutexattr_getprotocol(),
pthread_mutexattr_getpshared(),
pthread_mutexattr_gettype(),
pthread_mutexattr_getspin_np(),
pthread_mutexattr_setprioceiling(),
pthread_mutexattr_setprotocol(),
pthread_mutexattr_setpshared(),
pthread_mutexattr_settype(),
pthread_mutexattr_setspin_np()

Manage mutex attributes used for **pthread_mutex_init()**. Only the "prioceiling" attribute can be changed for an exiting mutex.

pthread_mutex_getyieldfreq_np(),
pthread_mutex_setyieldfreq_np()

These functions, together with the *spin* attributes, are used to tune mutex performance to the specific application.

pthread_cond_init(),
pthread_cond_destroy()

Initialize/destroy contents of a read-write lock.

pthread_cond_signal(),
pthread_cond_broadcast(),
pthread_cond_timedwait(),
pthread_cond_wait()

Wait upon or signal occurrence of a condition variable.

pthread_condattr_init(),
pthread_condattr_destroy(),
pthread_condattr_getpshared(),
pthread_condattr_setpshared()

Manage condition variable attributes used for **pthread_cond_init()**.

pthread_rwlock_init(),
pthread_rwlock_destroy()

Initialize/destroy contents of a read-write lock.

pthread_rwlock_rdlock(),
pthread_rwlock_tryrdlock(),
pthread_rwlock_wrlock(),
pthread_rwlock_trywrlock(),
pthread_rwlock_unlock()

Lock/unlock a read-write lock.

pthread_rwlockattr_init(),
pthread_rwlockattr_destroy(),
pthread_rwlockattr_getpshared(),
pthread_rwlockattr_setpshared()

Manage read-write lock attributes used for **pthread_rwlock_init()**.

POSIX 1.b SEMAPHORES

The semaphore functions specified in the POSIX 1.b standard can also be used for synchronization in a multithreaded application.

sem_init(),
sem_destroy()

Initialize/destroy contents of a semaphore.

sem_post(),
sem_wait(),
sem_trywait()

Increment/decrement semaphore value (possibly blocking).

p

SIGNALS

In a multithreaded process, all threads share signal actions. That is, a signal handler established by one thread is used in all threads. However, each thread has a separate signal mask, by which it can selectively block signals.

Signals can be sent to other threads within the same process, or to other processes. When a signal is sent to the process, exactly one thread which does not have that signal blocked will handle the signal. When sent to a thread within the same process, that thread will handle the signal, perhaps later if the signal is blocked. Signals whose action is to terminate, stop, or continue will terminate, stop, or continue the entire process, respectively, even if directed at a particular thread.

pthread_kill() Sends a signal to the given thread.

pthread_sigmask() Blocks selected signals for the thread.

sigwait(),
sigwaitinfo(),
sigtimedwait()

These functions synchronously wait for given signals.

THREAD-SPECIFIC DATA

Thread-specific data (TSD) is global data that is private or specific to a thread. Each thread has a different value for the same thread-specific data variable. The global *errno* is a perfect example of thread-specific global data.

Each thread-specific data item is associated with a key. The key is shared by all threads. However, when a thread references the key, it references its own private copy of the data.

pthread_key_create(),
pthread_key_destroy()

These functions manage the thread-specific data keys.

pthread_getspecific(),
pthread_setspecific()

These functions retrieve and assign the data value associated with a key.

The HP-UX compiler supports a *thread local storage* (TLS) storage class. (This is not a POSIX standard feature.) TLS is identical to TSD, except functions are not required to create/set/get values. TLS variables are accessed just like normal global variables. TLS variables can be declared using the following syntax:

```
__thread      int      zyx;
```

The keyword **__thread** tells the compiler that **zyx** is a TLS variable. Now each thread can set or get TLS with statements such as:

```
zyx = 21;
```

Each thread will have a different value associated with **zyx**.

TLS variables cannot be statically initialized, all are initially zero. Dynamically loaded libraries (via **shl_load()**) cannot declare (but may use) TLS variables.

TLS does have a cost in thread creation/termination operations, as TLS space for each thread must be allocated and zeroed, regardless of whether it ever will use the variables. If few threads actually use a large TLS area, it may be wise to instead use the POSIX TSD (above).

REENTRANT LIBC & STDIO

Because they return pointers to library-internal static data, a number of *libc* functions cannot be used in multi-threaded programs. This is because calling these functions in a thread will overwrite the results of previous calls in other threads. Alternate functions, having the suffix **_r** (for reentrant), are provided within *libc* for threaded programming.

Also, some primitives for synchronization of standard I/O operations are provided.

asctime_r(),
ctime_r(),
getgrgid_t(),
getgrnam_r(),
getlogin_r(),
getpwnam_r(),
getpwuid_r(),
gmtime_t(),
localtime_r(),
rand_r(),
readdir_r(),
strtok_r(),
ttyname_r()

Provide reentrant versions of previously existing *libc* functions.

flockfile(),
ftrylockfile(),
funlock()

Provide explicit synchronization for standard I/O streams.

MISCELLANEOUS FUNCTIONS

The section summarizes some miscellaneous pthread-related functions not covered in the preceding sections.

pthread_atfork()	Establish special functions to be called just prior to and just subsequent to a **fork()** operation.
pthread_equal()	Tests whether two pthread_t values represent the same pthread.
pthread_once()	Executes given function just once in a process, regardless of how many threads make the same call. (Useful for one-time data initialization.)

pthread_self() Returns identifier (pthread_t) of calling thread.

THREAD DEBUGGING

Debugging of multithreaded programs is supported in the standard HP-UX debugger, **dde**. When any thread is to be stopped due to a debugger event, the debugger will stop all threads. The register state, stack, and data for any thread can be interrogated and manipulated.

See the *dde*(1) manpage and built-in graphical help system for more information.

TRACING FACILITIES

HP-UX provides a tracing facility for pthread operations. To use it, you must link your application using the tracing version of the library:

```
cc -D_POSIX_C_SOURCE=199506L -o myapp myapp.c -lpthread_tr
```

When the application is executed, it produces a per-thread file of pthread events. This is used as input to the **ttv** thread trace visualizer facility available in the **HP/PAK performance application kit**.

There are environment variables defined to control trace data files:

THR_TRACE_DIR
Where to place the trace data files. If this is not defined, the files go to the current working directory.

THR_TRACE_SYNC
By default, trace records are buffered and only written to the file when the buffer is full. If this variable is set to any non-NULL value, data is immediately written to the trace file.

THR_TRACE_EVENTS
By default, all pthread events are traced. If this variable is defined, only the categories defined will be traced. Each category is separated by a ':'. The possible trace categories are:

```
thread:cond:mutex:rwlock
```

For example, to only trace thread and mutex operations set the THR_TRACE_EVENTS variable to:

```
thread:mutex
```

Details of the trace file record format can be found in **/usr/include/sys/trace_thread.h**.

See the *ttv*(1) manpage and built-in graphical help system for more information on the use of the trace information.

PERFORMANCE CONSIDERATIONS

Often, an application is designed to be multithreaded to improve performance over its single-threaded counterparts. However, the multithreaded approach requires some attention to issues not always of concern in the single-threaded case. These are issues traditionally associated with the programming of multiprocessor systems.

The design must employ a *lock granularity* appropriate to the data structures and access patterns. *Coarse-grained* locks, which protect relatively large amounts of data, can lead to undesired lock *contention*, reducing the potential parallelism of the application. On the other hand, employing very *fine-grained* locks, which protect very small amounts of data, can consume processor cycles with too much locking activity.

The use of *thread-specific data* (TSD) or *thread-local* storage (TLS) must be traded off, as described above (see **THREAD-SPECIFIC DATA**).

Mutex *spin* and *yield frequency* attributes can be used to tune mutex behavior to the application. See *pthread_mutexattr_setspin_np*(3T) and *pthread_mutex_setyieldfreq_np*(3T) for more information.

The *default stacksize* attribute can be set to improve system thread caching behavior. See *pthread_default_stacksize_np*(3T) for more information.

Because multiple threads are actually running simultaneously, they can be accessing the same data from multiple processors. The hardware processors coordinate their caching of data such that no processor is using *stale data*. When one processor accesses the data (especially for write operations), the other processors must flush the stale data from their caches. If multiple processors repeatedly read/write the same data, this can lead to *cache-thrashing* which slows execution of the instruction stream. This can also occur when threads access separate data items which just happen to reside in the same hardware-cachable unit (called a *cache line*). This latter situation is called *false-sharing* which can be avoided by spacing data such that popular items are not stored close together.

GLOSSARY

The following definitions were extracted from the text *ThreadTime* by Scott J. Norton and Mark D. DiPasquale, Prentice-Hall, ISBN 0-13-190067-6, 1996.

Application Programming Interface (API)
An interface is the conduit that provides access to an entity or communication between entities. In the programming world, an interface describes how access (or communication) with a function should take place. Specifically, the number of parameters, their names and purpose describe how to access a function. An API is the facility that provides access to a function.

Async-Cancel Safe
A function that may be called by a thread with the cancelability state set to **PTHREAD_CANCEL_ENABLE** and the cancelability type set to **PTHREAD_CANCEL_ASYNCHRONOUS**. If a thread is canceled in one of these functions, no state is left in the function. These functions generally do not acquire resources to perform the function's task.

Async-Signal Safe
An async-signal safe function is a function that may be called by a signal handler. Only a restricted set of functions may safely be called by a signal handler. These functions are listed in section 3.3.1.3 of the POSIX.1c standard.

Asynchronous Signal
An asynchronous signal is a signal that has been generated due to an external event. Signals sent via **kill()** and signals generated due to timer expiration or asynchronous I/O completion are all examples of asynchronously generated signals. Asynchronous signals are delivered to the process. All signals can be generated asynchronously.

Atfork Handler
Application-provided and registered functions that are called before and after a **fork()** operation. These functions generally acquire all mutex locks before the **fork()** and release these mutex locks in both the parent and child processes after the **fork()**.

Atomic Operation
An operation or sequence of events that is guaranteed to complete as if it were one instruction.

Barrier
A synchronization primitive that causes a certain number of threads to wait or rendezvous at specified points in an application. Barriers are used when a application needs to ensure that all threads have completed some operation before proceeding onto the next task.

Bound Thread
A user thread that is directly bound to a kernel-scheduled entity. These threads contain a system scheduling scope and are scheduled directly by the kernel.

Cache Thrashing
Cache thrashing is a situation in which a thread executes on different processors, causing cached data to be moved to and from the different processor caches. Cache thrashing can cause severe performance degradation.

Cancellation Cleanup Handler
An application-provided and registered function that is called when a thread is canceled. These functions generally perform thread cleanup actions during thread cancellation. These handlers are similar to signal handlers.

Condition Variable
A condition variable is a synchronization primitive used to allow a thread to wait for an event. Condition variables are often used in producer-consumer problems where a producer must provide something to one or more consumers.

Context Switch
The act of removing the currently running thread from the processor and running another thread. A context switch saves the register state of the currently running thread and restores the register state of the thread chosen to execute next.

Critical Section
A section of code that must complete atomically and uninterrupted. A critical section of code is generally one in which some global resource (variables, data structures, linked lists, etc.) is modified. The operation being performed must complete atomically so that other threads do not see the critical section in an inconsistent state.

Deadlock
A deadlock occurs when one or more threads can no longer execute. For example, thread *A* holds lock 1 and is blocked on lock 2. Meanwhile, thread *B* holds lock 2 and is blocked on lock 1. Threads *A* and *B* are permanently deadlocked. Deadlocks can occur with any number of resource holding threads. An *interactive deadlock* involves two or more threads. A *recursive* (or *self) deadlock* involves only one thread.

Detached Thread
A thread whose resources are automatically released by the system when the thread terminates. A detached thread cannot be joined by another thread. Consequently, detached threads cannot return an exit status.

Joinable Thread
A thread whose termination can be waited for by another thread. Joinable threads can return an exit status to a joining thread. Joinable threads maintain some state after termination until they are joined by another thread.

Kernel Mode
A mode of operation where all operations are allowed. While a thread is executing a system call it is executing in kernel mode.

Kernel Space
The kernel program exists in this space. Kernel code is executed in this space at the highest privilege level. In general, there are two privilege levels: one for user code (user mode) and the other for kernel code (kernel mode).

Kernel Stack
When a thread makes a system call, it executes in kernel mode. While in kernel mode, it does not use the stack allocated for use by the application. Instead, a separate kernel stack is used while in the system call. Each kernel-scheduled entity, whether a process, kernel thread or lightweight process, contains a kernel stack. See *Stack* for a generic description of a stack.

Kernel Thread
Kernel threads are created by the thread functions in the threads library. Kernel threads are *kernel-scheduled entities* that are visible to the operating system kernel. A kernel thread typically supports one or more user threads. Kernel threads execute kernel code or system calls on behalf of user threads. Some systems may call the equivalent of a kernel thread a *lightweight process.* See *Thread* for a generic description of a thread.

Lightweight Process
A kernel-scheduled entity. Some systems may call the equivalent of a lightweight process a kernel thread. Each process contains one or more lightweight process. How many lightweight processes a process contains depends on whether and how the process is multithreaded. See *Thread* for a generic description of a thread.

Multiprocessor
A system with two or more processors (CPUs). Multiprocessors allow multithreaded applications to obtain true parallelism.

Multithreading
A programming model that allows an application to have multiple threads of execution. Multithreading allows an application to have concurrency and parallelism (on multiprocessor systems).

Mutex
A mutex is a mutual exclusion synchronization primitive. Mutexes provide threads with the ability to regulate or serialize access to process shared data and resources. When a thread locks a mutex, other threads trying to lock the mutex block until the owning thread unlocks the mutex.

POSIX
Portable Operating System Interface. POSIX defines a set of standards that multiple vendors conform to in order to provide for application portability. The Pthreads standard (POSIX 1003.1c) provides a set of portable multithreading APIs to application developers.

Priority Inversion
A situation where a low-priority thread has acquired a resource that is needed by a higher priority thread. As the resource cannot be acquired, the higher priority thread must wait for the resource. The end result is that a low-priority thread blocks a high-priority thread.

Process
A process can be thought of as a container for one or more threads of execution, an address space, and shared process resources. All processes have at least one thread. Each thread in the process executes within the process' address space. Examples of process-shared resources are open file descriptors, message queue descriptors, mutexes, and semaphores.

Process Control Block (PCB)
This structure holds the register context of a process.

Process Structure
The operating system maintains a process structure for each process in the system. This structure represents the actual process internally in the system. A sample of process structure information includes the process ID, the process' set of open files, and the signal vector. The process structure and the values contained within it are part of the context of a process.

Program Counter (PC)
The program counter is part of the register context of a process. It holds the address of the current instruction to be executed.

Race Condition
When the result of two or more threads performing an operation depends on unpredictable timing factors, this is a race condition.

Read-Write Lock
A read-write lock is a synchronization primitive. Read-write locks provide threads with the ability to regulate or serialize access to process-shared data and resources. Read-write locks allow multiple readers to concurrently acquire the read lock whereas only one writer at a time may acquire the write lock. These locks are useful for shared data that is mostly read and only rarely written.

Reentrant Function
A reentrant function is one that when called by multiple threads, behaves as if the function was called serially, one after another, by the different threads. These functions may execute in parallel.

Scheduling Allocation Domain
The set of processors on which a thread is scheduled. The size of this domain may dynamically change over time. Threads may also be moved from one domain to another.

Scheduling Contention Scope
The scheduling contention scope defines the group of threads that a thread competes with for access to resources. The contention scope is most often associated with access to a processor. However, this scope may also be used when threads compete for other resources. Threads with the system scope compete for access to resources with all other threads in the system. Threads with the process scope compete for access to resources with other process scope threads in the process.

Scheduling Policy
A scheduling policy is a set of rules used to determine how and when multiple threads are scheduled to execute. The scheduling policy also determines how long a thread is allowed to execute.

Scheduling Priority
A scheduling priority is a numeric priority value assigned to threads in certain scheduling policies. Threads with higher priorities are given preference when scheduling decisions are made.

Semaphore
A semaphore is similar to a mutex. A semaphore regulates access to one or more shared objects. A semaphore has a value associated with it. The value is generally set to the number of shared resources regulated by the semaphore. When a semaphore has a value of one, it is a binary semaphore. A mutex is essentially a binary semaphore. When a semaphore has a value greater than one, it is known as a *counting semaphore*. A counting semaphore can be locked by multiple threads simultaneously. Each time the semaphore is locked, the value is decremented by one. After the value reaches zero, new attempts to lock the semaphore cause the locking thread to block until the semaphore is unlocked by another thread.

Shared Object
A shared object is a tangible entity that exists in the address space of a process and is accessible by all threads within the process. In the context of multithreaded programming, "shared objects" are global variables, file descriptors, and other such objects that require access by threads to be synchronized.

Signal
A signal is a simplified IPC mechanism that allows a process or thread to be notified of an event. Signals can be generated synchronously and asynchronously.

Signal Mask
A signal mask determines which signals a thread accepts and which ones are blocked from delivery. If a synchronous signal is blocked from delivery, it is held pending until either the thread unblocks the signal or the thread terminates. If an asynchronous signal delivered to the process is blocked from delivery by a thread, the signal may be handled by a different thread in the process that does not have the signal blocked.

Signal Vector
A signal vector is a table contained in each process that describes the action that should be taken when a signal is delivered to a thread within the process. Each signal has one of three potential behaviors: ignore the signal, execute a signal-handling function, or perform the default action of the signal (usually process termination).

Single-Threaded
means that there is only one *flow of control* (one thread) through the program code; only one instruction is executed at a time.

Spinlock
A synchronization primitive similar to a mutex. If the lock cannot be acquired, instead of blocking, the thread wishing to acquire the lock spins in a loop until the lock can be acquired. Spinlocks can be easily used improperly and can severely degrade performance if used on a single processor system.

Spurious Wakeup
A spurious wakeup occurs when a thread is incorrectly unblocked, even though the event it was waiting for has not occurred. A condition wait that is interrupted and returns because the blocked thread received a normal signal is an example of a spurious wakeup.

Stack
A stack is used by a thread to make function calls (and return from those calls), to pass arguments to a function call, and to create the space for local variables when in that function call. Bound threads have a user stack and a kernel stack. Unbound threads have only a user stack.

Synchronous Signal
A synchronous signal is a signal that has been generated due to some action of a specific thread. For example, when a thread does a divide by zero, causes a floating point exception, or executes an illegal instruction, a signal is generated synchronously. Synchronous signals are delivered to the thread that caused the signal to be sent.

Traditional Process
This is a single-threaded entity that can be scheduled to execute on a processor.

Thread
A thread is an independent flow of control within a process, composed of a context (which includes a register set and program counter) and a sequence of instructions to execute.

Thread Local Storage (TLS)
Thread local storage is essentially thread-specific data requiring support from the compilers. With TLS, an application can allocate the actual data as thread-specific data rather than using thread-specific data keys. Additionally, TLS does not require the thread to make a function call to obtain thread-specific data. The thread can access the data directly.

Thread-Safe Function
A thread-safe function is one that may be safely called by multiple threads at the same time. If the function accesses shared data or resources, this access is regulated by a mutex or some other form of synchronization.

Thread-Specific Data (TSD)
Thread-specific data is global data that is specific to a thread. All threads access the same data variable. However, each thread has its own thread-specific value associated with this variable. errno is an example of thread-specific data.

Thread Structure
The operating system maintains a thread structure for each thread in the system. This structure represents the actual thread internally in the system. A sample of thread structure information includes the thread ID, the scheduling policy and priority, and the signal mask. The thread structure and the values contained within it are part of the context of a thread.

User Mode
A mode of operation where a subset of operations are allowed. While a thread is executing an applications code, it is executing in user mode. When the thread makes a system call, it changes modes and executes in kernel mode until the system call completes.

User Space
The user code exists in this space. User code is executed in this space at the normal privilege level. In general, there are two privilege levels: one for user code (user mode) and the other for kernel code (kernel mode).

User Stack
When a thread is executing code in user space, it needs to use a stack to make function calls, pass parameters, and create local variables. While in user mode, a thread does not use the kernel stack. Instead, a separate user stack is allocated for use by each user thread. See *Stack* for a generic description of a stack.

User Thread
When **`pthread_create()`** is called, a user thread is created. Whether a kernel-scheduled entity (kernel thread or lightweight process) is also created depends on the user thread's scheduling contention scope. When a bound thread is created, both a user thread and a kernel-scheduled entity are created. When an unbound thread is created, generally only a user thread is created. See *Thread* for a generic description of a thread.

SEE ALSO
ThreadTime by Scott J. Norton and Mark D. DiPasquale, Prentice-Hall, ISBN 0-13-190067-6, 1996.

NAME
pthread_atfork() - register fork handlers.

SYNOPSIS
```
#include <pthread.h>

int pthread_atfork(
    void (*prepare)(void),
    void (*parent)(void),
    void (*child)(void)
);
```

PARAMETERS

prepare This function is called before performing the **fork()**.

parent This function is called in the parent process after performing the **fork()**.

child This function is called in the child process after performing the **fork()**.

DESCRIPTION
The **pthread_atfork()** function allows an application to install fork handlers. These fork handlers will be called before and after a **fork()** operation. These handlers will be called in the context of the thread calling **fork()**. Similar to the **atexit()** handlers, the application does not need to do anything special for these fork handlers to be called. They will be invoked by the system when a **fork()** operation occurs.

The **prepare()** function is called before the **fork()** operation in the parent process. The **parent()** function is called after the **fork()** operation in the parent process. The **child()** function is called after the **fork()** operation in the child process.

If a fork handler is not needed in one or more of these three places, the appropriate fork handler parameter may be set to **NULL**.

A process may install multiple fork handling functions. The **parent()** and **child()** fork handlers will be called in the order in which they were installed (i.e., **First-In, First-Out**). The **prepare()** fork handlers will be called in the opposite order (i.e., **Last-In, First-Out**).

RETURN VALUE
Upon successful completion, **pthread_atfork()** returns zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS
If any of the following occur, the **pthread_atfork()** function returns the corresponding error number:

[ENOMEM] There is insufficient table space to install the fork handlers.

AUTHOR
pthread_atfork() was derived from the IEEE POSIX P1003.1c standard.

SEE ALSO
atexit(2), fork(2).

STANDARDS CONFORMANCE
pthread_atfork(): POSIX 1003.1c.

NAME

pthread_attr_set*(), pthread_attr_get*() - set and get thread attributes

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);

int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detach-
    state);

int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);

int pthread_attr_getstacksize(const pthread_attr_t *attr, size_t *stack-
    size);

int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);

int   pthread_attr_getstackaddr(const   pthread_attr_t   *attr,   void
    **stackaddr);

int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);

int pthread_attr_getguardsize(const pthread_attr_t *attr, size_t *guard-
    size);

int pthread_attr_setinheritsched(pthread_attr_t *attr,
    int inheritsched);

int pthread_attr_getinheritsched(const pthread_attr_t *attr,
    int *inheritsched);

int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);

int  pthread_attr_getschedpolicy(const  pthread_attr_t  *attr,  int  *pol-
    icy);

int pthread_attr_setschedparam(pthread_attr_t *attr,
    const struct sched_param *param);

int pthread_attr_getschedparam(const pthread_attr_t *attr,
    struct sched_param *param);

int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);

int  pthread_attr_getscope(const  pthread_attr_t  *attr,  int  *contention-
    scope);

int pthread_attr_setprocessor_np(pthread_attr_t *attr,
    pthread_spu_t processor, int binding_type);

int pthread_attr_getprocessor_np(const pthread_attr_t *attr,
    pthread_spu_t *processor, int *binding_type);
```

PARAMETERS

attr Pointer to the thread attributes object whose attributes are to be set/retrieved.

detachstate This parameter either specifies the new value of the *detachstate* attribute (set function) or it points to the memory location where the *detachstate* attribute of *attr* is to be returned (get function).

stacksize This parameter either specifies the new value of the *stacksize* attribute (set function) or it points to the memory location where the *stacksize* attribute of *attr* is to be returned (get function).

stackaddr This parameter either specifies the new value of the *stackaddr* attribute (set function) or it points to the memory location where the *stackaddr* attribute of *attr* is to be returned (get function).

guardsize This parameter either specifies the new value of the *guardsize* attribute (set function) or it points to the memory location where the *guardsize* attribute of *attr* is to be returned (get function).

inheritsched This parameter either specifies the new value of the *inheritsched* attribute (set function) or it points to the memory location where the *inheritsched* attribute of *attr* is to be returned (get function).

policy This parameter either specifies the new value of the *schedpolicy* attribute (set function) or it points to the memory location where the *schedpolicy* attribute of *attr* is to be returned (get function).

param This parameter either specifies the new values of the *schedparam* attributes (set function) or it points to the memory location where the *schedparam* attributes of *attr* are to be returned (get function).

contentionscope
This parameter either specifies the new value of the *contentionscope* attribute (set function) or it points to the memory location where the *contentionscope* attribute of *attr* is to be returned (get function).

processor This parameter either specifies the new value of the *processor* attribute (set function) or it points to the memory location where the *processor* attribute of *attr* is to be returned (get function).

binding_type This parameter either specifies the new value of the *binding_type* attribute (set function) or it points to the memory location where the *binding_type* attribute of *attr* is to be returned (get function).

DESCRIPTION

These functions set and get the attributes as described below.

pthread_attr_setdetachstate() - set the detachstate attribute.
pthread_attr_getdetachstate() - get the detachstate attribute.

pthread_attr_setstacksize() - set the stacksize attribute.
pthread_attr_getstacksize() - get the stacksize attribute.

pthread_attr_setstackaddr() - set the stackaddr attribute.
pthread_attr_getstackaddr() - get the stackaddr attribute.

pthread_attr_setguardsize() - set the guardsize attribute.
pthread_attr_getguardsize() - get the guardsize attribute.

pthread_attr_setinheritsched() - set the inheritsched attribute.
pthread_attr_getinheritsched() - get the inheritsched attribute.

pthread_attr_setschedpolicy() - set the schedpolicy attribute.
pthread_attr_getschedpolicy() - get the schedpolicy attribute.

pthread_attr_setschedparam() - set the schedparam attributes.
pthread_attr_getschedparam() - get the schedparam attributes.

pthread_attr_setscope() - set the contentionscope attribute.
pthread_attr_getscope() - get the contentionscope attribute.

pthread_attr_setprocessor_np() - set the processor and binding_type attributes.
pthread_attr_getprocessor_np() - get the processor and binding_type attributes.

The attributes object *attr* must have previously been initialized with the function **pthread_attr_init()** before these functions are called.

ATTRIBUTE: detachstate

The legal values for the *detachstate* attribute are:

PTHREAD_CREATE_DETACHED
This option causes all threads created with *attr* to be in the detached state. The resources associated with threads having this state are reclaimed automatically by the system when the threads terminate. Calling the **pthread_detach()** or **pthread_join()** function for threads created with this attribute results in an error.

PTHREAD_CREATE_JOINABLE
This option causes all threads created with *attr* to be in the joinable state. The resources associated with threads having this state are not reclaimed when the threads terminate. An application must call the **pthread_detach()** or **pthread_join()** functions for threads created

with this attribute to reclaim the system resources.

The default value of *detachstate* is **PTHREAD_CREATE_JOINABLE**.

pthread_attr_setdetachstate() is used to set the *detachstate* attribute in the initialized attributes object *attr*. The new value of the *detachstate* attribute is passed to this function in the *detachstate* parameter.

pthread_attr_getdetachstate() retrieves the value of the *detachstate* attribute from the thread attributes object *attr*. This value is returned in the *detachstate* parameter.

ATTRIBUTE: stacksize

The legal values for the *stacksize* attribute are:

PTHREAD_STACK_MIN
This option specifies that the size of the user stack for threads created with this attributes object will be of default stack size. This value is the minimum stack size (in bytes) required for a thread. This minimum value may not be acceptable for all threads.

stacksize
This defines the size (in bytes) of the user stack for threads created with this attributes object. This value must be greater than or equal to the minimum stack size **PTHREAD_STACK_MIN**.

POSIX.1c does not define a default value. On HP-UX Release 10.30, the default value of the *stacksize* attribute is 64K.

pthread_attr_setstacksize() is used to set the *stacksize* attribute in the initialized attributes object *attr*. The new value of the *stacksize* attribute is passed to this function in the *stacksize* parameter.

pthread_attr_getstacksize() retrieves the value of the *stacksize* attribute from the thread attributes object *attr*. This value is returned in the *stacksize* parameter.

ATTRIBUTE: stackaddr

The legal values for the *stackaddr* attribute are:

NULL
This option specifies that the storage for the user stack of any threads created with this attributes object will be allocated and deallocated by the threads library. The application does not need to allocate and manage thread stacks.

stack_address
This option specifies the base address of a stack that the created thread will use. The application is completely responsible for allocating, managing, and deallocating these stacks. Some options for allocation of storage are the *malloc(3C)*, *brk(2)*, and *mmap(2)* functions. Note: if this option is used, only one thread should be created with this attributes object. If multiple threads are created, they will all use the same stack.

The default value of the *stackaddr* attribute is **NULL**.

pthread_attr_setstackaddr() is used to set the *stackaddr* attribute in the initialized attributes object *attr*. The new value of the *stackaddr* attribute is passed to this function in the *stackaddr* parameter.

pthread_attr_getstackaddr() retrieves the value of the *stackaddr* attribute from the thread attributes object *attr*. This value is returned in the *stackaddr* parameter.

The *guardsize* attribute is ignored if the storage for the thread's user stack is not allocated by the library (i.e., the *stackaddr* attribute is not **NULL**).

ATTRIBUTE: guardsize

The *guardsize* attribute allows an application to specify the size of the guard area for threads created with this attributes object. The size of the guard area is specified in bytes. Most systems will round up the guard size to a multiple of the system configurable variable **PAGESIZE**. If the value zero is specified, a guard area will not be created.

The default value of *guardsize* is **PAGESIZE** bytes. The actual value of **PAGESIZE** is implementation-dependent and may not be the same on all implementations. The *guardsize* attribute is ignored if the storage for the user stack is not allocated by the pthread library. The application is responsible for protecting against stack overflow.

pthread_attr_setguardsize() is used to set the *guardsize* attribute in the initialized attributes object *attr*. The new value of the *guardsize* attribute is passed to this function in the *guardsize* parameter.

pthread_attr_getguardsize() retrieves the value of the *guardsize* attribute from the thread attributes object *attr*. This value is returned in the *guardsize* parameter. If the guard area is rounded up to a multiple of **PAGESIZE**, a call to this function shall store in the *guardsize* parameter the guard size specified in the previous **pthread_attr_setguardsize()** function call.

ATTRIBUTE: inheritsched

The legal values for the *inheritsched* attribute are:

PTHREAD_INHERIT_SCHED
This option specifies that the scheduling policy and associated attributes are to be inherited from the creating thread. The scheduling policy and associated attributes in the *attr* argument will be ignored when a thread is created with *attr*.

PTHREAD_EXPLICIT_SCHED
This option specifies that the scheduling policy and associated attributes for the created thread(s) are to be taken from this attributes object. These values will not be inherited from the creating thread.

POSIX.1c does not define a default value for the *inheritsched* attribute. On HP-UX, the default value is **PTHREAD_INHERIT_SCHED**.

pthread_attr_setinheritsched() is used to set the *inheritsched* attribute in the initialized attributes object *attr*. The new value of the *inheritsched* attribute is passed in the *inheritsched* parameter.

pthread_attr_getinheritsched() retrieves the value of the *inheritsched* attribute from the thread attributes object *attr*. This value is returned in the *inheritsched* parameter.

ATTRIBUTE: schedpolicy

The *schedpolicy* attribute allows threads created with this attributes object to use a specific scheduling policy. To use this attribute, the *inheritsched* attribute must be set to **PTHREAD_EXPLICIT_SCHED**. For a complete list of valid scheduling policies, refer to *rtsched*(2) and <**sched.h**>.

POSIX.1c does not specify a default value for the *schedpolicy* attribute. On HP-UX, the default value for system scope threads is **SCHED_TIMESHARE**.

pthread_attr_setschedpolicy() is used to set the *schedpolicy* attribute in the initialized attributes object *attr*. The new value of the *schedpolicy* attribute is passed to this function in the *policy* parameter.

pthread_attr_getschedpolicy() retrieves the value of the *schedpolicy* attribute from the thread attributes object *attr*. This value is returned in the *policy* parameter.

ATTRIBUTE: schedparam

The legal values for the *schedparam* attribute associated with the *schedpolicy* attribute vary depending upon the scheduling policy. For the **SCHED_FIFO** and **SCHED_RR** scheduling policies, only the *sched_priority* member of the *schedparam* attribute is required. Legal values for *sched_priority* can be obtained through **sched_get_priority_max()** and **sched_get_priority_min()**. The required contents of *schedparam* for other scheduling policies is undefined. For a complete list of required and valid scheduling parameters for all scheduling policies, refer to *rtsched*(2) and <**sched.h**>.

pthread_attr_setschedparam() is used to set the *schedparam* attribute in the initialized attributes object *attr*. The new value of the *schedparam* attribute is passed to this function in the *param* parameter.

pthread_attr_getschedparam() retrieves the value of the *schedparam* attribute from the thread attributes object *attr*. This value is returned in the *param* parameter.

ATTRIBUTE: contentionscope

The legal values for the *contentionscope* attribute are:

PTHREAD_SCOPE_SYSTEM
Threads created with this contention scope contend for resources with all other threads in the system (and within the same scheduling domain). This attribute is generally used to indicate that the user thread should be bound directly to a kernel-scheduled entity.

PTHREAD_SCOPE_PROCESS
Threads created with this contention scope contend directly with other threads within their process that were created with this scheduling contention scope. This attribute is generally used to indicate that the user thread should be unbound (in the *M*x1 and *M*x*N* Threads Model).

This value is currently not supported on HP-UX release 10.30.

The default value of the *contentionscope* attribute is not defined by POSIX.1c. On HP-UX release 10.30, the current default value of the *contentionscope* attribute is **PTHREAD_SCOPE_SYSTEM**. Note: This default value will change to **PTHREAD_SCOPE_PROCESS** when HP-UX supports the *MxN* Threads Model.

pthread_attr_setscope() is used to set the *contentionscope* attribute in the initialized attributes object *attr*. The new value of the *contentionscope* attribute is passed to this function in the *contentionscope* parameter.

pthread_attr_getscope() retrieves the value of the *contentionscope* attribute from the thread attributes object *attr*. This value is returned in the *contentionscope* parameter.

ATTRIBUTES: processor And binding_type

The legal values for the *processor* attribute are:

PTHREAD_SPUINHERIT_NP
Threads created with this *processor* attribute inherit their processor binding attributes from the creating thread. This is the default value of the *processor* attribute. The *binding_type* attribute is ignored.

PTHREAD_SPUFLOAT_NP
Threads created with this *processor* attribute are allowed to execute on any processor the system chooses. No processor binding is maintained. The *binding_type* attribute is ignored.

(pthread_spu_t)*processor_id*
Threads created with this *processor* attribute are bound to the processor specified in the *processor* parameter. The type of binding (advisory or mandatory) is specified in the *binding_type* attribute.

The legal values for the *binding_type* attribute (if the *processor* attribute is not **PTHREAD_SPUINHERIT_NP** or **PTHREAD_SPUFLOAT_NP**) are:

PTHREAD_BIND_ADVISORY_NP
Threads created with this *binding_type* attribute have advisory processor binding. Refer to *pthread_processor_bind_np*(3T) for more information on advisory binding. This is the default value of the *binding_type* attribute.

PTHREAD_BIND_FORCED_NP
Threads created with this *binding_type* attribute have forced (or mandatory) processor binding. Refer to *pthread_processor_bind_np*(3T) for more information on forced binding.

The default value of the *processor* attribute is **PTHREAD_SPUINHERIT_NP**. The default value of the *binding_type* attribute is **PTHREAD_BIND_ADVISORY_NP**.

pthread_attr_setprocessor_np() is used to set the *processor* and *binding_type* attributes in the initialized attributes object *attr*. The new values of the *processor* and *binding_type* attributes are passed to this function in the *processor* and *binding_type* parameters, respectively.

pthread_attr_getprocessor_np() retrieves the values of the *processor* and *binding_type* attributes from the thread attributes object *attr*. These values are returned in the *processor* and *binding_type* parameters, respectively.

RETURN VALUE

Upon successful completion, the following functions return zero: **pthread_attr_setstacksize()**, **pthread_attr_getstacksize()**, **pthread_attr_setstackaddr()**, **pthread_attr_getstackaddr()**, **pthread_attr_setguardsize()**, **pthread_attr_getguardsize()**, **pthread_attr_setdetachstate()**, **pthread_attr_getdetachstate()**, **pthread_attr_setinheritsched()**, **pthread_attr_getinheritsched()**, **pthread_attr_setschedpolicy()**, **pthread_attr_getschedpolicy()**, **pthread_attr_setschedparam()**, **pthread_attr_getschedparam()**, **pthread_attr_setprocessor_np()**, **pthread_attr_getprocessor_np()**, **pthread_attr_setscope()**, and **pthread_attr_getscope()**. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

If any of the following occur, the **pthread_attr_setscope()**, **pthread_attr_getscope()**, **pthread_attr_setinheritsched()**, **pthread_attr_getinheritsched()**,

pthread_attr_setschedpolicy(), and **pthread_attr_getschedpolicy()** functions return the corresponding error number:

[ENOSYS] **_POSIX_THREAD_PRIORITY_SCHEDULING** is not defined and these functions are not supported.

If any of the following occur, the **pthread_attr_getstackaddr()** and **pthread_attr_setstackaddr()** functions return the corresponding error number:

[ENOSYS] **_POSIX_THREAD_ATTR_STACKADDR** is not defined and these functions are not supported.

If any of the following occur, the **pthread_attr_getstacksize()** and **pthread_attr_setstacksize()** functions return the corresponding error number:

[ENOSYS] **_POSIX_THREAD_ATTR_STACKSIZE** is not defined and these functions are not supported.

If any of the following occur, **pthread_attr_setstacksize()**, **pthread_attr_getstacksize()**, **pthread_attr_setstackaddr()**, **pthread_attr_getstackaddr()**, **pthread_attr_setguardsize()**, **pthread_attr_getguardsize()**, **pthread_attr_setdetachstate()**, **pthread_attr_getdetachstate()**, **pthread_attr_setinheritsched()**, **pthread_attr_getinheritsched()**, **pthread_attr_setschedpolicy()**, **pthread_attr_getschedpolicy()**, **pthread_attr_setschedparam()**, **pthread_attr_getschedparam()**, **pthread_attr_setprocessor_np()**, **pthread_attr_getprocessor_np()**, **pthread_attr_setscope()**, and **pthread_attr_getscope()** return the corresponding error number:

[EINVAL] The value specified by *attr* is invalid.

[EINVAL] The value specified by *stacksize* is less than the minimum required stacksize of **PTHREAD_STACK_MIN** or exceeds a system-imposed limit.

[EINVAL] *detachstate*, *guardsize*, *inheritsched*, *processor*, *binding_type*, *policy*, *param*, or *scope* contains an invalid value.

[ENOTSUP] The value contained in *policy* is not a supported value.

AUTHOR

pthread_attr_setstacksize(), **pthread_attr_getstacksize()**, **pthread_attr_setstackaddr()**, **pthread_attr_getstackaddr()**, **pthread_attr_setdetachstate()**, **pthread_attr_getdetachstate()**, **pthread_attr_setinheritsched()**, **pthread_attr_getinheritsched()**, **pthread_attr_setschedpolicy()**, **pthread_attr_getschedpolicy()**, **pthread_attr_setschedparam()**, **pthread_attr_getschedparam()**, **pthread_attr_setscope()**, and **pthread_attr_getscope()** were derived from the IEEE POSIX P1003.1c standard.

pthread_attr_setguardsize() and **pthread_attr_getguardsize()** were developed by X/Open.

pthread_attr_setprocessor_np() and **pthread_attr_getprocessor_np()** were developed by HP.

SEE ALSO

pthread_create(3T), pthread_attr_init(3T), pthread_processor_bind_np(3T).

STANDARDS CONFORMANCE

pthread_attr_setstacksize(): POSIX 1003.1c.
pthread_attr_getstacksize(): POSIX 1003.1c.
pthread_attr_setstacksize(): POSIX 1003.1c.
pthread_attr_getstacksize(): POSIX 1003.1c.
pthread_attr_setstackaddr(): POSIX 1003.1c.
pthread_attr_getstackaddr(): POSIX 1003.1c.
pthread_attr_setguardsize(): X/Open.
pthread_attr_getguardsize(): X/Open.
pthread_attr_setdetachstate(): POSIX 1003.1c.

pthread_attr_getdetachstate(): POSIX 1003.1c.
pthread_attr_setinheritsched(): POSIX 1003.1c.
pthread_attr_getinheritsched(): POSIX 1003.1c.
pthread_attr_setschedpolicy(): POSIX 1003.1c.
pthread_attr_getschedpolicy(): POSIX 1003.1c.
pthread_attr_setschedparam(): POSIX 1003.1c.
pthread_attr_getschedparam(): POSIX 1003.1c.
pthread_attr_setscope(): POSIX 1003.1c.
pthread_attr_getscope(): POSIX 1003.1c.
pthread_attr_setprocessor_np(): None.
pthread_attr_getprocessor_np(): None.

p

NAME

pthread_attr_init(), pthread_attr_destroy() - initialize or destroy a thread attributes object.

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_init(
   pthread_attr_t *attr
);

int pthread_attr_destroy(
   pthread_attr_t *attr
);
```

PARAMETERS

attr Pointer to the thread attributes object to be initialized or destroyed.

DESCRIPTION

pthread_attr_init() initializes a thread attributes object *attr* with the default value for all the thread attributes.

When a thread attributes object is used to create a thread, the values of the individual attributes determine the characteristics of the new thread. Attributes objects act like additional parameters to object creation. A single attributes object can be used in multiple calls to **pthread_create()**.

After a thread attributes object has been used to initialize one or more threads, any function affecting the attributes object does not affect the previously created threads.

The thread attributes and their default values are:

stacksize POSIX.1c does not define a default value. On HP-UX release 10.30, the default value is 64K.

guardsize The default value is **PAGESIZE** bytes.

stackaddr The default value is **NULL**.

detachstate The default value is **PTHREAD_CREATE_JOINABLE**.

contentionscope POSIX.1c does not define a default value. On HP-UX release 10.30, the current default value is **PTHREAD_SCOPE_SYSTEM**. Note: This default value will change to **PTHREAD_SCOPE_PROCESS** when HP-UX supports the *M*x*N* Threads Model.

inheritsched POSIX.1c does not define a default value. On HP-UX, the default value is **PTHREAD_INHERIT_SCHED**.

schedpolicy POSIX.1c does not define a default value. On HP-UX, the default value is **SCHED_TIMESHARE**.

schedparam POSIX.1c does not define a default value.

processor The default value is **PTHREAD_SPUINHERIT_NP**.

binding_type The default value is **PTHREAD_BIND_ADVISORY_NP**.

If an initialized thread attributes object is reinitialized, the initialization results in undefined behavior.

pthread_attr_destroy() destroys the thread attributes object *attr*. The destroyed thread attributes object ceases to exist and its resources are reclaimed. Referencing the object after it has been destroyed results in undefined behavior. A destroyed thread attributes object can be reinitialized using thread attribute initialization routine **pthread_attr_init()**.

Threads that have already been created using this attributes object are not affected by the destruction of the thread attributes object.

RETURN VALUE

Upon successful completion, **pthread_attr_init()** and **pthread_attr_destroy()** return zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

If any of the following occur, the **pthread_attr_init()** function returns the corresponding error number:

[ENOMEM] There is insufficient memory available in which to initialize the pthread attributes object.

[EINVAL] The value specified by *attr* is invalid.

If any of the following occur, the **pthread_attr_destroy()** function returns the corresponding error number:

[EINVAL] The value specified by *attr* is invalid.

AUTHOR

pthread_attr_init() and **pthread_attr_destroy()** were derived from the IEEE POSIX P1003.1c standard.

SEE ALSO

pthread_create(3T).

STANDARDS CONFORMANCE

pthread_attr_init(): POSIX 1003.1c.
pthread_attr_destroy(): POSIX 1003.1c.

p

NAME

pthread_cancel() - cancel execution of a thread

SYNOPSIS

```
#include <pthread.h>

int pthread_cancel(
   pthread_t thread
);
```

PARAMETERS

thread Target thread to be canceled.

DESCRIPTION

pthread_cancel() requests that *thread* (hereby referred to as target thread) be canceled. It allows a thread to terminate the execution of any thread in the process in a controlled manner.

The target thread's cancelability state and type determine when the cancellation takes effect. Cancellation only occurs when the target thread's cancelability state is **PTHREAD_CANCEL_ENABLE**. When the target thread's cancelability state is **PTHREAD_CANCEL_DISABLE**, cancellation requests against the target thread are held pending and will be acted upon when cancellation is enabled.

When the cancelability type is **PTHREAD_CANCEL_ASYNCHRONOUS** for the target *thread*, new or pending cancellation requests are acted upon at any time. When the target thread's cancelability type is **PTHREAD_CANCEL_DEFERRED**, cancellation requests are held pending until the target thread reaches a cancellation point (see below).

If the target thread's cancelability state is disabled, the cancelability type does not matter. When cancelability is enabled, the cancelability type will take effect.

When the cancellation is acted on, the cancellation cleanup handlers for *thread* are called. The cancellation cleanup handlers are called in the opposite order in which they were installed. When the last cancellation cleanup handler returns, the thread-specific data destructor functions for *thread* are called. When the last destructor function returns, *thread* shall be terminated.

The caller of **pthread_cancel()** will not wait for the target thread to be canceled.

Cancellation Points

Cancellation points are points inside of certain functions where a thread must act on any pending cancellation request when cancelability is enabled if the function would block.

RETURN VALUE

Upon successful completion, **pthread_cancel()** returns zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

For each of the following conditions, if the condition is detected, the **pthread_cancel()** function returns the corresponding error number:

[ESRCH] No thread could be found corresponding to *thread*.

WARNINGS

Use of asynchronous cancelability while holding resources that need to be released may result in resource loss. Applications must carefully follow static lexical scoping rules in their execution behavior. For instance, the use of **setjmp()**, return, goto, etc., to leave user-defined cancellation scopes without doing the necessary scope pop will result in undefined behavior.

AUTHOR

pthread_cancel() was derived from the IEEE POSIX P1003.1c standard.

SEE ALSO

pthread_exit(3T), pthread_join(3T), pthread_setcancelstate(3T), pthread_cleanup_pop(3T), pthread_cond_wait(3T).

STANDARDS CONFORMANCE

`pthread_cancel()`: POSIX 1003.1c.

NAME

pthread_cleanup_push(), pthread_cleanup_pop() - register or remove a cancellation cleanup handler

SYNOPSIS

```
#include <pthread.h>

void pthread_cleanup_push(
   void (*routine)(void *),
   void *arg
);

void pthread_cleanup_pop(
   int execute
);
```

PARAMETERS

routine	Routine registered as a cancellation cleanup handler.
arg	Parameter to be passed to the cancellation cleanup handler **routine()**.
execute	Indicates if the popped cancellation cleanup handler is to be executed.

DESCRIPTION

pthread_cleanup_push() installs the cancellation cleanup handler *routine* onto the calling thread's cancellation cleanup stack. This handler will be popped from the calling thread's cancellation cleanup stack and called with the *arg* parameter when any of the following occur:

(a) the thread calls **pthread_exit()** or returns from its start routine.

(b) the thread acts upon a cancellation request.

(c) the thread calls **pthread_cleanup_pop()** with a non-zero *execute* argument.

When a thread terminates, it will execute each of the cancellation cleanup handlers on its cancellation cleanup stack. These handlers will be popped and executed in the reverse order that they were installed (**Last-In, First-Out**).

pthread_cleanup_pop() removes the cancellation cleanup handler at the top of the calling thread's cancellation stack. If *execute* is non-zero, the cancellation cleanup handler is called after it is removed from the cancellation stack. If *execute* is zero, the cancellation cleanup handler is simply removed and will not be called.

pthread_cleanup_push() and **pthread_cleanup_pop()** must appear as statements and in pairs within the same lexical scope. These functions may be macros which contain the opening '{' in the push function and the closing '}' in the pop function.

Calling **longjmp()** or **siglongjmp()** is undefined if there have been any calls to **pthread_cleanup_push()** or **pthread_cleanup_pop()** made without the matching call since the jump buffer was filled.

Calling **longjmp()** or **siglongjmp()** from inside a cancellation cleanup handler results in undefined behavior unless the corresponding **setjmp()** or **sigsetjmp()** was also done inside the cancellation cleanup handler.

RETURN VALUE

The **pthread_cleanup_push()** and **pthread_cleanup_pop()** functions must be used as statements. They do not have return values or errors.

ERRORS

None.

WARNINGS

The functions **pthread_cleanup_push()** and **pthread_cleanup_pop()** must be called in the same lexical scope or the result is undefined behavior.

AUTHOR

pthread_cleanup_push() and **pthread_cleanup_pop()** were derived from the IEEE POSIX P1003.1c standard.

SEE ALSO
pthread_cancel(3T), pthread_setcancelstate(3T).

STANDARDS CONFORMANCE
`pthread_cleanup_push()`: POSIX 1003.1c.
`pthread_cleanup_pop()`: POSIX 1003.1c.

NAME

pthread_cond_init(), pthread_cond_destroy() - initialize or destroy a condition variable.

SYNOPSIS

```
#include <pthread.h>

int pthread_cond_init(
   pthread_cond_t *cond,
   const pthread_condattr_t *attr
);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int pthread_cond_destroy(
   pthread_cond_t *cond
);
```

PARAMETERS

cond Pointer to the condition variable to be initialized or destroyed.

attr Pointer to the attributes object that defines the characteristics of the condition variable to be initialized. If the pointer is **NULL**, default attributes are used.

DESCRIPTION

The **pthread_cond_init()** function initializes the condition variable *cond* with the attributes *attr*. If *attr* is **NULL**, the default condition variable attributes are used to initialize the attributes object. Refer to **pthread_condattr_init()** for a list of the default condition variable attributes. After successful initialization, the condition variable may be used in condition variable operations. A condition variable should be initialized only once or the resulting behavior is undefined. The **pthread_once()** function provides a way to ensure that a condition variable is only initialized once.

The macro **PTHREAD_COND_INITIALIZER** can be used to initialize condition variables that are statically allocated. These condition variables will be initialized with default attributes. The **pthread_cond_init()** function does not need to be called for statically initialized condition variables.

If the *process-shared* attribute in the condition variable attributes object referenced by *attr* is defined as **PTHREAD_PROCESS_SHARED**, the condition variable must be allocated such that the processes sharing the condition variable have access to it. This may be done through the memory-mapping functions (see *mmap*(2)) or the shared memory functions (see *shmget*(2)).

pthread_cond_destroy() destroys the condition variable *cond*. This function may set *cond* to an invalid value. The destroyed condition variable can be reinitialized using the function **pthread_cond_init()**. If the condition variable is used after destruction in any condition variable call, the resulting behavior is undefined.

A condition variable should be destroyed only when there are no threads currently blocked on it. Destroying a condition variable that is currently in use results in undefined behavior.

RETURN VALUE

Upon successful completion, **pthread_cond_init()** and **pthread_cond_destroy()** return zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

If any of the following occur, the **pthread_cond_init()** function returns the corresponding error number:

[EAGAIN] The system does not have the available resources (other than memory) to initialize the condition variable.

[ENOMEM] There is insufficient memory available in which to initialize the condition variable.

For each of the following conditions, if the condition is detected, the **pthread_cond_init()** function returns the corresponding error number:

[EINVAL] The value specified by *cond* or *attr* is invalid.

[EBUSY] The specified condition variable is an already initialized condition variable.

[EFAULT] The *cond* parameter points to an illegal address.

For each of the following conditions, if the condition is detected, the **pthread_cond_destroy()** function returns the corresponding error number:

[EINVAL] *cond* is not a valid condition variable.

[EBUSY] An attempt to destroy *cond* while it is in use by another thread.

WARNINGS

The space for condition variable must be allocated before calling **pthread_cond_init()**. Undefined behavior will result if the *process-shared* attribute of *attr* is **PTHREAD_PROCESS_SHARED** and the space allocated for the condition variable is not accessible to cooperating threads.

AUTHOR

pthread_cond_init() and **pthread_cond_destroy()** were derived from the IEEE POSIX P1003.1c standard.

SEE ALSO

pthread_cond_wait(3T), pthread_cond_signal(3T).

STANDARDS CONFORMANCE

pthread_cond_init(): POSIX 1003.1c.
pthread_cond_destroy(): POSIX 1003.1c.

p

NAME

pthread_cond_signal(), pthread_cond_broadcast() - unblock one or all threads waiting on a condition variable.

SYNOPSIS

```
#include <pthread.h>

int pthread_cond_signal(
   pthread_cond_t *cond
);

int pthread_cond_broadcast(
   pthread_cond_t *cond
);
```

PARAMETERS

cond Pointer to the condition variable to be signaled or broadcast.

DESCRIPTION

The **pthread_cond_signal()** function is used to wake-up one of the threads that are waiting for the occurrence of a condition associated with condition variable *cond*. If there are no threads blocked on *cond*, this function has no effect. If more than one thread is blocked on *cond*, the scheduling policy determines which thread is unblocked. It is possible that more than one thread can be unblocked due to a spurious wakeup.

The **pthread_cond_broadcast()** function is used to wake-up all threads that are waiting for the occurrence of a condition associated with the condition variable *cond*. If there are no threads blocked on *cond*, this function has no effect. If more than one thread is blocked on *cond*, the scheduling policy determines the order in which threads are unblocked.

The condition variabled denoted by *cond* must have been dynamically initialized by a call to **pthread_cond_init()** or statically initialized with the macro **PTHREAD_COND_INITIALIZER**.

An unblocked thread will reacquire the mutex it held when it started the condition wait before returning from **pthread_cond_wait()** or **pthread_cond_timedwait()**. The threads that are unblocked contend for the mutex according to their scheduling policy and priority.

The **pthread_cond_signal()** or **pthread_cond_broadcast()** functions can be called by a thread whether or not it currently owns the condition variable's associated mutex. For predictable scheduling behavior and to prevent lost wake-ups, the mutex should be held when signaling a condition variable.

Usage

When using condition variables, there is a boolean predicate associated with each condition wait. If this predicate is false, the thread should do a condition wait. Spurious wakeups may occur when waiting on a condition variable. Because the return values from **pthread_cond_wait()** and **pthread_cond_timedwait()** do not imply anything about the value of this predicate, the predicate should always be re-evaluated.

Applications using condition variables typically acquire a mutex and enter a loop which checks the predicate. Depending on the value of the predicate, the thread either breaks out of the loop or waits on the condition. On return from the condition wait, the predicate is re-evaluated.

RETURN VALUE

Upon successful completion, **pthread_cond_signal()** and **pthread_cond_broadcast()** return zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

For each of the following conditions, if the condition is detected, the **pthread_cond_signal()** and **pthread_cond_broadcast()** functions return the corresponding error number:

[EINVAL] *cond* is not a valid condition variable.

[EFAULT] *cond* parameter points to an illegal address.

AUTHOR

pthread_cond_signal() and **pthread_cond_broadcast()** were derived from the IEEE POSIX P1003.1c standard.

SEE ALSO
pthread_cond_init(3T), pthread_cond_wait(3T).

STANDARDS CONFORMANCE
`pthread_cond_signal()`: POSIX 1003.1c.
`pthread_cond_broadcast()`: POSIX 1003.1c.

NAME

pthread_cond_wait(), pthread_cond_timedwait() - wait or timed wait on a condition variable.

SYNOPSIS

```
#include <pthread.h>

int pthread_cond_wait(
   pthread_cond_t *cond,
   pthread_mutex_t *mutex
);

int pthread_cond_timedwait(
   pthread_cond_t *cond,
   pthread_mutex_t *mutex,
   const struct timespec *abstime
);
```

PARAMETERS

cond Pointer to the condition variable to be waited on.

mutex Pointer to the mutex associated with the condition variable *cond*.

abstime Absolute time at which the wait expires, even if the condition has not been signaled or broadcast.

DESCRIPTION

The **pthread_cond_wait()** function is used to wait for the occurrence of a condition associated with the condition variable *cond*.

The **pthread_cond_timedwait()** function is used to wait a limited amount of time for the occurrence of a condition associated with the condition variable *cond*. The *abstime* parameter specifies the time at which this function should time out. If the absolute time specified by *abstime* passes and the indicated condition has not been signaled, the function returns an error to the caller. NOTE: *abstime* is the time at which the wait expires, not the length of time the thread will wait.

The condition variabled denoted by *cond* must have been dynamically initialized by a call to **pthread_cond_init()** or statically initialized with the macro **PTHREAD_COND_INITIALIZER**.

Both functions should be called with *mutex* locked by the calling thread. If *mutex* is not locked by the calling thread, undefined behavior will result. These functions atomically release *mutex* and cause the calling thread to block on the condition variable *cond*. If another thread is able to acquire the mutex after the about-to-block thread has released it but before it has actually blocked, a subsequent call to **pthread_cond_signal()** or **pthread_cond_broadcast()** by the other thread will behave as if it were issued after the about-to-block thread has blocked.

When the condition is signaled or the timed wait expires, the caller is unblocked and will reacquire *mutex* before returning. Whether these functions succeed or fail, *mutex* will always be reacquired before returning to the caller.

Using different mutexes for concurrent calls to these functions on the same condition variable results in undefined behavior.

When using condition variables, there is a predicate associated with the condition wait. If this predicate is false, the thread should perform a condition wait. Spurious wakeups may occur when waiting on a condition variable. A spurious wakeup occurs when a thread returns from a condition wait when it should really continue waiting. A normal signal being delivered to a thread may cause a spurious wakeup during a condition wait. Since the return values from **pthread_cond_wait()** and **pthread_cond_timedwait()** do not imply anything about the value of the predicate, the predicate should be re-evaluated.

A condition wait is a **cancellation point**. When the calling thread has deferred cancellation enabled, cancellation requested will be acted upon. If a cancellation request is acted upon while a thread is blocked in one of these functions, *mutex* is reacquired before calling the cancellation cleanup handlers. The cancellation cleanup handlers should release *mutex* so that application deadlock does not occur. If the condition signal and the cancellation request both occur, the canceled thread will not consume the condition signal (i.e., a different thread will be unblocked due to the condition signal).

If a signal is delivered to a thread waiting for a condition variable, upon return from the signal handler, the thread may return zero due to a spurious wakeup or continue waiting for the condition.

RETURN VALUE

Upon successful completion, **pthread_cond_wait()** and **pthread_cond_timedwait()** return zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

If any of the following occur, the **pthread_cond_timedwait()** function returns the corresponding error number.

[ETIMEDOUT]
abstime has passed and a condition signal has not been received.

For each of the following conditions, if the condition is detected, the **pthread_cond_wait()** and **pthread_cond_timedwait()** functions return the corresponding error number:

[EINVAL] The value specified by *cond*, *mutex* or *abstime* is invalid.

mutex is not owned by the calling thread. This error is not returned for a **PTHREAD_MUTEX_NORMAL** or **PTHREAD_MUTEX_DEFAULT** mutex on HP-UX.

[EFAULT] *cond*, *mutex* or *abstime* parameter points to an illegal address.

[EINVAL] Different mutexes are being used for *cond*. This error is not detected on HP-UX.

WARNINGS

It is important to note that when **pthread_cond_wait()** or **pthread_cond_timedwait()** return without error, the associated predicate may still be false. When **pthread_cond_timedwait()** returns with the timeout error, the associated predicate may be true. It is recommended that a condition wait be enclosed in the equivalent of a "while loop," which checks the predicate.

Undefined behavior results if these functions are called with a **PTHREAD_MUTEX_RECURSIVE** mutex.

EXAMPLES

pthread_cond_wait() is recommended to be used in a loop testing the predicate associated with it. This will take care of any spurious wakeups that may occur.

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

(void)pthread_mutex_lock(&mutex);
while (predicate == FALSE) {
     (void)pthread_cond_wait(&cond, &mutex);
}

(void)pthread_mutex_unlock(&mutex);
```

pthread_cond_timedwait() is also recommended to be used in a loop. This function can return success even if the predicate is not true. It should be called in a loop while checking the predicate. If the function times out, the predicate may still have become true. The predicate should be checked before processing the timeout case. The example given below does not do any other error checking.

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
struct timespec abstime;

(void)pthread_mutex_lock(&mutex);
abstime = absolute time to timeout.

while (predicate == FALSE) {
     ret = pthread_cond_timedwait(&cond, &mutex, &abstime);
     if (ret == ETIMEDOUT) {
          if (predicate == FALSE) {
               /* Code for time-out condition */
          } else {
               /* success condition */
```

```
                    break;
                }
            }
        }
        (void)pthread_mutex_unlock(&mutex);

        Code for success condition.
```

AUTHOR

pthread_cond_wait() and **pthread_cond_timedwait()** were derived from the IEEE POSIX P1003.1c standard.

SEE ALSO

pthread_cond_init(3T), pthread_cond_signal(3T).

STANDARDS CONFORMANCE

pthread_cond_wait(): POSIX 1003.1c.
pthread_cond_timedwait(): POSIX 1003.1c.

NAME

pthread_condattr_setpshared(), pthread_condattr_getpshared() - set or get the process-shared attribute

SYNOPSIS

```
#include <pthread.h>

int pthread_condattr_getpshared(
   const pthread_condattr_t *attr,
   int *pshared
);

int pthread_condattr_setpshared(
   pthread_condattr_t *attr,
   int pshared
);
```

PARAMETERS

attr Pointer to the condition variable attributes object whose *process-shared* attribute is to be set/retrieved.

pshared This parameter either specifies the new value of the *process-shared* attribute (set function) or it points to the memory location where the *process-shared* attribute of *attr* is to be returned (get function).

DESCRIPTION

The attributes object *attr* must have been previously initialized with the function **pthread_condattr_init()** before these functions are called.

The functions are used to set and retrieve the *process-shared* attribute in a condition variable attributes object. The legal values for the *process-shared* attribute are:

PTHREAD_PROCESS_SHARED
This option permits a condition variable to be operated upon by any thread that has access to the memory where the condition variable is allocated. The application is responsible for allocating the condition variable in memory that multiple processes can access.

PTHREAD_PROCESS_PRIVATE
The condition variable can be operated on only by threads created within the same process as the thread that initialized the condition variable. If threads of differing processes attempt to operate on such condition variable, the behavior is undefined.

The default value of *process-shared* is **PTHREAD_PROCESS_PRIVATE**.

pthread_condattr_setpshared() sets the *process-shared* attribute in *attr*. The new value of the *process-shared* attribute of *attr* is set to the value specified in the *pshared* parameter.

pthread_condattr_getpshared() retrieves the value of the *process-shared* attribute from *attr*. The value of the *process-shared* attribute of *attr* is returned in the *pshared* parameter.

RETURN VALUE

Upon successful completion, **pthread_condattr_getpshared()** and **pthread_condattr_setpshared()** return zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

If any of the following occur, the **pthread_condattr_getpshared()** and **pthread_condattr_setpshared()** functions return the corresponding error number:

[ENOSYS] **_POSIX_THREAD_PROCESS_SHARED** is not defined and these functions are not supported.

For each of the following conditions, if the condition is detected, the **pthread_condattr_setpshared()** function returns the corresponding error number:

[EINVAL] *attr* is not a valid condition variable attributes object.

[EINVAL] The value specified by *pshared* is not a legal value.

For each of the following conditions, if the condition is detected, the **pthread_condattr_getpshared()** function returns the corresponding error number:

[EINVAL] The value specified by *attr* or *pshared* is invalid.

WARNINGS

If a condition variable is created with the *process-shared* attribute defined as **PTHREAD_PROCESS_SHARED**, the cooperating processes should have access to the memory in which the condition variable is allocated.

AUTHOR

pthread_condattr_setpshared() and **pthread_condattr_getpshared()** were derived from the IEEE POSIX P1003.1c standard.

SEE ALSO

pthread_create(3T), pthread_condattr_init(3T), pthread_cond_init(3T), pthread_mutex_init(3T).

STANDARDS CONFORMANCE

pthread_condattr_setpshared(): POSIX 1003.1c.
pthread_condattr_getpshared(): POSIX 1003.1c.

NAME

pthread_condattr_init(), pthread_condattr_destroy() - initialize or destroy a condition variable attributes object.

SYNOPSIS

```
#include <pthread.h>

int pthread_condattr_init(
   pthread_condattr_t *attr
);

int pthread_condattr_destroy(
   pthread_condattr_t *attr
);
```

PARAMETERS

attr Pointer to the condition variable attributes object to be initialized or destroyed.

DESCRIPTION

pthread_condattr_init() initializes the condition variable attributes object *attr* with the default values for all attributes. The attributes object describes a condition variable in detail and is passed to the condition variable initialization function.

When a condition variable attributes object is used to initialize a condition variable, the values of the individual attributes determine the characteristics of the new condition variable. Attributes objects act like additional parameters to object initialization. A single attributes object can be used in multiple calls to the function **pthread_cond_init()**.

When a condition variable is initialized with an attributes object, the attributes are, in effect, copied into the condition variable. Consequently, any change to the attributes object will not affect any previously initialized condition variables. Once all condition variables needing a specific attributes object have been initialized, the attributes object is no longer needed.

The condition variable attributes and their default values are:

process-shared
The default value is **PTHREAD_PROCESS_PRIVATE**.

If an initialized condition variable attributes object is reinitialized, undefined behavior results.

pthread_condattr_destroy() destroys the condition variable attributes object *attr*. The destroyed condition variable attributes object ceases to exist and its resources are reclaimed. Using *attr* after it has been destroyed results in undefined behavior. A destroyed condition variable attributes object can be reinitialized using the function **pthread_condattr_init()**.

Condition variables that have been already initialized using this attributes object are not affected by the destruction of the condition variable attributes object.

RETURN VALUE

Upon successful completion, **pthread_condattr_init()** and **pthread_condattr_destroy()** return zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

If any of the following occur, the **pthread_condattr_init()** function returns the corresponding error number:

[ENOMEM] There is insufficient memory available in which to initialize the condition variable attributes object.

[EINVAL] *attr* is not a valid condition variable attributes object.

For each of the following conditions, if the condition is detected, the **pthread_condattr_destroy()** function returns the corresponding error number:

[EINVAL] *attr* is not a valid condition variable attributes object.

AUTHOR

pthread_condattr_init() and **pthread_condattr_destroy()** were derived from the IEEE POSIX P1003.1c standard.

SEE ALSO
pthread_create(3T), pthread_condattr_getpshared(3T), pthread_cond_init(3T).

STANDARDS CONFORMANCE
`pthread_condattr_init()`: POSIX 1003.1c.
`pthread_condattr_destroy()`: POSIX 1003.1c.

NAME

pthread_create() - create a new thread of execution.

SYNOPSIS

```
#include <pthread.h>

int pthread_create(
   pthread_t *thread,
   const pthread_attr_t *attr,
   void *(*start_routine)(void *),
   void *arg
);
```

PARAMETERS

thread Pointer to the location where the created thread's ID is to be returned.

attr Pointer to the thread attributes object describing the characteristics of the created thread. If the value is **NULL**, default attributes will be used.

start_routine
Function to be executed by the newly created thread.

arg Parameter to be passed to the created thread's *start_routine*.

DESCRIPTION

The **pthread_create()** function is used to create a new independent thread within the calling process. The thread will be created according to the attributes specified by *attr*. If *attr* is **NULL**, the default attributes will be used. The values of the attributes in *attr* describe the characteristics of the to-be-created thread in detail. Refer to the function **pthread_attr_init()** for a list of the default attribute values. A single attributes object can be used in multiple calls to the function **pthread_create()**.

When a thread is created with an attributes object, the attributes are, in effect, copied into the created thread. Consequently, any change to the attributes object will not affect any previously created threads. Once all threads needing a specific attributes object have been created, the attributes object is no longer needed and may be destroyed.

When the new thread is created, it will execute **start_routine()**, which has only one parameter, *arg*. If **start_routine()** returns, an implicit call to **pthread_exit()** is made. The return value of **start_routine()** is used as the thread's exit status.

The created thread's scheduling policy and priority, contention scope, detach state, stack size, and stack address are initialized according to their respective attributes in *attr*. The thread's signal mask is inherited from the creating thread. The thread's set of pending signals is cleared.

Refer to *pthread_exit*(3T), *pthread_detach*(3T), and *pthread_join*(3T) for more information on thread termination and synchronizing with terminated threads.

On success, the ID of the created thread is returned in *thread*. If **pthread_create()** fails, a thread is not created and the contents of *thread* are undefined.

Thread IDs are guaranteed to be unique only within a process.

NOTE: If the main thread returns from **main**(), an implicit call to **exit**() is made. The return value of **main**() is used as the process' exit status. The main thread can terminate without causing the process to terminate by calling **pthread_exit()**.

RETURN VALUE

Upon successful completion, **pthread_create()** returns zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

If any of the following occur, the **pthread_create()** function returns the corresponding error number:

[EINVAL] *attr* in an invalid thread attributes object.

[EINVAL] The value specified by *thread* is invalid.

[EAGAIN] The necessary resources to create another thread are not available, or the number of threads in the calling process already equals **PTHREAD_THREADS_MAX**.

[EINVAL] The scheduling policy or scheduling attributes specified in *attr* are invalid.

[EPERM] The caller does not have the appropriate privileges to create a thread with the scheduling policy and parameters specified in *attr*.

NOTES

It is unspecified whether joinable threads that have exited but haven't been joined count against the **PTHREAD_THREADS_MAX** limit.

AUTHOR

pthread_create() was derived from the IEEE POSIX P1003.1c standard.

SEE ALSO

pthread_exit(3T), pthread_join(3T), fork(2).

STANDARDS CONFORMANCE

pthread_create(): POSIX 1003.1c.

NAME

pthread_default_stacksize_np() - change the default stacksize.

SYNOPSIS

```
#include <pthread.h>

int pthread_default_stacksize_np(
   size_t new_size,
   size_t *old_size
);
```

PARAMETERS

new_size The new default stack size.

old_size Pointer to where the old default stack size is returned.

DESCRIPTION

The **pthread_default_stacksize_np()** function allows an application to change the default value for the *stacksize* attribute. This function must be called before any threads have been created. The new default stack size is passed in the *new_size* parameter. If not **NULL**, the previous default stack size is returned in *old_size*. If *new_size* is zero, this function can be used (at any time) to query the current default stack size.

On HP-UX, threads with default stack sizes are cached after they terminate. The next time a thread is created with a default stack size, a cached thread (and its stack) are reused. This can result in significant performance improvements for **pthread_create()**.

However, if the default stack size is not appropriate for an application, it cannot take advantage of this performance enhancement. By using the **pthread_default_stacksize_np()** function, the threads library will change the default stack size so that it matches the applications needs. This allows the application to utilize the performance benefit of cached threads.

RETURN VALUE

Upon successful completion, **pthread_default_stacksize_np()** returns zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

If any of the following occur, the **pthread_default_stacksize_np()** function returns the corresponding error number:

[EINVAL] The value specified by *new_size* is less than **PTHREAD_STACK_MIN**.

[EPERM] The calling process has already created threads (this must be called before any threads are created).

AUTHOR

pthread_default_stacksize_np() was developed by HP.

SEE ALSO

pthread_attr_getstacksize(3T), pthread_attr_setstacksize(3T).

STANDARDS CONFORMANCE

pthread_default_stacksize_np(): None.

NAME

pthread_detach() - mark a thread as detached to reclaim its resources when it terminates.

SYNOPSIS

```
#include <pthread.h>

int pthread_detach(
   pthread_t thread
);
```

PARAMETERS

thread Thread whose resources are to be reclaimed immediately when it terminates.

DESCRIPTION

pthread_detach() is used to detach the thread *thread*. When *thread* terminates, its resources will automatically be reclaimed by the system. If *thread* has already terminated, **pthread_detach()** causes the resources of *thread* to be reclaimed by the system.

pthread_detach() does not cause *thread* to terminate.

Once a detached thread has terminated, its resources, including the thread ID, may be reused by the system. The return status of a detached thread is lost when the thread terminates.

Calling this function multiple times for the same thread results in undefined behavior.

RETURN VALUE

Upon successful completion, **pthread_detach()** returns zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

If any of the following occur, the **pthread_detach()** function returns the corresponding error number:

[EINVAL] *thread* does not refer to a joinable thread.

[ESRCH] No thread could be found corresponding to *thread*.

AUTHOR

pthread_detach() was derived from the IEEE POSIX P1003.1c standard.

SEE ALSO

pthread_create(3T), pthread_join(3T), wait(2).

STANDARDS CONFORMANCE

pthread_detach(): POSIX 1003.1c.

NAME
pthread_equal() - compare two thread identifiers.

SYNOPSIS
```
#include <pthread.h>

int pthread_equal(
   pthread_t t1,
   pthread_t t2
);
```

PARAMETERS

t1	First thread ID to be compared.
t2	Second thread ID to be compared.

DESCRIPTION
pthread_equal() compares the thread IDs *t1* and *t2*. **Thread IDs** are opaque data types. They should be compared only with this function.

RETURN VALUE
pthread_equal() returns a nonzero value if the two thread IDs are equal; otherwise, zero is returned.

This function does not verify that *t1* or *t2* are valid thread IDs.

ERRORS
None.

AUTHOR
pthread_equal() was derived from the IEEE POSIX P1003.1c standard.

SEE ALSO
pthread_create(3T), pthread_self(3T).

STANDARDS CONFORMANCE
pthread_equal(): POSIX 1003.1c.

p

NAME
pthread_exit() - cause the calling thread to terminate.

SYNOPSIS
```
#include <pthread.h>
void pthread_exit(
   void *value_ptr
);
```

PARAMETERS
value_ptr The calling thread's exit status.

DESCRIPTION
pthread_exit() terminates the calling thread. The calling thread returns an exit status in *value_ptr*. This value is returned to a joining thread calling **pthread_join()** on the terminating thread. Only threads created with the *detachstate* attribute value **PTHREAD_CREATE_JOINABLE** can return an exit status to **pthread_join()**. The exit status of a detached thread is lost when the thread terminates.

When a thread terminates, process-shared resources are not released. Examples of process-shared resources include mutexes, condition variables, semaphores, message queue descriptors, and file descriptors. The **atexit()** routines are not called when a thread terminates as this is a process termination action.

An implicit call to **pthread_exit()** is made when a thread returns from its start routine. The function's return value serves as the thread's exit status (see *pthread_create*(3T)). If the main thread returns from **main()** without calling **pthread_exit()**, the process will exit using the return value from **main()** as the exit status. If the main thread calls **pthread_exit()**, the process will continue executing until the last thread terminates or a thread calls **exit()**. After the last thread in the process terminates, the process will exit with an exit status of zero.

Any installed cancellation cleanup handlers will be popped and executed in the reverse order that they were installed. After the cancellation cleanup handlers have been executed, if the thread has any non-**NULL** thread-specific data values with associated destructor functions, the destructor functions are called. The order in which these destructor functions are called is unspecified.

Calling **pthread_exit()** from a cancellation cleanup handler or destructor function that was invoked because of thread termination results in undefined behavior.

After a thread has terminated, the result of access to local (auto) variables of the thread is undefined. The terminating thread should not use local variables for the *value_ptr* parameter value.

RETURN VALUE
None.

ERRORS
None, this function does not return.

AUTHOR
pthread_exit() was derived from the IEEE POSIX P1003.1c standard.

SEE ALSO
pthread_create(3T), pthread_join(3T), exit(2), wait(2).

STANDARDS CONFORMANCE
pthread_exit(): POSIX 1003.1c.

NAME

pthread_setconcurrency(), pthread_getconcurrency() - set or get the concurrency level of unbound threads

SYNOPSIS

```
#include <pthread.h>

int pthread_setconcurrency(
   int new_level
);

int pthread_getconcurrency(void);
```

PARAMETERS

new_level New concurrency level for the unbound threads in the calling process.

DESCRIPTION

The unbound threads in a process may or may not be required to be simultaneously active. By default, the threads implementation ensures that a sufficient number of threads are active so that the process can continue to make progress. While this conserves system resources, it may not produce the most effective level of concurrency. The **pthread_setconcurrency()** function allows an application to inform the threads implementation of its desired concurrency level, *new_level*. The actual level of concurrency provided by the system as a result of this function call is unspecified.

If *new_level* is zero, it will cause the implementation to maintain the concurrency level at its discretion as if **pthread_setconcurrency()** were never called.

The **pthread_getconcurrency()** function returns the value set by a previous call to **pthread_setconcurrency()**. If the **pthread_setconcurrency()** function was not previously called, this function returns zero to indicate that the system is maintaining the concurrency level.

Note: When an application calls **pthread_setconcurrency()**, it is informing the implementation of its desired concurrency level. The implementation will use this as a hint, not a requirement. A call to **pthread_getconcurrency()** immediately after a call to **pthread_setconcurrency()** may return a different concurrency level specified in **pthread_setconcurrency()**.

If the system does not support the multiplexing of user threads on top of several kernel-scheduled entities, the functions **pthread_getconcurrency()** and **pthread_setconcurrency()** functions will be provided for source code compatibility, but they shall have no effect when called. To maintain the function semantics, the *new_level* parameter will be saved when **pthread_setconcurrency()** is called so that a subsequent call to **pthread_getconcurrency()** will return the same value.

RETURN VALUE

If successful, **pthread_setconcurrency()** returns zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

The **pthread_getconcurrency()** function always returns the concurrency level set by a previous call to **pthread_setconcurrency()**. If the **pthread_setconcurrency()** function has never been called, **pthread_getconcurrency()** shall return zero.

ERRORS

If any of the following occur, the **pthread_setconcurrency()** function shall return the corresponding error number.

[EINVAL] The value specified by *new_level* is invalid.

[EAGAIN] The value specified by *new_level* would cause a system resource to be exceeded.

APPLICATION USAGE

Use of these functions changes the state of the underlying concurrency level upon which the application depends. Library developers are advised to not use the **pthread_getconcurrency()** and **pthread_setconcurrency()** functions as their use may conflict with an applications use of these functions.

AUTHOR

pthread_getconcurrency() and **pthread_setconcurrency()** were developed by X/Open.

SEE ALSO
pthread_num_processors_np(3T), pthread_processor_bind_np(3T), pthread_processor_id_np(3T).

STANDARDS CONFORMANCE
pthread_getconcurrency(): X/Open.
pthread_setconcurrency(): X/Open.

NAME

pthread_setschedparam(), pthread_getschedparam() - set or get the scheduling policy and associated parameters

SYNOPSIS

```
#include <pthread.h>

int pthread_getschedparam(
   pthread_t thread,
   int *policy,
   struct sched_param *param
);

int pthread_setschedparam(
   pthread_t thread,
   int policy,
   const struct sched_param *param
);
```

PARAMETERS

thread The thread whose scheduling policy and associated parameters are to be set/retrieved.

policy This parameter either points to the memory location where the scheduling policy of *thread* is returned (get function) or it specifies the new value of the scheduling policy for *thread* (set function).

param This parameter either points to the memory location where the scheduling parameters of *thread* are returned (get function) or it specifies the new scheduling parameters for *thread* (set function).

DESCRIPTION

These functions allow the scheduling policy and associated parameters of threads within a multithreaded process to be retrieved and changed. The legal values for the scheduling policy and associated scheduling parameters are defined in **<sched.h>**.

pthread_setschedparam() changes the scheduling policy and associated scheduling parameters for *thread* to the policy and the associated parameters provided in *policy* and *param*, respectively. On HP-UX, appropriate privileges are required to change the scheduling parameters of a thread. The calling process must have appropriate privileges or be a member of a group having **PRIV_RTSCHED** access to successfully call **pthread_setschedparam()**.

The **pthread_getschedparam()** function retrieves the scheduling policy and associated parameters for *thread* and stores those values in *policy* and *param*, respectively. The values returned represent the actual scheduling values, not any temporary values that may be in effect due to priority inheritance or priority ceiling features.

RETURN VALUE

Upon successful completion, **pthread_setschedparam()** and **pthread_getschedparam()** return zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

If any of the following occur, the **pthread_getschedparam()** and **pthread_setschedparam()** functions return the corresponding error number:

[ENOSYS] **_POSIX_THREAD_PRIORITY_SCHEDULING** is not defined and these functions are not supported.

For each of the following conditions, if the condition is detected, the **pthread_setschedparam()** function returns the corresponding error number:

[EINVAL] *policy* or one of the scheduling parameters in *param* is invalid.

[ENOTSUP] Either the policy or scheduling parameters contain an unsupported value.

[EPERM] The caller does not have permission to set either the scheduling policy specified in *policy* or the scheduling parameters specified in *param* for *thread*.

[ESRCH] No thread could be found corresponding to *thread*.

For each of the following conditions, if the condition is detected, the **pthread_getschedparam()** function returns the corresponding error number:

[ESRCH] No thread could be found corresponding to *thread*.

[EINVAL] The value specified by *policy* or *param* is invalid.

NOTES

For the **SCHED_FIFO** and **SCHED_RR** scheduling policies, only the *sched_priority* member of the *sched_param* structure is required in the associated scheduling parameters. All other scheduling policies have implementation-defined scheduling policies. Refer to the documentation for *rtsched*(2) and <**sched.h**> for further information on implementation-defined scheduling policies.

AUTHOR

pthread_getschedparam() and **pthread_setschedparam()** were derived from the IEEE POSIX P1003.1c standard.

SEE ALSO

pthread_attr_setschedparam(3T), pthread_attr_setschedpolicy(3T), pthread_attr_getschedparam(3T), pthread_attr_getschedpolicy(3T), rtsched(2).

STANDARDS CONFORMANCE

pthread_getschedparam(): POSIX 1003.1c.
pthread_setschedparam(): POSIX 1003.1c.

NAME

pthread_getspecific(), pthread_setspecific() - get or set the thread-specific data associated with a key

SYNOPSIS

```
#include <pthread.h>

void *pthread_getspecific(
   pthread_key_t key
);

int pthread_setspecific(
   pthread_key_t key,
   const void *value
);
```

PARAMETERS

key Thread-specific data key whose value for the calling thread is to be set or retrieved.

value Value to be assigned to the thread-specific data key for the calling thread.

DESCRIPTION

The **pthread_getspecific()** function returns the thread-specific data value associated with *key* for the calling thread. If no value has been associated with *key* for the calling thread, **NULL** is returned.

The **pthread_setspecific()** function associates the thread-specific data *value* with *key*. Each thread may bind a different value to *key*. These values are usually pointers to memory dynamically allocated by the calling thread.

key must be a valid thread-specific data key created by calling **pthread_key_create()**. If *key* is not a valid thread-specific data key, undefined behavior results when calling these functions.

These functions may be called from a thread-specific data destructor function. However, calling **pthread_setspecific()** from a destructor may result in lost storage.

RETURN VALUE

The function **pthread_getspecific()** returns the thread-specific data value associated with *key*. If no thread-specific data value is currently associated with *key*, the value **NULL** is returned.

If successful, **pthread_setspecific()** returns zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

No errors are returned by the **pthread_getspecific()** function.

If any of the following occur, the **pthread_setspecific()** function returns the corresponding error number:

[ENOMEM] There is insufficient memory available in which to associate *value* with *key*.

For each of the following conditions, if the condition is detected, the **pthread_setspecific()** function returns the corresponding error number:

[EINVAL] *key* is an invalid thread-specific data key.

AUTHOR

pthread_getspecific() and **pthread_setspecific()** were derived from the IEEE POSIX P1003.1c standard.

SEE ALSO

pthread_key_create(3T), pthread_key_delete(3T).

STANDARDS CONFORMANCE

pthread_getspecific(): POSIX 1003.1c.
pthread_setspecific(): POSIX 1003.1c.

NAME

pthread_join() - wait for the termination of a specified thread.

SYNOPSIS

```
#include <pthread.h>

int pthread_join(
   pthread_t thread,
   void **value_ptr
);
```

PARAMETERS

thread Thread whose termination is awaited by the caller.

value_ptr Pointer to the location where the exit status of *thread* is returned.

DESCRIPTION

The **pthread_join()** function waits for the termination of the target *thread*. If the target *thread* has already terminated, this function returns immediately. Only threads created with a *detachstate* attribute value of **PTHREAD_CREATE_JOINABLE** may be specified in the target *thread* parameter. On successful return from **pthread_join()**, the *value_ptr* argument, if it is not a null pointer, will contain the value passed to **pthread_exit()** by the terminating thread.

When a **pthread_join()** call returns successfully, the caller is guaranteed the target thread has terminated. If more than one thread calls **pthread_join()** for the same target thread, one thread is guaranteed to return successfully. Undefined behavior results for other callers specifying the same thread.

If the thread calling **pthread_join()** is canceled, the target thread shall not be joined. The exit status of the target *thread* will remain available for another thread to call **pthread_join()**.

If the target *thread* was canceled, its exit status is **PTHREAD_CANCELED**.

It is unspecified whether a thread that has exited, but remains unjoined, counts against the **{_POSIX_THREAD_THREADS_MAX}** limit.

RETURN VALUE

Upon successful completion, **pthread_join()** returns zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

If any of the following occur, the **pthread_join()** function returns the corresponding error number:

[EINVAL] The value specified by *thread* does not refer to a joinable thread.

[ESRCH] No thread could be found corresponding to *thread*.

For each of the following conditions, if the condition is detected, the **pthread_join()** function returns the corresponding error number:

[EDEADLK] This operation would result in process deadlock or *thread* specifies the calling thread.

AUTHOR

pthread_join() was derived from the IEEE POSIX P1003.1c standard.

SEE ALSO

pthread_create(3T), wait(2).

STANDARDS CONFORMANCE

pthread_join(): POSIX 1003.1c.

NAME

pthread_key_create(), pthread_key_delete() - create or delete a thread-specific data key.

SYNOPSIS

```
#include <pthread.h>

int pthread_key_create(
   pthread_key_t *key,
   void (*destructor)(void *)
);

int pthread_key_delete(
   pthread_key_t key
);
```

PARAMETERS

key This is either a pointer to the location where the new key value will to be returned (create function) or the thread-specific data key to be deleted (delete function).

destructor
Function to be called to destroy a data value associated with *key* when the thread terminates.

DESCRIPTION

pthread_key_create() creates a unique thread-specific data *key*. The *key* may be used by threads within the process to maintain thread-specific data. The same key is used by all threads, but each thread has its own thread-specific value associated with *key*. For each thread, the value associated with *key* persists for the life of the thread.

A process may create up to **PTHREAD_KEYS_MAX** thread-specific data keys. When a new thread-specific data key is created, each thread will initially have the value **NULL** associated with the new key. Each time a thread is created, the new thread will have the value **NULL** for each thread-specific data key that has been created in the process. A thread may use **pthread_setspecific()** to change the value associated with a thread-specific data key. Note: **pthread_key_t** is an opaque data type.

When a thread terminates, it may have non-**NULL** values associated with some or all of its thread-specific data keys. Typically, these values will be pointers to dynamically allocated memory. If this memory is not released when the thread terminates, memory leaks in the process occur. An optional **destructor()** function may be provided at key creation time to destroy the thread-specific data of a terminating thread. When a thread terminates, the thread-specific data values associated with the thread will be examined. For each key that has a non-**NULL** thread-specific data value and a destructor function, the destructor function will be called with the thread-specific data value as its sole argument. The order in which destructor functions are called is unspecified.

Once all the destructor functions have been called, the thread-specific data values for the terminating thread are examined again. If there are still non-**NULL** values in which the associated keys have destructor functions, the process of calling destructor functions is repeated. If after **PTHREAD_DESTRUCTOR_ITERATIONS** iterations of this loop there are still some non-**NULL** values with associated destructors, the system may stop calling the destructors or continue calling the destructors until there are no non-**NULL** values. Note: This may result in an infinite loop.

If a destructor function is not desired for *key*, the value **NULL** may be passed in the *destructor* parameter.

The **pthread_key_delete()** function deletes a thread-specific data *key*. The *key* must have been previously created by **pthread_key_create()**. The thread-specific data values associated with *key* are not required to be **NULL** when this function is called. Using *key* after it has been deleted results in undefined behavior.

If a destructor function is associated with *key*, it will not be invoked by the **pthread_key_delete()** function. Once *key* has been deleted, any **destructor** function that was associated with *key* is not called when a thread exits. It is the responsibility of the application to free any application storage for each of the threads using *key*.

The **pthread_key_delete()** function can be called from a destructor function.

RETURN VALUE

If successful, **pthread_key_create()** and **pthread_key_delete()** return zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

If any of the following occur, the **pthread_key_create()** function returns the corresponding error number:

[EINVAL] The value specified by *key* is invalid.

[EAGAIN] The necessary resources to create another thread-specific data key are not available, or the total number of keys per process has exceeded **PTHREAD_KEYS_MAX**.

[ENOMEM] There is insufficient memory available in which to create *key*.

For each of the following conditions, if the condition is detected, the **pthread_key_delete()** function returns the corresponding error number:

[EINVAL] The value specified by *key* is invalid.

AUTHOR

pthread_key_create() and **pthread_key_delete()** were derived from the IEEE POSIX P1003.1c standard.

SEE ALSO

pthread_getspecific(3T), pthread_setspecific(3T).

STANDARDS CONFORMANCE

pthread_key_create(): POSIX 1003.1c.
pthread_key_delete(): POSIX 1003.1c.

NAME

pthread_kill() - send a signal to a thread.

SYNOPSIS

```
#include <signal.h>

int pthread_kill(
   pthread_t thread,
   int sig
);
```

PARAMETERS

thread Thread to which the signal is to be delivered.

sig Signal to be delivered to *thread*.

DESCRIPTION

The **pthread_kill()** function is used to request that a signal be delivered to *thread*. The signal is asynchronously directed to *thread* in the calling process. The signal is handled in the context of the given thread; if the signal action results in the thread terminating or stopping, this action is applied to the whole process.

If *sig* is zero, error checking is performed but a signal is not sent.

RETURN VALUE

Upon successful completion, **pthread_kill()** returns zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

If any of the following occur, the **pthread_kill()** function returns the corresponding error number:

[EINVAL] *sig* is an invalid or unsupported signal number.

[ESRCH] No thread could be found corresponding to *thread*.

AUTHOR

pthread_kill() was derived from the IEEE POSIX P1003.1c standard.

SEE ALSO

kill(2), sigaction(2), pthread_self(3T), raise(2).

STANDARDS CONFORMANCE

pthread_kill(): POSIX 1003.1c.

NAME

pthread_mutex_setprioceiling(), pthread_mutex_getprioceiling() - set or get the prioceiling of a mutex.

SYNOPSIS

```
#include <pthread.h>

int pthread_mutex_getprioceiling(
   pthread_mutex_t *mutex,
   int *prioceiling
);

int pthread_mutex_setprioceiling(
   pthread_mutex_t *mutex,
   int prioceiling,
   int *old_ceiling
);
```

PARAMETERS

mutex Pointer to the mutex whose *prioceiling* attribute is to be set/retrieved.

prioceiling
This parameter either points to the memory location where the *prioceiling* attribute of *mutex* is to be returned (get function) or specifies the new value of the *prioceiling* attribute for *mutex* (set function).

old_ceiling
This parameter points to the memory location where the old *prioceiling* attribute of *mutex* is to be returned (set function only).

DESCRIPTION

The **pthread_mutex_setprioceiling()** function will first lock *mutex*. If the mutex is currently locked, the calling thread will block until the mutex can be locked. Once the mutex has been locked, the *prioceiling* attribute of *mutex* will be changed to the value specified in the *prioceiling* parameter and *mutex* will be unlocked. The old priority ceiling for the mutex will be returned in *old_prioceiling*.

The **pthread_mutex_getprioceiling()** function returns the current value of the *prioceiling* attribute for *mutex* in the *prioceiling* parameter.

Be sure to check for the definition of **_POSIX_THREAD_PRIO_PROTECT** before using these functions. Not all systems will support these functions.

p

RETURN VALUE

Upon successful completion, **pthread_mutex_getprioceiling()** and **pthread_mutex_setprioceiling()** return zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

If any of the following occur, the **pthread_mutex_getprioceiling()** and **pthread_mutex_setprioceiling()** functions return the corresponding error number:

[ENOSYS] **_POSIX_THREAD_PRIO_PROTECT** is not defined and these functions are not supported.

For each of the following conditions, if the condition is detected, the **pthread_mutex_getprioceiling()** and **pthread_mutex_setprioceiling()** functions return the corresponding error number:

[EINVAL] The priority value *prioceiling* is not a legal value.

[EINVAL] *mutex* is not a valid mutex.

[ENOSYS] The prioceiling protocol is not supported for mutexes.

[EPERM] The caller does not have the appropriate privileges to change priority ceiling for *mutex*.

[EFAULT] *mutex* parameter points to an illegal address.

AUTHOR

pthread_mutex_getprioceiling() and **pthread_mutex_setprioceiling()** were derived from the IEEE POSIX P1003.1c standard.

SEE ALSO

pthread_create(3T), pthread_mutex_init(3T), pthread_mutexattr_setprioceiling(3T), pthread_mutexattr_getprioceiling(3T), pthread_mutex_lock(3T), pthread_mutex_trylock(3T), pthread_mutex_unlock(3T).

STANDARDS CONFORMANCE

pthread_mutex_getprioceiling(): POSIX 1003.1c.
pthread_mutex_setprioceiling(): POSIX 1003.1c.

p

NAME

pthread_mutex_init(), pthread_mutex_destroy() - initialize or destroy a mutex.

SYNOPSIS

```
#include <pthread.h>

int pthread_mutex_init(
   pthread_mutex_t *mutex,
   const pthread_mutexattr_t *attr
);

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_destroy(
   pthread_mutex_t *mutex
);
```

PARAMETERS

mutex Pointer to the mutex to be initialized or destroyed.

attr Pointer to the attributes object that defines the characteristics of the mutex to be initialized. If the pointer is **NULL**, default attributes are used.

DESCRIPTION

The **pthread_mutex_init()** function initializes the mutex referenced by *mutex* with the attributes *attr*. If the parameter *attr* is **NULL**, the default mutex attributes are used. Refer to *pthread_mutexattr_init*(3T) for a list of default mutex attributes. After successful initialization, the mutex is initialized, unlocked, and ready to be used in mutex operations. A mutex should be initialized only once or the resulting behavior is undefined. The **pthread_once()** function provides a way to ensure that a mutex is initialized only once.

The macro **PTHREAD_MUTEX_INITIALIZER** can be used to initialize mutexes that are statically allocated. These mutexes will be initialized with default attributes. The **pthread_mutex_init()** function does not need to be called for statically initialized mutexes.

If the *process-shared* attribute in the mutex attributes object referenced by *attr* is defined as **PTHREAD_PROCESS_SHARED**, the mutex must be allocated such that the processes sharing the mutex have access to it. This may be done through the memory-mapping functions (see *mmap*(2)) or the shared memory functions (see *shmget*(2)).

The **pthread_mutex_destroy()** function destroys the mutex referenced by *mutex*. This function may set *mutex* to an invalid value. The destroyed mutex can be reinitialized using the function **pthread_mutex_init()**. If the mutex is used after destruction in any mutex call, the resulting behavior is undefined.

A mutex should be destroyed only when it is unlocked. Destroying a mutex that is currently being used results in undefined behavior.

RETURN VALUE

Upon successful completion, **pthread_mutex_init()** and **pthread_mutex_destroy()** return zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

If any of the following occur, the **pthread_mutex_init()** function returns the corresponding error number:

[EAGAIN] The necessary resources (other than memory) to initialize *mutex* were not available.

[ENOMEM] There is insufficient memory available in which to initialize *mutex*.

[EPERM] The caller does not have the necessary permission to perform the mutex initialization.

For each of the following conditions, if the condition is detected, the **pthread_mutex_init()** function returns the corresponding error number:

[EINVAL] The value specified by *mutex* or *attr* is invalid.

[EBUSY] *mutex* is an already initialized mutex.

[EFAULT] *mutex* parameter points to an illegal address.

For each of the following conditions, if the condition is detected, the **pthread_mutex_destroy()** function returns the corresponding error number:

[EINVAL] *mutex* is not a valid mutex.

[EBUSY] *mutex* is currently locked or in use by another thread.

WARNINGS

The space for the mutex must be allocated before calling **pthread_mutex_init()**. Undefined behavior will result if the *process-shared* attribute of *attr* is **PTHREAD_PROCESS_SHARED** and the space allocated for the mutex is not accessible to cooperating threads.

AUTHOR

pthread_mutex_init() and **pthread_mutex_destroy()** were derived from the IEEE POSIX P1003.1c standard.

SEE ALSO

pthread_mutex_lock(3T), pthread_mutex_unlock(3T), pthread_mutex_trylock(3T).

STANDARDS CONFORMANCE

pthread_mutex_init(): POSIX 1003.1c.
pthread_mutex_destroy(): POSIX 1003.1c.

p

NAME

pthread_mutex_lock(), pthread_mutex_trylock() - lock or attempt to lock a mutex.

SYNOPSIS

```
#include <pthread.h>
int pthread_mutex_lock(
   pthread_mutex_t *mutex
);
int pthread_mutex_trylock(
   pthread_mutex_t *mutex
);
```

PARAMETERS

mutex Pointer to the mutex to be locked.

DESCRIPTION

The mutex object *mutex* is locked by calling the **pthread_mutex_lock()** function. How the calling thread acquires the mutex is dependent upon the *type* attribute for the mutex. This operation returns with the mutex object referenced by *mutex* in the locked state with the calling thread as its owner.

If the mutex *type* is **PTHREAD_MUTEX_NORMAL**, deadlock detection is not provided. Attempting to relock the mutex causes deadlock. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, undefined behavior results.

If the mutex *type* is **PTHREAD_MUTEX_ERRORCHECK**, the mutex maintains the concept of an owner. If a thread attempts to relock a mutex that it has already locked, an error shall be returned. If a thread attempts to unlock a mutex that it has not locked or a mutex that is unlocked, an error shall be returned.

If the mutex *type* is **PTHREAD_MUTEX_RECURSIVE**, then the mutex maintains the concept of an owner and a lock count. When a thread successfully acquires a mutex for the first time, the count field shall be set to one. Every time a thread relocks this mutex, the count field shall be incremented by one. Each time the thread unlocks the mutex, the count field shall be decremented by one. When the count field reaches zero, the mutex shall become available for other threads to acquire. If a thread attempts to unlock a mutex that it has not locked, an error shall be returned.

If the mutex *type* is **PTHREAD_MUTEX_DEFAULT**, attempting to recursively lock the mutex results in undefined behavior. Attempting to unlock the mutex if it was not locked by the calling thread results in undefined behavior. Attempting to unlocked the mutex if it is not locked results in undefined behavior.

The function **pthread_mutex_trylock()** is identical to the **pthread_mutex_lock()** function except that if the mutex object referenced by *mutex* cannot be acquired after one attempt, the call returns immediately with an error.

If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler, the thread shall resume waiting for the mutex as if it was not interrupted.

RETURN VALUE

Upon successful completion, **pthread_mutex_lock()** and **pthread_mutex_trylock()** return zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

If any of the following occur, the **pthread_mutex_lock()** and **pthread_mutex_trylock()** functions return the corresponding error number:

[EINVAL] *mutex* is a **PTHREAD_PRIO_PROTECT** mutex and the caller's priority is higher than *mutex*'s priority ceiling.

If any of the following occur, the **pthread_mutex_trylock()** function returns the corresponding error number:

[EBUSY] *mutex* is currently locked by another thread.

If any of the following occur, the **pthread_mutex_lock()** and **pthread_mutex_trylock()** functions return the corresponding error number:

[EAGAIN] *mutex* could not be acquired because the maximum number of recursive locks for *mutex* has been exceeded. This error is not detected on HP-UX.

For each of the following conditions, if the condition is detected, the **pthread_mutex_lock()** and **pthread_mutex_trylock()** functions return the corresponding error number:

[EINVAL] *mutex* is not an initialized mutex.

[EFAULT] *mutex* parameter points to an illegal address.

[EDEADLK] The current thread already owns the mutex. This error will be detected only for **PTHREAD_MUTEX_ERRORCHECK** mutexes on HP-UX.

WARNINGS

A recursive mutex can be locked more than once by the same thread without causing that thread to deadlock. Undefined behavior may result if the owner of a recursive mutex tries to lock the mutex too many times.

AUTHOR

pthread_mutex_lock() and **pthread_mutex_trylock()** was derived from the IEEE POSIX P1003.1c standard and X/Open

SEE ALSO

pthread_mutex_init(3T), pthread_mutex_destroy(3T), pthread_mutex_unlock(3T).

STANDARDS CONFORMANCE

pthread_mutex_lock(): POSIX 1003.1c.
pthread_mutex_trylock(): POSIX 1003.1c.

p

NAME

pthread_mutex_unlock() - unlock a mutex.

SYNOPSIS

```
#include <pthread.h>

int pthread_mutex_unlock(
   pthread_mutex_t *mutex
);
```

PARAMETERS

mutex Pointer to the mutex to be unlocked.

DESCRIPTION

The function **pthread_mutex_unlock()** is called by the owner of the mutex referenced by *mutex* to unlock the mutex. The manner in that the mutex is released is dependent upon the mutex's *type* attribute. For normal and default mutexes, undefined behavior will result if **pthread_mutex_unlock()** is called on an unlocked mutex or by a thread that is not the current owner. For recursive and error-checking mutexes, an error is returned if **pthread_mutex_unlock()** is called on an unlocked mutex or by a thread which is not the current owner.

For recursive mutexes, the owner must call **pthread_mutex_unlock()** as many times as the mutex was locked before another thread can lock the mutex.

If there are threads blocked on the mutex referenced by *mutex* when **pthread_mutex_unlock()** releases the mutex, the scheduling policy is used to determine which thread will acquire the mutex next.

RETURN VALUE

Upon successful completion, **pthread_mutex_unlock()** returns zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

For each of the following conditions, if the condition is detected, the **pthread_mutex_unlock()** function returns the corresponding error number:

[EINVAL] *mutex* is not an initialized mutex.

[EPERM] The calling thread does not own *mutex*. On HP-UX, this error is not detected for **PTHREAD_MUTEX_FAST** or **PTHREAD_MUTEX_DEFAULT** mutexes.

[EFAULT] *mutex* parameter points to an illegal address.

AUTHOR

pthread_mutex_unlock() was derived from the IEEE POSIX P1003.1c standard and HP extensions.

SEE ALSO

pthread_mutex_init(3T), pthread_mutex_destroy(3T), pthread_mutex_lock(3T), pthread_mutex_trylock(3T).

STANDARDS CONFORMANCE

pthread_mutex_unlock(): POSIX 1003.1c.

NAME

pthread_mutexattr_getprioceiling(), pthread_mutexattr_setprioceiling(), pthread_mutexattr_getprotocol(), pthread_mutexattr_setprotocol() - get and set the prioceiling and protocol attributes

SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
    int prioceiling);

int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *attr,
    int *prioceiling);

int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
    int protocol);

int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *attr,
    int *protocol);
```

PARAMETERS

attr Pointer to the mutex attributes object whose attributes are to be set/retrieved.

prioceiling
This parameter either specifies the new value of the *prioceiling* attribute (set function) or points to the memory location where the *prioceiling* attribute of *attr* is to be returned (get function).

protocol This parameter either specifies the new value of the *protocol* attribute (set function) or points to the memory location where the *protocol* attribute of *attr* is to be returned (get function).

DESCRIPTION

Be sure to check for the definitions of **_POSIX_THREAD_PRIO_PROTECT** and **_POSIX_THREAD_PRIO_INHERIT** before using these functions. Not all systems will support these functions.

The attributes object *attr* must have previously been initialized with the function **pthread_mutexattr_init()** before these functions are called.

ATTRIBUTE: protocol

Mutexes can be initialized with a priority protocol to help avoid or minimize the priority inversion that can be caused by locked mutexes. The *protocol* attribute in a mutex attributes object describes the priority protocol to be used when the mutex is locked by a thread. The legal values for the *protocol* attribute are:

PTHREAD_PRIO_NONE
A thread's scheduling priority is not changed when it locks this type of mutex.

PTHREAD_PRIO_PROTECT
These types of mutexes have an associated priority value in the *prioceiling* attribute. When a thread locks a mutex of this type, its scheduling priority will be changed to be the value contained in the *prioceiling* attribute. The value of *prioceiling* must be higher than the locking thread's scheduling priority. When the mutex is unlocked, the thread's previous scheduling priority will be restored.

If a thread owns several mutexes of this type, its scheduling priority will be changed to the higher of all the *prioceiling* attributes for all mutexes of this type that it owns.

PTHREAD_PRIO_INHERIT
When a thread must block waiting for a mutex of this type, the system will change the scheduling priority of the thread that owns the mutex to be the higher of its own priority or the priority of the highest priority thread blocked on the mutex. When the mutex is unlocked, the thread's previous scheduling priority will be restored.

If a thread owns one or more mutexes having the *protocol* attribute value of **PTHREAD_PRIO_PROTECT** or **PTHREAD_PRIO_INHERIT**, the thread will not be moved to the tail of its priority list if its original priority is changed or when it unlocks the mutex(es).

If a thread owns mutexes of different priority protocols, it will execute at the highest of the priorities that would be obtained by each of these protocols. If this thread becomes blocked on another mutex, the priority behavior is recursive and is passed on to the thread that owns the mutex on which this thread is blocked.

POSIX.1c does not define a default value for the *protocol* attribute. On HP-UX, the default value is **PTHREAD_PRIO_NONE**.

pthread_mutexattr_setprotocol() is used to set the *protocol* attribute in the initialized attributes object *attr*. The new value of the *protocol* attribute of *attr* is set to the value specified in the *protocol* parameter.

pthread_mutexattr_getprotocol() retrieves the value of the *protocol* attribute from the mutex attributes object *attr*. The value of the *protocol* attribute of *attr* is returned in the *protocol* parameter.

ATTRIBUTE: prioceiling

If the *protocol* attribute has a value of **PTHREAD_PRIO_PROTECT**, the *prioceiling* attribute indicates the priority ceiling of the mutex. Otherwise, this attribute is not used when the mutex is initialized.

The priority values that are valid for this attribute are the same values that are valid for the **SCHED_FIFO** scheduling policy.

POSIX.1c does not define a default value for the *prioceiling* attribute. On HP-UX, the default value is the minimum priority value for the **SCHED_FIFO** scheduling policy.

pthread_mutexattr_setprioceiling() is used to set the *prioceiling* attribute in the initialized attributes object *attr*. The new value of the *prioceiling* attribute of *attr* is set to the value specified in the *prioceiling* parameter.

pthread_mutexattr_getprioceiling() retrieves the value of the *prioceiling* attribute from the mutex attributes object *attr*. The value of the *prioceiling* attribute of *attr* is returned in the *prioceiling* parameter.

RETURN VALUE

Upon successful completion, **pthread_mutexattr_setprioceiling()**, **pthread_mutexattr_getprioceiling()**, **pthread_mutexattr_setprotocol()**, and **pthread_mutexattr_getprotocol()** return zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

If any of the following occur, the **pthread_mutexattr_setprioceiling()**, **pthread_mutexattr_getprioceiling()**, **pthread_mutexattr_setprotocol()**, and **pthread_mutexattr_getprotocol()** functions return the corresponding error number:

[ENOSYS] **_POSIX_THREAD_PRIO_PROTECT** is not defined and these functions are not supported.

[ENOTSUP] *protocol* contains an unsupported value.

For each of the following conditions, if the condition is detected, the **pthread_mutexattr_setprioceiling()**, **pthread_mutexattr_getprioceiling()**, **pthread_mutexattr_setprotocol()**, and **pthread_mutexattr_getprotocol()** functions return the corresponding error number:

[EINVAL] The value specified by *attr*, *prioceiling*, or *protocol* is invalid.

[EPERM] The caller does not have the appropriate privilege to set the priority ceiling or priority protocol to the specified values.

AUTHOR

pthread_mutexattr_setprioceiling(), **pthread_mutexattr_getprioceiling()**, **pthread_mutexattr_setprotocol()**, and **pthread_mutexattr_getprotocol()** were derived from the IEEE POSIX P1003.1c standard.

SEE ALSO

pthread_create(3T), pthread_mutexattr_init(3T), pthread_mutex_init(3T).

STANDARDS CONFORMANCE

pthread_mutexattr_setprioceiling(): POSIX 1003.1c.
pthread_mutexattr_getprioceiling(): POSIX 1003.1c.
pthread_mutexattr_setprotocol(): POSIX 1003.1c.
pthread_mutexattr_getprotocol(): POSIX 1003.1c.

NAME

pthread_mutexattr_getpshared(), pthread_mutexattr_setpshared(), pthread_mutexattr_gettype(), pthread_mutexattr_settype() - get and set the process-shared attribute and type attribute

SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
    int pshared);

int pthread_mutexattr_getpshared(const pthread_mutexattr_t *attr,
    int *pshared);

int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);

int pthread_mutexattr_gettype(const pthread_mutexattr_t *attr,
    int *type);
```

PARAMETERS

attr Pointer to the mutex attributes object whose attributes are to be set/retrieved.

pshared This parameter either specifies the new value of the *process-shared* attribute (set function) or points to the memory location where the *process-shared* attribute of *attr* is to be returned (get function).

type This parameter either specifies the new value of the *type* attribute (set function) or points to the memory location where the *type* attribute of *attr* is to be returned (get function).

DESCRIPTION

The attributes object *attr* must have been previously initialized with the function **pthread_mutexattr_init()** before these functions are called.

ATTRIBUTE: pshared

Mutexes can be used by threads only within a process or shared by threads in multiple processes. The *process-shared* attribute in a mutex attributes object describes who may use the mutex. The legal values for the *process-shared* attribute are:

PTHREAD_PROCESS_SHARED
This option permits a mutex to be operated upon by any thread that has access to the memory where the mutex is allocated. The application is responsible for allocating the mutex in memory that multiple processes can access.

PTHREAD_PROCESS_PRIVATE
The mutex can be operated upon only by threads created within the same process as the thread that initialized the mutex. If threads of differing processes attempt to operate on such mutex, the behavior is undefined.

The default value of *process-shared* is **PTHREAD_PROCESS_PRIVATE**.

pthread_mutexattr_setpshared() is used to set the *process-shared* attribute in *attr*. The new value of the *process-shared* attribute of *attr* is set to the value specified in the *pshared* parameter.

pthread_mutexattr_getpshared() retrieves the value of the *process-shared* attribute from *attr*. The value of the *process-shared* attribute of *attr* is returned in the *pshared* parameter.

ATTRIBUTE: type

Mutexes can be created with four different types. The type of a mutex is contained in the *type* attribute of the mutex attributes object. Valid values for the *type* attribute are:

PTHREAD_MUTEX_NORMAL
This type of mutex does not provide deadlock detection. A thread attempting to relock this mutex without first unlocking it shall deadlock. An error is not returned to the caller. Attempting to unlock a mutex locked by a different thread results in undefined behavior. Attempting to unlock an unlocked mutex results in undefined behavior.

PTHREAD_MUTEX_ERRORCHECK
This type of mutex provides error checking. An owner field is maintained. Only the mutex lock owner shall successfully unlock this mutex. A thread attempting to relock this mutex shall return with an error. A thread attempting to unlock a mutex locked by a different thread shall return with an error. A thread attempting to unlock an unlocked mutex shall return with an

error. This type of mutex is useful for debugging.

PTHREAD_MUTEX_RECURSIVE
Deadlock cannot occur with this type of mutex. An owner field is maintained. A thread attempting to relock this mutex shall successfully lock the mutex. Multiple locks of this mutex shall require the same number of unlocks to release the mutex before another thread can lock the mutex. A thread attempting to unlock a mutex locked by a different thread shall return with an error. A thread attempting to unlock an unlocked mutex shall return with an error.

PTHREAD_MUTEX_DEFAULT
Attempting to recursively lock a mutex of this type results in undefined behavior. Attempting to unlock a mutex locked by a different thread results in undefined behavior. Attempting to unlock an unlocked mutex results in undefined behavior. An implementation shall be allowed to map this mutex to one of the other mutex types.

The default value of the *type* attribute is **PTHREAD_MUTEX_DEAFULT**.

pthread_mutexattr_settype() is used to set the *type* attribute in *attr*. The new value of the *type* attribute of *attr* is set to the value specified in the *type* parameter.

pthread_mutexattr_gettype() retrieves the value of the *type* attribute from *attr*. The value of the *type* attribute of *attr* is returned in the *type* parameter.

Never use a **PTHREAD_MUTEX_RECURSIVE** mutex with condition variables because the implicit unlock performed for a **pthread_cond_wait()** or **pthread_cond_timedwait()** may not actually release the mutex if it had been locked multiple times. If this situation happens, no other thread can satisfy the condition of the predicate.

RETURN VALUE

Upon successful completion, **pthread_mutexattr_getpshared()**, **pthread_mutexattr_setpshared()**, **pthread_mutexattr_gettype()**, and **pthread_mutexattr_settype()** return zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

If any of the following occur, the **pthread_mutexattr_getpshared()** and **pthread_mutexattr_setpshared()** functions return the corresponding error number:

[ENOSYS] **_POSIX_THREAD_PROCESS_SHARED** is not defined and these functions are not supported.

For each of the following conditions, if the condition is detected, the **pthread_mutexattr_getpshared()**, **pthread_mutexattr_setpshared()**, **pthread_mutexattr_gettype()**, and **pthread_mutexattr_settype()** functions return the corresponding error number:

[EINVAL] *attr* is not a valid mutex attributes object.

[EINVAL] The value specified by *pshared* or *type* is not a legal value. [EINVAL] The value *pshared* or *type* points to an illegal address.

WARNINGS

If a mutex is created with the *process-shared* attribute defined as **PTHREAD_PROCESS_SHARED**, the cooperating processes should have access to the memory in which the mutex is allocated.

AUTHOR

pthread_mutexattr_setpshared() and **pthread_mutexattr_getpshared()** were derived from the IEEE POSIX P1003.1c standard.

pthread_mutexattr_settype(), and **pthread_mutexattr_gettype()** were developed by X/Open.

SEE ALSO

pthread_create(3T), pthread_mutexattr_init(3T), pthread_mutex_init(3T).

STANDARDS CONFORMANCE

pthread_mutexattr_setpshared(): POSIX 1003.1c.
pthread_mutexattr_getpshared(): POSIX 1003.1c.
pthread_mutexattr_settype(): X/Open.
pthread_mutexattr_gettype(): X/Open.

p

NAME

pthread_mutexattr_getspin_np(), pthread_mutexattr_setspin_np(), pthread_mutex_getyieldfreq_np(), pthread_mutex_setyieldfreq_np() - get and set the mutex spin and yield frequency attributes

SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_setspin_np(pthread_mutexattr_t *attr, int spin);

int pthread_mutexattr_getspin_np(const pthread_mutexattr_t *attr,
    int *spin);

int pthread_mutex_setyieldfreq_np(int yield);

int pthread_mutex_getyieldfreq_np(int *yield);
```

PARAMETERS

attr Pointer to the mutex attributes object whose attributes are to be set/retrieved.

spin This parameter either specifies the new value of the *spin* attribute (set function) or points to the memory location where the *spin* attribute of *attr* is to be returned (get function).

yield This parameter either specifies the new value of the *yield frequency* process-wide attribute (set function) or points to the memory location where the *yield frequency* process-wide attribute is to be returned (get function).

DESCRIPTION

These attributes are used to tune the mutex locking behavior for optimized application performance on multiprocessor systems.

For **pthread_mutexattr_setspin_np()** and **pthread_mutexattr_getspin_np()** the attributes object *attr* must have previously been initialized with the function **pthread_mutexattr_init()** before these functions are called.

ATTRIBUTE: spin

Mutexes can be initialized with a spin value which is used by **pthread_mutex_lock()** for busy-wait iterations on the mutex lock. It is effective only on multiprocessor systems. It is ignored on uniprocessor systems.

For bound threads, the procedure to block on a busy mutex is quite costly. On a multiprocessor system, the thread holding the mutex may be a few instruction cycles away from releasing it. By performing a brief busy-wait before actually blocking, the lock path can avoid a great deal of overhead in these situations.

For many applications, these situations are not rare. Consider a mutex used to protect a short sequence of code (e.g., to increment a counter). The mutex is held only for a few instruction cycles (plus the function call overhead for *pthread_mutex_unlock()*.

The ability to set the *spin* attribute allows the application writer to adjust the busy-wait to suit the duration of the common-case critical section protected by each mutex.

The legal values for the *spin* attribute are:

positive integer
The **pthread_mutex_lock()** function will busy-wait on the mutex lock for the specified number of iterations before blocking the thread, unless the lock is acquired sooner. Larger *spin* values would be appropriate for mutexes associated with longer critical sections.

PTHREAD_MUTEX_SPINONLY_NP
Inhibits blocking on the mutex lock altogether. The **pthread_mutex_lock()** function will busy-wait on the mutex lock until it is acquired.

PTHREAD_MUTEX_SPINDEFAULT_NP
Uses a built-in default value for the number of busy-wait iterations.

ATTRIBUTE: yield

Note that the *yield* attribute is not a per-mutex attribute, but is process-wide. That is, it affects the behavior of all mutexes.

The *yield* attribute specifies, for the **pthread_mutex_lock()** busy-wait, how frequently the processor should be yielded (via **sched_yield()**), allowing other threads to execute.

When the number of threads exceeds the number of processors in a system, a busy-wait on a mutex can sometimes have an adverse effect. The busy-wait itself can prevent the thread holding the lock from completing the associated critical section of code. By yielding the processor on occasion, the thread attempting the lock may allow the thread holding the lock to reach the point at which it can release the lock. However, it still avoids the costly path to block on the mutex.

The legal values for the *yield* attribute are:

positive integer
The busy-wait loop in **pthread_mutex_lock()** will yield the processor after each specified number of iterations of the spin loop (where the total number of iterations is controlled by the per-mutex *spin* attribute). **PTHREAD_MUTEX_YIELDNEVER_NP** Inhibits yielding in the mutex lock altogether.

PTHREAD_MUTEX_YIELDFREQDEFAULT_NP
Uses a built-in default value for the frequency of yields in the busy-wait loop.

WARNINGS

The setting of the *spin* and *yield* attributes can, as well as improve application performance, easily lead to degraded performance. The CPU consumption of the application may be increased. Settings which work well for a small number of threads may do poorly for larger numbers of threads. The optimal settings will vary depending upon hardware and operating system configuration. Minor changes in the application itself may require retuning of these attributes.

The programmer must analyze performance carefully, to obtain an understanding of mutex contention within the application. Then, experiment with different attribute values, evaluating how mutex contention is affected, response time, and CPU consumption.

RETURN VALUE

Upon successful completion, **pthread_mutexattr_setspin_np()**, **pthread_mutexattr_getspin_np()**, **pthread_mutex_setyieldfreq_np()**, and **pthread_mutex_getyieldfreq_np()** return zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

For each of the following conditions, if the condition is detected, the **pthread_mutexattr_setspin_np()**, **pthread_mutexattr_getspin_np()**, **pthread_mutex_setyieldfreq_np()**, and **pthread_mutex_getyieldfreq_np()** functions return the corresponding error number:

[EINVAL] The value specified by *attr*, *spin*, or *yield* is invalid.

AUTHOR

pthread_mutexattr_getspin_np(), **pthread_mutexattr_setspin_np()**, **pthread_mutex_getyieldfreq_np()**, and **pthread_mutex_setyieldfreq_np()** were developed by HP.

SEE ALSO

pthread_create(3T), pthread_mutexattr_init(3T), pthread_mutex_init(3T), rtsched(2)

STANDARDS CONFORMANCE

pthread_mutexattr_setspin_np(): None.
pthread_mutexattr_getspin_np(): None.
pthread_mutex_setyieldfreq_np(): None.
pthread_mutex_getyieldfreq_np(): None.

NAME

pthread_mutexattr_init(), pthread_mutexattr_destroy() - initialize or destroy a mutex attributes object.

SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_init(
   pthread_mutexattr_t *attr
);

int pthread_mutexattr_destroy(
   pthread_mutexattr_t *attr
);
```

PARAMETERS

attr Pointer to the mutex attributes object to be initialized or destroyed.

DESCRIPTION

pthread_mutexattr_init() initializes the mutex attributes object *attr* with the default values for all attributes. The attributes object describes a mutex in detail and is passed to the mutex initialization function.

When a mutex attributes object is used to initialize a mutex, the values of the individual attributes determine the characteristics of the new mutex. Attributes objects act like additional parameters to object initialization. A single attributes object can be used in multiple calls to the function **pthread_mutex_init()**.

When a mutex is initialized with an attributes object, the attributes are, in effect, copied into the mutex. Consequently, any change to the attributes object will not affect any previously initialized mutexes. Once all mutexes needing a specific attributes object have been initialized, the attributes object is no longer needed.

The mutex attributes and their default values are:

process-shared The default value is **PTHREAD_PROCESS_PRIVATE**.

type The default value is **PTHREAD_MUTEX_DEAFULT**.

If an initialized mutex attributes object is reinitialized, undefined behavior results.

pthread_mutexattr_destroy() destroys the mutex attributes object *attr*. The destroyed mutex attributes object ceases to exist and its resources are reclaimed. Using *attr* after it has been destroyed results in undefined behavior. A destroyed mutex attributes object can be reinitialized using the **pthread_mutexattr_init()** function.

Mutexes that have been already initialized using this attributes object are not affected by the destruction of the mutex attributes object.

RETURN VALUE

Upon successful completion, **pthread_mutexattr_init()** and **pthread_mutexattr_destroy()** return zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

For each of the following conditions, if the condition is detected, the **pthread_mutexattr_init()** and **pthread_mutexattr_destroy()** functions return the corresponding error number:

[ENOMEM] There is insufficient memory available in which to initialize *attr*.

[EINVAL] The value specified by *attr* is invalid.

AUTHOR

pthread_mutexattr_init() and **pthread_mutexattr_destroy()** were derived from the IEEE POSIX P1003.1c standard.

SEE ALSO

pthread_create(3T), pthread_mutexattr_getpshared(3T), pthread_mutexattr_setpshared(3T), pthread_mutexattr_gettype(3T), pthread_mutexattr_settype(3T), pthread_mutex_init(3T).

STANDARDS CONFORMANCE

pthread_mutexattr_init(): POSIX 1003.1c.
pthread_mutexattr_destroy(): POSIX 1003.1c.

NAME

pthread_once() - call an initialization routine only once.

SYNOPSIS

```
#include <pthread.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;

int pthread_once(
   pthread_once_t *once_control,
   void (*init_routine)(void)
);
```

PARAMETERS

once_control

Pointer to the once-control object associated with the one-time initialization function **init_routine()**.

init_routine

The one-time initialization routine. This routine is called only once, regardless of the number of times it and its associated *once_control* are passed to **pthread_once()**.

DESCRIPTION

The **pthread_once()** function guarantees that **init_routine()** is only called one time in an application. This function will use the *once_control* object to determine if **init_routine()** has previously been called via **pthread_once()**.

The first time **pthread_once()** is called with *once_control* and **init_routine()** causes **init_routine()** to be called with no arguments. Subsequent calls to **pthread_once()** with the same *once_control* will not cause **init_routine()** to be called again. When **pthread_once()** returns, the caller is guaranteed that **init_routine()** has been called (either just now or via a previous call).

The macro **PTHREAD_ONCE_INIT** is used to statically initialize a once control block. This initialization must be done before calling **pthread_once()**.

pthread_once() is not a cancellation point. However, the caller supplied **init_routine()** may be a cancellation point. If the thread executing **init_routine()** is canceled, the *once_control* argument will be set to a state which indicates that **init_routine()** has not been called yet (see *pthread_cancel*(3T)). The next time the **pthread_once()** function is called with *once_control*, the **init_routine()** function will be called.

The behavior of **pthread_once()** is undefined if *once_control* has automatic storage duration or is not initialized by **PTHREAD_ONCE_INIT**.

RETURN VALUE

Upon successful completion, **pthread_once()** returns zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

For each of the following conditions, if the condition is detected, the **pthread_once()** function returns the corresponding error number:

[EINVAL] Either *once_control* or *init_routine* is invalid.

EXAMPLES

Some modules are designed for dynamic initialization, i.e., global initialization is performed when the first function of the module is invoked. In a single-threaded program, this is generally implemented as follows:

```
static int initialized = FALSE;
extern void initialize();

if (!initialized) {
     initialize();
     initialized = TRUE;
}
```

```
Rest of the code after initialization.
```

For a multithreaded process, a simple initialization flag is not sufficient; the flag must be protected against modification by multiple threads. Consequently, this flag has to be protected by a mutex that has to be initialized only once, and so on. A multithreaded program should use initialization similar to:

```
static pthread_once_t  once_control = PTHREAD_ONCE_INIT;
extern void initialize();

(void)pthread_once(&once_control, initialize);

Rest of the code after initialization.
```

AUTHOR

pthread_once() was derived from the IEEE POSIX P1003.1c standard.

SEE ALSO

pthread_create(3T).

STANDARDS CONFORMANCE

pthread_once(): POSIX 1003.1c.

p

NAME

pthread_num_processors_np(), pthread_processor_bind_np(), pthread_processor_id_np() - determine how many processors are installed in the system, bind threads to processors, and determine processor IDs; respectively.

SYNOPSIS

```
#include <pthread.h>

int pthread_num_processors_np(void);

int pthread_processor_bind_np(
   int request,
   pthread_spu_t *answer,
   pthread_spu_t spu,
   pthread_t tid
);

int pthread_processor_id_np(
   int request,
   pthread_spu_t *answer,
   pthread_spu_t spu
);
```

PARAMETERS

request This parameter determines the precise action to be taken by these functions.

answer This parameter is an output parameter in which values are returned. The meaning of *answer* depends on *request* parameter.

spu This parameter gives the value of the spu for certain requests.

tid This parameter gives the value of the thread id for certain requests.

DESCRIPTION

These functions provide a means of determining how many processors are installed in the system and assigning threads to run on specific processors.

The **pthread_num_processors_np()** function returns the number of processors currently installed on the system.

The **pthread_processor_id_np()** function obtains the processor ID of a specific processor on the system. The processor ID is returned in *answer*. The *request* parameter determines the precise action to be taken and is one of the following:

PTHREAD_GETFIRSTSPU_NP
This request stores in the *answer* parameter the ID of the first processor in the system. The *spu* argument is ignored.

PTHREAD_GETNEXTSPU_NP
This request stores in the *answer* parameter the ID of the next processor in the system after *spu*. Typically, **PTHREAD_GETFIRSTSPU_NP** is called to determine the first spu. **PTHREAD_GETNEXTSPU_NP** is then called in a loop (until the call returns **EINVAL**) to determine the IDs of the remaining spus.

PTHREAD_GETCURRENTSPU_NP
This request stores in the *answer* parameter the ID of the processor the thread is currently running on. The *spu* argument is ignored. Note: This option returns the current processor on which the caller is executing, NOT the processor assignment of the caller.

This information may be out-of-date arbitrarily soon after the call completes.

The **pthread_processor_bind_np()** function is expected to be used to increase performance in certain applications to prevent cache thrashing or to cause threads to execute in parallel on different processors. It should not be used to ensure correctness of an application. Specifically, cooperating threads should not rely on processor assignment in lieu of a synchronization mechanism (such as mutexes).

The **pthread_processor_bind_np()** function binds a thread to a specific processor. The thread specified by *tid* is the target thread whose binding is changed. The *spu* parameter specifies the new processor binding for *tid*. The *request* parameter determines the precise action to be taken and is one of the

following:

PTHREAD_BIND_ADVISORY_NP
This request assigns thread *tid* to processor *spu*. Since the new spu assignment is returned in the *answer* parameter, the spu **PTHREAD_SPUNOCHANGE_NP** may be passed to read the current assignment. The tid **PTHREAD_SELFTID_NP** can be used to refer to the calling thread. The value **PTHREAD_SPUFLOAT_NP** may be passed in the *spu* parameter to break any specific processor assignment and allow the implementation to choose which processor the thread should execute on when it is scheduled to execute. This allows the thread to run on any processor the implementation chooses.

This request is only advisory. If the scheduling policy for the thread conflicts with this processor assignment, the scheduling policy shall overrule the processor assignment. For example, when a processor is ready to choose another thread to execute, if the highest priority **SCHED_FIFO** thread on the run queue is bound to a different processor, that thread will execute on the available processor rather than waiting for the processor to which it is bound.

PTHREAD_BIND_FORCED_NP
This request is identical to **PTHREAD_BIND_ADVISORY_NP** except that this thread to processor binding will over rule the scheduling policy. For example, when a processor is ready to choose another thread to execute, if the highest priority **SCHED_FIFO** thread on the run queue is bound to a different processor, that thread will not be chosen by the available processor. That thread will wait until the *wanted* processor becomes available. The available processor will choose a lower priority thread to execute instead of completely honoring the scheduling policies.

Note: binding a thread to a specific processor essentially changes the scheduling allocation domain size for that thread to be one. Having a thread float and be scheduled on whatever processor the system chooses sets a thread's scheduling allocation domain size to a value greater than one (it will generally be equal to the number of processors on the system).

RETURN VALUE

pthread_num_processors_np() always returns the number of processors on the system. It never fails.

Upon successful completion, **pthread_processor_id_np()** returns zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set). Upon successful completion, **pthread_processor_bind_np()** returns zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

p

ERRORS

If any of the following occur, the **pthread_processor_id_np()** and **pthread_processor_bind_np()** functions return the corresponding error number:

[EINVAL] The *request* parameter contains an illegal value.

[EINVAL] The *request* parameter is **PTHREAD_GETNEXTSPU_NP** and *spu* identifies the last processor.

[EINVAL] The value specified by *answer* is illegal.

[ESRCH] No thread could be found in the current process that matches the thread ID specified in *tid*.

[EPERM] *request* is **PTHREAD_BIND_ADVISORY_NP** or **PTHREAD_BIND_FORCED_NP**, *spu* is not **PTHREAD_SPUNOCHANGE_NP**, and the caller does not have the appropriate permission to change a threads binding to a specific processor.

AUTHOR

pthread_num_processors_np(), **pthread_processor_id_np()**, and **pthread_num_processor_bind_np()** were developed by HP.

SEE ALSO

sleep(3C), rtsched(2).

STANDARDS CONFORMANCE

pthread_num_processors_np(): None.
pthread_processor_id_np(): None.
pthread_processor_bind_np(): None.

NAME

pthread_continue(), pthread_resume_np(), pthread_suspend() - continue execution of a thread, resume execution of a thread, and suspend execution of a thread; respectively.

SYNOPSIS

```
#include <pthread.h>

int pthread_continue(
   pthread_t thread
);

int pthread_resume_np(
   pthread_t thread,
   int flags
);

int pthread_suspend(
   pthread_t thread
);
```

PARAMETERS

thread whose execution is to be suspended or resumed.

flags Flags to be used by **pthread_resume_np().** The valid values are:

PTHREAD_COUNT_RESUME_NP
The target thread's suspension count is decremented by one. If the target thread was suspended and has a suspend count greater than one, the thread will not resume execution.

PTHREAD_FORCE_RESUME_NP
The target thread's suspension count is set to zero. The target will resume execution even if its suspend count was greater than one.

DESCRIPTION

The **pthread_suspend()** function suspends execution of the target thread specified by *thread*. The target thread may not be suspended immediately (at that exact instant). On successful return from the **pthread_suspend()** function, *thread* is no longer executing. Once a thread has been suspended, subsequent calls to the **pthread_suspend()** function increment a per thread suspension count and return immediately.

Calling **pthread_suspend()** with the calling thread specified in *thread* is allowed. Note that in this case the calling thread shall be suspended during execution of the **pthread_suspend()** function call and shall only return after another thread has called the **pthread_resume_np()** or **pthread_continue()** function for *thread*.

The **pthread_continue()** function resumes the execution of the target thread *thread*. If *thread* was suspended by multiple calls to **pthread_suspend()**, only one call to **pthread_continue()** is required to resume the execution of *thread*. Calling **pthread_continue()** for a target thread that is not suspended shall have no effect and return no errors. Calling **pthread_continue()** is equivalent to calling **pthread_resume_np()** with the *flags* parameter specified as **PTHREAD_FORCE_RESUME_NP**.

The **pthread_resume_np()** function resumes the execution of the target thread specified by *thread*. If the *flags* argument is **PTHREAD_COUNT_RESUME_NP**, the target thread's suspension count is decremented by one. If the *flags* argument is **PTHREAD_FORCE_RESUME_NP,** the target thread's suspension count is set to zero. When the target thread's suspension count reaches zero, the target thread is allowed to continue execution. Calling **pthread_resume_np()** for a target thread that is not suspended shall have no effect and return no errors.

RETURN VALUE

If successful, **pthread_continue()**, **pthread_suspend()** and **pthread_resume_np()** return zero. Otherwise, an error number shall be returned to indicate the error (the **errno** variable is not set).

ERRORS

If any of the following occur, the **pthread_suspend()** function returns the corresponding error number.

[ESRCH] The target thread *thread* is not in the current process.

[EDEADLK] The target thread *thread* is the last running thread in the process. The operation would result in deadlock for the process.

If any of the following occur, the **pthread_continue()** and **pthread_resume_np()** functions return the corresponding error number.

[ESRCH] The target thread *thread* is not in the current process.

[EINVAL] The value specified by *flags* is invalid.

APPLICATION USAGE

This functionality enables a process that is multithreaded to temporarily suspend all activity to a single thread of control. When the process is single threaded, the address space is not changing, and a consistent view of the process can be gathered. One example of its use is for garbage collecting. The garbage collector runs asynchronously within the process and assumes that the process is not changing while it is running.

Suspending a thread may have adverse effects on an application. If a thread is suspended while it holds a critical resource, such as a mutex or a read-write lock, the application may stop or even deadlock until the thread is continued. While the thread is suspended, other threads which may contend for the same resource must block until the thread is continued. Depending on application behavior, this may even result in deadlock. Application programmers are advised to either a) only suspend threads which call async-signal safe functions or b) ensure that the suspending thread does not contend for the same resources that the suspended thread may have acquired. Note: this includes resources that may be acquired by libraries.

The **pthread_suspend()**, **pthread_continue()**, and **pthread_resume_np()** functions cannot reliably be used for thread synchronization. Synchronization primitives like mutexes, semaphores, read-write locks, and condition variables should be used instead.

AUTHOR

pthread_suspend() and **pthread_continue()** were developed by X/Open.

pthread_resume_np() was developed by HP.

SEE ALSO

pthread_create(3T).

STANDARDS CONFORMANCE

pthread_continue(): X/Open.
pthread_resume_np(): None.
pthread_suspend(): X/Open.

NAME

pthread_rwlock_init(), pthread_rwlock_destroy() - initialize or destroy a read-write lock.

SYNOPSIS

```
#include <pthread.h>

int pthread_rwlock_init(
   pthread_rwlock_t *rwlock,
   const pthread_rwlockattr_t *attr
);

pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;

int pthread_rwlock_destroy(
   pthread_rwlock_t *rwlock
);
```

PARAMETERS

rwlock Pointer to the read-write lock to be initialized or destroyed.

attr Pointer to the attributes object that defines the characteristics of the read-write lock to be initialized. If the pointer is **NULL**, default attributes are used.

DESCRIPTION

pthread_rwlock_init() initializes the read-write lock referenced by *rwlock* with the attributes *attr*. If *attr* is **NULL**, the default read-write lock attributes are used. Upon successful initialization, the state of the read-write lock becomes initialized and unlocked. Attempting to initialize an already initialized read-write lock object results in undefined behavior.

The macro **PTHREAD_RWLOCK_INITIALIZER** can be used to initialize read-write locks which are statically allocated. The effect is equivalent to dynamic initialization by a call to **pthread_rwlock_init()** with the *attr* parameter specified as **NULL**, except that no error checks are performed. The read-write lock will be initialized with default attributes.

If the *process-shared* attribute in the read-write lock attributes object referenced by *attr* is defined as **PTHREAD_PROCESS_SHARED**, the read-write lock must be allocated such that the processes sharing the read-write lock has access to it. This may be done through the **memory-mapping** functions (see *mmap*(2)) or **shared memory** functions (see *shmget*(2)).

pthread_rwlock_destroy() destroys the read-write referenced by *rwlock*. This function may set *rwlock* to an invalid value. The destroyed read-write lock can be reinitialized using the function **pthread_rwlock_init()**. If the read-write lock is referenced after destruction in any read-write lock call, the resulting behavior is undefined.

A read-write lock should be destroyed only when no threads are currently using it. Destroying a read-write lock which is currently in use results in undefined behavior.

RETURN VALUE

Upon successful completion, **pthread_rwlock_init()** and **pthread_rwlock_destroy()** returns zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

If any of the following occur, the **pthread_rwlock_init()** function returns the corresponding error number:

[EAGAIN] The necessary resources (other than memory) to initialize *rwlock* are not available.

[ENOMEM] There is insufficient memory available in which to initialize the read-write lock *rwlock*.

[EPERM] The caller does not have the privilege to perform the operation.

For each of the following conditions, if the condition is detected, the **pthread_rwlock_init()** function returns the corresponding error number:

[EINVAL] The value specified by *rwlock* or *attr* is invalid.

[EBUSY] *rwlock* is an already initialized read-write lock.

For each of the following conditions, if the condition is detected, the **pthread_rwlock_destroy()** function returns the corresponding error number:

[EINVAL] The value specified by *rwlock* is invalid.

[EBUSY] *rwlock* is currently locked or being used by other threads.

WARNINGS

The space for the read-write lock must to be allocated before calling **pthread_rwlock_init()**. Undefined behavior may result if the *process-shared* attribute of *attr* is **PTHREAD_PROCESS_SHARED** and the space allocated for the read-write lock is not accessible to cooperating threads.

AUTHOR

pthread_rwlock_init() and **pthread_rwlock_destroy()** were developed by X/Open.

SEE ALSO

pthread_rwlock_rdlock(3T), pthread_rwlock_wrlock(3T), pthread_rwlock_unlock(3T), pthread_rwlock_tryrdlock(3T), pthread_rwlock_trywrlock(3T).

STANDARDS CONFORMANCE

pthread_rwlock_init(): X/Open.
pthread_rwlock_destroy(): X/Open.

NAME

pthread_rwlock_rdlock(), pthread_rwlock_tryrdlock() - lock or attempt to lock a read-write lock for reading.

SYNOPSIS

```
#include <pthread.h>

int pthread_rwlock_rdlock(
   pthread_rwlock_t *rwlock
);

int pthread_rwlock_tryrdlock(
   pthread_rwlock_t *rwlock
);
```

PARAMETERS

rwlock Pointer to the read-write lock to be locked for reading.

DESCRIPTION

The **pthread_rwlock_rdlock()** function applies a read lock to the read-write lock object referenced by *rwlock*. The calling thread shall acquire a read lock if a writer does not hold the lock and there are no writers blocked on the lock. It is unspecified whether the calling thread acquires the lock when a writer does not hold the lock and there are writers waiting for the lock. If a writer holds the lock, the calling thread shall not acquire the read lock. If the read lock is not acquired, the calling thread blocks (that is, it does not return from the **pthread_rwlock_rdlock()** call) until it can acquire the lock. Results are undefined if the calling thread currently owns a write lock on *rwlock*.

Implementations shall be allowed to favor writers over readers to avoid writer starvation.

A thread may hold multiple concurrent locks on *rwlock* (that is, successfully call the **pthread_rwlock_rdlock()** function *n* times). If so, the thread must perform the matching unlocks (that is, it must call the **pthread_rwlock_unlock()** function *n* times).

The function **pthread_rwlock_tryrdlock()** applies a read lock as in the **pthread_rwlock_rdlock()** function with the exception that the function fails if any thread holds a write lock on *rwlock* or there are writers blocked on *rwlock*.

Results are undefined if any of these functions are called with an uninitialized read-write lock.

If a signal is delivered to a thread waiting for a read-write lock, upon return from the signal handler, the thread shall resume waiting for the read-write lock as if it was not interrupted.

p

RETURN VALUE

Upon successful completion, **pthread_rwlock_rdlock()** and **pthread_rwlock_tryrdlock()** return zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

If any of the following occur, the **pthread_rwlock_tryrdlock()** function returns the corresponding error number:

[EBUSY] The read-write lock *rwlock* could not be acquired for reading because a writer holds the lock or was blocked on it.

For each of the following conditions, if the condition is detected, the **pthread_rwlock_rdlock()** and **pthread_rwlock_tryrdlock()** functions return the corresponding error number:

[EINVAL] The value specified by *rwlock* does not refer to an initialized read-write lock.

[EDEADLK] The current thread already owns the read-write lock for writing.

[EAGAIN] The read lock could not be acquired because the maximum number of read locks for *rwlock* has been exceeded. This error is not detected on HP-UX.

AUTHOR

pthread_rwlock_rdlock() and **pthread_rwlock_rdlock()** were developed by X/Open.

SEE ALSO

pthread_rwlock_init(3T), pthread_rwlock_destroy(3T), pthread_rwlock_trywrlock(3T), pthread_rwlock_wrlock(3T), pthread_rwlock_unlock(3T).

STANDARDS CONFORMANCE

pthread_rwlock_rdlock(): X/Open.
pthread_rwlock_tryrdlock(): X/Open.

NAME

pthread_rwlock_unlock() - unlock a read-write lock.

SYNOPSIS

```
#include <pthread.h>

int pthread_rwlock_unlock(
   pthread_rwlock_t *rwlock
);
```

PARAMETERS

rwlock Pointer to the read-write lock to be unlocked.

DESCRIPTION

The function **pthread_rwlock_unlock()** is called by the owner to release the read-write lock referenced by *rwlock*. Results are undefined if the read-write lock *rwlock* is not held by the calling thread.

If this function is called to release a read lock on the read-write lock *rwlock* and there are other read locks currently held on this read-write lock, the read-write lock shall remain in the read locked state but without the current thread as one of its owners. If this function releases the last read lock for this read-write lock, the object shall be put in the unlocked state with no owners.

If this function is called to release a write lock on the read-write lock *rwlock*, the read-write lock shall be put in the unlocked state with no owners.

If the call to the **pthread_rwlock_unlock()** function results in the read-write lock becoming unlocked and there are threads waiting to acquire the read-write lock for writing, the scheduling policy is used to determine which thread shall acquire the read-write lock for writing. If there are threads waiting to acquire the read-write lock object for reading, the scheduling policy is used to determine the order in which the waiting threads shall acquire the read-write lock object for reading. If there are multiple threads blocked on *rwlock* for both read locks and write locks, it is unspecified whether the readers will acquire the lock first or whether a writer will acquire the lock first.

Results are undefined if this function is called with an uninitialized read-write lock.

RETURN VALUE

Upon successful completion, **pthread_rwlock_unlock()** returns zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

For each of the following conditions, if the condition is detected, the **pthread_rwlock_unlock()** function returns the corresponding error number:

[EINVAL] The value specified by *rwlock* does not refer to an initialized read-write lock object.

[EPERM] The current thread does not own the read-write lock.

AUTHOR

pthread_rwlock_unlock() was developed by X/Open.

SEE ALSO

pthread_rwlock_init(3T), pthread_rwlock_destroy(3T), pthread_rwlock_rdlock(3T), pthread_rwlock_wrlock(3T), pthread_rwlock_tryrdlock(3T), pthread_rwlock_trywrlock(3T).

STANDARDS CONFORMANCE

pthread_rwlock_unlock(): X/Open.

NAME

pthread_rwlock_wrlock(), pthread_rwlock_trywrlock() - lock or attempt to lock a read-write lock for writing.

SYNOPSIS

```
#include <pthread.h>

int pthread_rwlock_wrlock(
   pthread_rwlock_t *rwlock
);

int pthread_rwlock_trywrlock(
   pthread_rwlock_t *rwlock
);
```

PARAMETERS

rwlock Pointer to the read-write lock to be locked for writing.

DESCRIPTION

The **pthread_rwlock_wrlock()** function applies a write lock to the read-write lock object referenced by *rwlock*. The calling thread acquires the write lock if no other thread (reader or writer) holds the read-write lock *rwlock*. Otherwise, the thread blocks (that is, it does not return from the **pthread_rwlock_wrlock()** call) until it can acquire the lock. Results are undefined if the calling thread holds the read-write lock (whether a read or a write lock) at the time the call is made.

The function **pthread_rwlock_trywrlock()** applies a write lock as in the **pthread_rwlock_wrlock()** function with the exception that the function fails if any thread currently holds *rwlock* (for reading or writing).

Results are undefined if any of these functions are called with an uninitialized read-write lock.

If a signal is delivered to a thread waiting for a read-write lock, upon return from the signal handler, the thread shall resume waiting for the read-write lock as if it was not interrupted.

RETURN VALUE

Upon successful completion, **pthread_rwlock_wrlock()** and **pthread_rwlock_trywrlock()** return zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

If any of the following occur, the **pthread_rwlock_trywrlock()** function returns the corresponding error number:

[EBUSY] The read-write lock *rwlock* could not be acquired for writing because it was already locked for reading or writing.

For each of the following conditions, if the condition is detected, the **pthread_rwlock_wrlock()** and **pthread_rwlock_trywrlock()** functions return the corresponding error number:

[EINVAL] The value specified by *rwlock* does not refer to an initialized read-write lock.

[EDEADLK] The current thread already owns the read-write lock for reading or writing.

AUTHOR

pthread_rwlock_wrlock() and **pthread_rwlock_trywrlock()** were developed by X/Open.

SEE ALSO

pthread_rwlock_init(3T), pthread_rwlock_destroy(3T), pthread_rwlock_tryrdlock(3T), pthread_rwlock_rdlock(3T), pthread_rwlock_unlock(3T).

STANDARDS CONFORMANCE

pthread_rwlock_wrlock(): X/Open.
pthread_rwlock_trywrlock(): X/Open.

NAME

pthread_rwlockattr_getpshared(), pthread_rwlockattr_setpshared() - get or set the process-shared attribute

SYNOPSIS

```
#include <pthread.h>

int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
    int pshared);

int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *attr,
    int *pshared);
```

PARAMETERS

attr Pointer to the read-write lock attributes object whose attributes are to be set/retrieved.

pshared This parameter either specifies the new value of the *process-shared* attribute (set function) or points to the memory location where the *process-shared* attribute of *attr* is to be returned (get function).

DESCRIPTION

The attributes object *attr* must have been previously initialized with the function **pthread_rwlockattr_init()** before these functions are called.

Read-Write locks can be used only by threads within the process or shared by threads in multiple processes. The *process-shared* attribute in a read-write lock attributes object describes who may use the read-write lock. The legal values for the *process-shared* attribute are:

PTHREAD_PROCESS_SHARED
This option permits a read-write lock to be operated upon by any thread that has access to the memory where the read-write lock is allocated. The application is responsible for allocating the read-write lock in memory that multiple processes can access.

PTHREAD_PROCESS_PRIVATE
The read-write lock can only be operated upon by threads created within the same process as the thread that initialized the read-write lock. If threads of differing processes attempt to operate on such read-write lock, the behavior is undefined.

The default value of *process-shared* is **PTHREAD_PROCESS_PRIVATE**.

pthread_rwlockattr_setpshared() is used to set the *process-shared* attribute in the initialized attributes object *attr*. The new value of the *process-shared* attribute of *attr* is set to the value specified in the *pshared* parameter.

pthread_rwlockattr_getpshared() retrieves the value of the *process-shared* attribute from the read-write lock attributes object *attr*. The value of the *process-shared* attribute of *attr* is returned in the *pshared* parameter.

RETURN VALUE

Upon successful completion, **pthread_rwlockattr_getpshared()** and **pthread_rwlockattr_setpshared()** return zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

If any of the following occur, the **pthread_rwlockattr_getpshared()** and **pthread_rwlockattr_setpshared()** functions return the corresponding error number:

[ENOSYS] **_POSIX_THREAD_PROCESS_SHARED** is not defined and these functions are not supported.

For each of the following conditions, if the condition is detected, the **pthread_rwlockattr_getpshared()** and **pthread_rwlockattr_setpshared()** functions return the corresponding error number:

[EINVAL] The value specified by *attr* is invalid.

[EINVAL] The value specified by *pshared* is not a legal value.

[EINVAL] The value *pshared* points to an illegal address.

WARNINGS

If a read-write lock is created with the *process-shared* attribute defined as **PTHREAD_PROCESS_SHARED**, the cooperating processes should have access to the memory in which the read-write lock is allocated.

AUTHOR

pthread_rwlockattr_setpshared() and **pthread_rwlockattr_getpshared()** were developed by X/Open.

SEE ALSO

pthread_create(3T), pthread_rwlockattr_init(3T), pthread_rwlock_init(3T).

STANDARDS CONFORMANCE

pthread_rwlockattr_setpshared(): X/Open.
pthread_rwlockattr_getpshared(): X/Open.

NAME

pthread_rwlockattr_init(), pthread_rwlockattr_destroy() - initialize or destroy a read-write lock attributes object.

SYNOPSIS

```
#include <pthread.h>

int pthread_rwlockattr_init(
   pthread_rwlockattr_t *attr
);

int pthread_rwlockattr_destroy(
   pthread_rwlockattr_t *attr
);
```

PARAMETERS

attr Pointer to the read-write lock attributes object to be initialized or destroyed.

DESCRIPTION

pthread_rwlockattr_init() initializes the read-write lock attributes object *attr* with the default value for all attributes. The attributes object describes a read-write lock in detail and is passed to the read-write lock initialization function.

When a read-write lock attributes object is used to initialize a read-write lock, the values of the individual attributes determine the characteristics of the new read-write lock. Attributes objects act like additional parameters to object initialization.

After a read-write lock attributes object has been used to initialize one or more read-write lock, any function affecting the attributes object does not affect the previously initialized read-write locks.

The read-write lock attributes and their default values are:

process-shared The default value is **PTHREAD_PROCESS_PRIVATE**.

If an initialized read-write lock attributes object is reinitialized, undefined behavior results.

pthread_rwlockattr_destroy() destroys the read-write lock attributes object *attr*. The destroyed read-write lock attributes object ceases to exist and its resources are reclaimed. Referencing the object after it has been destroyed results in undefined behavior. A destroyed read-write lock attributes object can be reinitialized using the function **pthread_rwlockattr_init()**.

Read-write locks which have been already initialized using this attributes object are not affected by the destruction of the read-write lock attributes object.

RETURN VALUE

Upon successful completion, **pthread_rwlockattr_init()** and **pthread_rwlockattr_destroy()** return zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

For each of the following conditions, if the condition is detected, the **pthread_rwlockattr_init()** and **pthread_rwlockattr_destroy()** functions return the corresponding error number:

[ENOMEM] There is insufficient memory available in which to initialize *attr*.

[EINVAL] The value specified by *attr* is invalid.

AUTHOR

pthread_rwlockattr_init() and **pthread_rwlockattr_destroy()** were developed by X/Open.

SEE ALSO

pthread_create(3T), pthread_rwlockattr_getpshared(3T), pthread_rwlockattr_setpshared(3T), pthread_rwlock_init(3T).

STANDARDS CONFORMANCE
pthread_rwlockattr_init(): X/Open.
pthread_rwlockattr_destroy(): X/Open.

NAME
pthread_self() - obtain the thread ID for the calling thread.

SYNOPSIS
```
#include <pthread.h>
pthread_t pthread_self(void);
```

PARAMETERS
None.

DESCRIPTION
pthread_self() returns the **thread ID** of the calling thread. The thread ID returned is the same ID that is returned in the *thread* parameter to the creating thread at thread creation time. Thread IDs are guaranteed to be unique only within a process.

RETURN VALUE
pthread_self() always returns the thread ID of the current thread.

ERRORS
None.

AUTHOR
pthread_self() was derived from the IEEE POSIX P1003.1c standard.

SEE ALSO
pthread_create(3T), pthread_equal(3T), getpid(2).

STANDARDS CONFORMANCE
pthread_self(): POSIX 1003.1c.

p

NAME

pthread_setcancelstate(), pthread_setcanceltype() - set and retrieve the current thread's cancelability state or type.

SYNOPSIS

```
#include <pthread.h>

int pthread_setcancelstate(
   int state,
   int *oldstate
);

int pthread_setcanceltype(
   int type,
   int *oldtype
);
```

PARAMETERS

state Value to which the cancelability state of the calling thread is to be set.

oldstate Pointer to the location where the old cancelability state of the calling thread will be returned.

type Value to which the cancelability type of the calling thread is to be set.

oldtype Pointer to the location where the old cancelability type of the calling thread will be returned.

DESCRIPTION

pthread_setcancelstate() atomically sets the calling thread's cancelability state to the value in *state* and returns the previous cancelability state in *oldstate.* The legal values for *state* are:

PTHREAD_CANCEL_DISABLE
Disable cancelability for the calling thread. Cancellation requests against the calling thread are held pending.

PTHREAD_CANCEL_ENABLE
Enable cancelability for the calling thread. Cancellation requests against the calling thread may be acted upon. When a pending cancellation request will be acted upon depends on the thread's cancelability type.

By default, a thread's cancelability state is set to **PTHREAD_CANCEL_ENABLE** when it is created.

pthread_setcanceltype() atomically sets the calling thread's cancelability type to the value in *type* and returns the previous cancelability type in *oldtype.* The legal values for *type* are:

PTHREAD_CANCEL_ASYNCHRONOUS
New or pending cancellation requests against the calling thread may be acted upon at any time (if cancellation is enabled for the calling thread).

PTHREAD_CANCEL_DEFERRED
Cancellation requests for the calling thread are held pending until a cancellation point is reached.

A thread's cancelability type is set to **PTHREAD_CANCEL_DEFERRED** when it is created.

If a thread's cancelability state is disabled, the setting of the thread's cancelability type has no immediate effect. All cancellation requests are held pending. However, once cancelability is enabled again, the new type will be in effect.

RETURN VALUE

Upon successful completion, **pthread_setcancelstate()** and **pthread_setcanceltype()** return zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

For each of the following conditions, if the condition is detected, the **pthread_setcancelstate()** and **pthread_setcanceltype()** functions return the corresponding error number:

[EINVAL] *state* contains an invalid value.

[EINVAL] *type* contains an invalid value.

NOTES

Only functions that are async-cancel safe should be called from a thread that is asynchronously cancelable.

AUTHOR

pthread_setcancelstate() and **pthread_setcanceltype()** were derived from the IEEE POSIX P1003.1c standard.

SEE ALSO

pthread_exit(3T), pthread_join(3T), pthread_cancel(3T), pthread_cond_wait(3T), pthread_cond_timedwait(3T).

STANDARDS CONFORMANCE

pthread_setcancelstate(): POSIX 1003.1c.
pthread_setcanceltype(): POSIX 1003.1c.

p

NAME

pthread_sigmask() - examine/change the signal mask of the calling thread.

SYNOPSIS

```
#include <pthread.h>

int pthread_sigmask(
   int how,
   const sigset_t *set,
   sigset_t *oset
);
```

PARAMETERS

how	This parameter defines how the signal mask of the calling thread will be changed.
set	Pointer to the set of signals that will be used to change the currently blocked signal set.
oset	Pointer to where the previous signal mask will be returned.

DESCRIPTION

pthread_sigmask() allows the calling thread to examine and/or change its signal mask.

Unless it is a null pointer, the argument *set* points to a set of signals which are to be used to change the currently blocked signal set.

The argument *how* indicates how the set is changed. The legal values are:

SIG_BLOCK	The resulting set is the union of the current set and the signal set pointed to by *set*.
SIG_UNBLOCK	The resulting set is the intersection of the current set and the complement of the signal set pointed to by *set*.
SIG_SETMASK	The resulting set is the signal set pointed to by *set*.

If the argument *oset* is not a null pointer, the previous signal mask is returned in *oset*. If *set* is a null pointer, the value of the argument *how* is insignificant and the thread's signal mask is unchanged; thus, the call can be used to inquire about currently blocked signals.

If there any pending unblocked signals after the call to **pthread_sigmask()**, at least one of those signals is delivered before the call to **pthread_sigmask()** returns.

It is impossible to block the **SIGKILL** or **SIGSTOP** signal. This is enforced by the system without causing an error to be indicated.

The thread's signal mask is not changed if **pthread_sigmask()** fails for any reason.

RETURN VALUE

Upon successful completion, **pthread_sigmask()** returns zero. Otherwise, an error number is returned to indicate the error (the **errno** variable is not set).

ERRORS

If any of the following occur, the **pthread_sigmask()** function returns the corresponding error number:

[EINVAL]	*how* contains an invalid value.
[EFAULT]	*set* or *oset* points to an invalid address. The reliable detection of this error is implementation dependent.

AUTHOR

pthread_sigmask() was derived from the IEEE POSIX P1003.1c standard.

SEE ALSO

sigprocmask(2).

STANDARDS CONFORMANCE

pthread_sigmask(): POSIX 1003.1c.

NAME

pthread_testcancel() - process any pending cancellation requests.

SYNOPSIS

```
#include <pthread.h>
void pthread_testcancel(void);
```

PARAMETERS

None.

DESCRIPTION

The **pthread_testcancel()** function checks for any pending cancellation requests against the calling thread. If a cancellation request is pending and the calling thread has its cancelability state enabled, the cancellation request will be acted upon. If the cancelability state of the calling thread is disabled, this function will have no effect.

RETURN VALUE

None. The **pthread_testcancel()** function does not return a value.

If the calling thread acts upon a cancellation request, this function will not return; the calling thread will be terminated.

ERRORS

None.

AUTHOR

pthread_testcancel() was derived from the IEEE POSIX P1003.1c standard.

SEE ALSO

pthread_exit(3T), pthread_join(3T), pthread_setcancelstate(3T), pthread_setcanceltype(3T), pthread_cleanup_push(3T), pthread_cleanup_pop(3T), pthread_cond_wait(3T), pthread_cond_timedwait(3T).

STANDARDS CONFORMANCE

pthread_testcancel(): POSIX 1003.1c.

p

NAME

ptsname, ptsname_r - get the pathname of a slave pty (pseudo-terminal)

SYNOPSIS

```
char * ptsname (int fildes);
char * ptsname_r (int fildes, char *slavename, int len);
```

Remarks:

ptsname() and **ptsname_r()** support STREAMS pty (see *ptm*(7) and *pts*(7)), and non-STREAMS pty (see *pty*(7)) which have different device naming conventions. Notice that the STREAMS pty, being an optional feature, is supported only when it is installed on the system.

ptsname() and **ptsname_r()** are useful only on systems that follow the *insf*(1M) naming conventions for pty (STREAMS and non-STREAMS).

DESCRIPTION

The passed parameter, *fildes*, is a file descriptor of an opened master pty. **ptsname()** generates the name of the slave pty corresponding to this master pty. This means that their minor numbers will be identical.

ptsname_r() is the reentrant version of the **ptsname()** function. The passed parameter, *slavename*, is a pointer to a character array for the resulting null-terminated pathname of the slave pty. The passed parameter, *len*, indicates the length of this character array which must be at least 32 bytes long.

RETURN VALUE

Upon successful completion, **ptsname()** returns a string containing the full path name of a slave pty. Otherwise, a NULL pointer is returned. The return value is pointed to static data area which is overwritten with each call to **ptsname()**, so it should be copied if it is to be saved.

Upon successful completion, **ptsname_r()** stores the resulting slave name in the character array pointed to by the *slavename* parameter, and returns a value of 0 (zero). Otherwise, it returns a value of -1.

ERRORS

ptsname() fails and returns a NULL pointer under the following conditions:

- File descriptor does not refer to an open master pty.
- Request falls outside pty name-space.
- Pty device naming conventions have not been followed.
- **ptsname()** failed to find a match.

ptsname_r() also fails under the above-mentioned conditions but instead it returns a -1 and sets **errno** to ENXIO.

ptsname_r() returns a -1 and sets **errno** to ERANGE if the **slavename** parameter is invalid or the **len** parameter is too small,

EXAMPLES

The following example shows how **ptsname()** is typically used for non-STREAMS pty to obtain the pathname of the slave pty corresponding to a master pty obtained through a pty clone open.

```
int fd_master;
char *path;
    ...
     fd_master = open("/dev/ptym/clone", O_RDONLY);
     path = ptsname(fd_master);
```

The following example shows how **ptsname()** is typically used on obtaining the pathname of the STREAMS slave pty corresponding to a STREAMS master pty.

```
int fd_master, fd_slave;
char *slave;
    ...
```

```
fd_master = open("/dev/ptmx", O_RDWR);
grantpt(fd_master);
unlockpt(fd_master);
slave = ptsname(fd_master);
fd_slave = open(slave, O_RDWR);
ioctl(fd_slave, I_PUSH, "ptem");
ioctl(fd_slave, I_PUSH, "ldterm");
```

AUTHOR

ptsname() and **ptsname_r()** were developed by HP and OSF.

SEE ALSO

insf(1M), devnm(3), pty(7), grantpt(3C), unlockpt(3C), ptm(7), pts(7), ptem(7), ldterm(7).

p

NAME

putc(), putchar(), fputc(), putw() - put character or word on a stream

SYNOPSIS

```
#include <stdio.h>
int putc(int c, FILE *stream);
int putchar(int c);
int fputc(int c, FILE *stream);
int putw(int w, FILE *stream);
int putc_unlocked(int c, FILE *stream);
int putchar_unlocked(int c);
```

Obsolescent Interfaces

```
int putw_unlocked(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 defined both as macros and as functions.

fputc() Same as **putc()**, but is a function rather than a macro, and can therefore be used as an argument. **fputc()** runs more slowly than **putc()**, but takes less space per invocation, and its name can be passed as an argument to a function.

putw() Writes the word (i.e., **int** in C) *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)) causes it to become buffered or line-buffered. **setbuf()** or **setvbuf()** (see *setbuf*(3S)) can be used to change the stream's buffering strategy. **putc_unlocked()**, **putchar_unlocked()**, put character on a stream.

Obsolescent Interfaces

putw_unlocked() put character or word on a stream.

APPLICATION USAGE

putc(), **fputc()**, **putchar()** and **putw()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing one of these interfaces.

RETURN VALUE

On success, **putc()**, **putc_unlocked()**, **fputc()**, **putchar()** and **putchar_unlocked()** each return the value they have written. On failure, they return the constant EOF, set the error indicator for the stream, and set **errno** to indicate the error.

On success, **putw()** and **putw_unlocked()** return 0. Otherwise, a non-zero value is returned, the error indicator for the stream is set, and **errno** is set to indicate the error.

ERRORS

putc(), **putc_unlocked()**, **putchar()**, **putchar_unlocked()**, **fputc()**, **putw()**, and **putw_unlocked()** fail if, either the *stream* is unbuffered or *stream*'s buffer needed to be flushed causing an underlying **write()** call to be invoked, and:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.

[EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for writing.

[EFBIG] An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size (see *ulimit*(2)).

[EINTR]	A signal was caught during the **write()** system call.
[EIO]	A physical I/O error has occurred, or the process is in a background process group and is attempting to write to its controlling terminal, **TOSTOP** is set, the process is neither ignoring nor blocking the **SIGTTOU** signal, and the process group of the process is orphaned.
[ENOSPC]	There was no free space remaining on the device containing the file.
[EPIPE]	An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal is also sent to the process.

Additional **errno** values can be set by the underlying **write()** function (see *write*(2)).

WARNINGS

The **putc()** and **putchar()** routines are implemented as both library functions and macros. The macro versions, which are used by default, are defined in <**stdio.h**>. To obtain the library function either use a **#undef** to remove the macro definition or, if compiling in ANSI-C mode, enclose the function name in parentheses or use the function address. The following example illustrates each of these methods:

```
#include <stdio.h>
#undef putc
  ...
main()
{
     int (*put_char()) ();
      ...
     return_val=putc(c,fd);
      ...
     return_val=(putc)(c,fd1);
      ...
     put_char = putchar;
};
```

Line buffering may cause confusion or malfunctioning of programs that use standard I/O routines but use **read()** themselves to read from standard input. When a large amount of computation is done after printing part of a line on an output terminal, it is necessary to **fflush()** (see *fclose*(3S)) the standard output before beginning the computation.

The macro version of **putc()** incorrectly treats the argument *stream* with side effects. In particular, the following call may not work as expected:

```
putc(c, *f++);
```

The function version of **putc()** or **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 readable by **getw()** on a different processor.

putw_unlocked() is an obsolescent interface supported only for compatibility with existing DCE applications. New multithreaded applications should use **putc()**, **putchar()** and **putw()**.

Reentrant Interfaces

If **_REENTRANT** is defined, the locked versions of the library functions for **putc()** and **putchar()** are used by default.

SEE ALSO

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

STANDARDS CONFORMANCE

putc(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

fputc(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

putchar(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

putw(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

NAME

putenv() - change or add value to environment

SYNOPSIS

```
#include <stdlib.h>
int putenv(const 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 changes the environment. The space used by *string* is no longer used once a new string-defining *name* is passed to **putenv()**.

APPLICATION USAGE

putenv() is thread-safe. It is not async-cancel-safe.

EXTERNAL INFLUENCES

Locale

The **LC_CTYPE** category determines the interpretation of characters in *string* as single- and/or multi-byte characters.

International Code Set Support

Single- and multi-byte character code sets are supported.

DIAGNOSTICS

putenv() returns non-zero if it was unable to obtain enough space via **malloc()** for an expanded environment, or if an invalid multibyte character sequence was encountered in the string argument; otherwise it returns zero.

ERRORS

putenv() fails under the following conditions:

[ENOMEM] There is insufficient space to expand the environment.

[EILSEQ] An invalid multibyte character sequence was encountered in the string argument.

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()** to enlarge the environment (see *malloc*(3C)).

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.

SEE ALSO

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

STANDARDS CONFORMANCE

putenv(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

(ENHANCED CURSES)

NAME

putp, tputs — output commands to the terminal

SYNOPSIS

```
#include <term.h>
int putp(const char *str);
int tputs(const char *str, int affcnt, int (*putfunc)(int));
```

DESCRIPTION

These functions output commands contained in the **terminfo** database to the terminal.

The **putp()** function is equivalent to *tputs*(*str*, 1, *putchar*). The output of **putp()** always goes to **stdout**, not to the *fildes* specified in **setupterm()**.

The **tputs()** function outputs *str* to the terminal. The *str* argument must be a **terminfo** string variable or the return value from **tgetstr()**, **tgoto()**, **tigetstr()** or **tparm()**. The *affcnt* argument is the number of lines affected, or 1 if not applicable. If the **terminfo** database indicates that the terminal in use requires padding after any command in the generated string, **tputs()** inserts pad characters into the string that is sent to the terminal, at positions indicated by the **terminfo** database. The **tputs()** function outputs each character of the generated string by calling the user-supplied function *putfunc* (see below).

The user-supplied function *putfunc* (specified as an argument to **tputs**) is either **putchar()** or some other function with the same prototype. The **tputs()** function ignores the return value of *putfunc*.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

After use of any of these functions, the model Curses maintains of the state of the terminal might not match the actual state of the terminal. The application should touch and refresh the window before resuming conventional use of Curses.

Use of these functions requires that the application contain so much information about a particular class of terminal that it defeats the purpose of using Curses.

On some terminals, a command to change rendition conceptually occupies space in the screen buffer (with or without width). Thus, a command to set the terminal to a new rendition would change the rendition of some characters already displayed.

SEE ALSO

doupdate(3X), is_linetouched(3X), putchar() (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification), tgetent(3X), tigetflag(3X), <term.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

putpwent() - write password file entry

SYNOPSIS

```
#include <pwd.h>
#include <stdio.h>
int putpwent(const struct passwd *p, FILE *f);
```

DESCRIPTION

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

putpwent() ignores the audit ID and audit flag in the **passwd** structure; and *does not* create the corresponding entries used in the protected password database, used for trusted systems. **putprpwnam()**, which produces entries that match the trusted password database file format, must be used to create these entries. See *getprpwent*(3).

APPLICATION USAGE

putpwent() is thread-safe. It is not async-cancel-safe. A cancellation point may occur when a thread is executing **putpwent()**.

DIAGNOSTICS

putpwent() returns non-zero if an error was detected during its operation; otherwise it returns zero.

FILES

/etc/passwd	System Password file

SEE ALSO

getpwent(3C), getprpwent(3), passwd(4), prpwd(4), stdio(3S), fopen(3S).

STANDARDS CONFORMANCE

putpwent(): SVID2, SVID3, XPG2

NAME

puts(), fputs() - put a string on a stream

SYNOPSIS

```
#include <stdio.h>
int puts(const char *s);
int fputs(const char *s, FILE *stream);
```

Obsolescent Interfaces

```
int puts_unlocked(const char *s);
int fputs_unlocked(const 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*, but does *not* append a new-line character.

Neither function writes the terminating null character.

Obsolescent Interfaces

puts_unlocked() and **fputs_unlocked()** put a string on a stream.

APPLICATION USAGE

puts() and **fputs()** are thread-safe interfaces. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing **puts()** or **fputs()**.

RETURN VALUE

Upon successful completion, these routines return a non-negative number. Otherwise they return EOF, set the error indicator for the stream, and set **errno** to indicate the error.

ERRORS

These routines fail if, either the *stream* is unbuffered or *stream*'s buffer needed to be flushed causing an underlying **write()** call to be invoked, and:

[EAGAIN]	The **O_NONBLOCK** flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.
[EBADF]	The file descriptor underlying *stream* is not a valid file descriptor open for writing.
[EFBIG]	An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size (see *ulimit*(2)).
[EINTR]	A signal was caught during the **write()** system call.
[EIO]	The process is in a background process group and is attempting to write to its controlling terminal, **TOSTOP** is set, the process is neither ignoring nor blocking the **SIGTTOU** signal, and the process group of the process is orphaned.
[ENOSPC]	There was no free space remaining on the device containing the file.
[EPIPE]	An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A SIGPIPE signal is also sent to the process.

Additional **errno** values may be set by the underlying **write()** function (see *write*(2)).

WARNINGS

puts_unlocked() and **fputs_unlocked()** are obsolescent interfaces supported only for compatibility with existing DCE applications. New multithreaded applications should use **puts()** and **fputs()**.

NOTES

puts() and **puts_unlocked()** append a new-line character; **fputs()** and **fputs_unlocked()** do not.

SEE ALSO
ferror(3S), flockfile(3S), fopen(3S), fread(3S), printf(3S), putc(3S).

STANDARDS CONFORMANCE
puts(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
fputs(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

putwc(), putwchar(), fputwc() - put a wide character on a stream file

SYNOPSIS

```
#include <wchar.h>
wint_t putwc(wint_t wc, FILE *stream);
wint_t putwchar(wint_t wc);
wint_t fputwc(wint_t wc, FILE *stream);
```

Obsolescent Interfaces

```
wint_t putwc_unlocked(wint_t wc, FILE *stream);
wint_t putwchar_unlocked(wint_t wc);
wint_t fputwc_unlocked(wint_t wc, FILE *stream);
```

Remarks:

These functions are compliant with the XPG4 Worldwide Portability Interface wide-character I/O functions. They parallel the 8-bit character I/O functions defined in *putc*(3S).

DESCRIPTION

putwc() Writes the character corresponding to the wide character *wc* onto the output *stream* at the position where the file pointer is pointing. **putwchar(***wc***)** is defined as **putwc(***wc***, stdout)**. **putwc()** and **putwchar()** are defined both as macros and as functions.

fputwc() Behaves like **putwc()**, but is a function rather than a macro, and can therefore be used as an argument.

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)) causes it to become buffered or line-buffered. **setbuf()** or **setvbuf()** (see *setbuf*(3S)) can be used to change the stream's buffering strategy.

Definitions for these functions, the type *wint_t* and the value WEOF are provided in the <**wchar.h**> header.

p

Obsolescent Interfaces

putwc_unlocked(), **putwchar_unlocked()**, and **fputwc_unlocked()** put a wide character on a stream file.

APPLICATION USAGE

putwc(), **putwchar()** and **fputwc()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing **putwc()**, **putwchar()** or **fputwc()**.

EXTERNAL INFLUENCES

Locale

The **LC_CTYPE** category determines how wide character conversions are done.

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

On success, **putwc()**, **putwc_unlocked()**, **fputwc()**, **fputwc_unlocked()**, **putwchar()**, and **putwchar_unlocked()** each return the wide character corresponding to the value they have written. On failure, they return the constant **WEOF**, set the error indicator for the stream, and set **errno** to indicate the error.

ERRORS

putwc(), **putwc_unlocked()**, **putwchar()**, **putwchar_unlocked()**, **fputwc()**, and **fputwc_unlocked()** fail if either the *stream* is unbuffered, or *stream*'s buffer needed to be flushed causing an underlying **write()** call to be invoked, and:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.

[EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for writing.

[EFBIG] An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size (see *ulimit*(2)).

[EINTR] A signal was caught during the **write()** system call.

[EIO] A physical I/O error has occurred, or the process is in a background process group and is attempting to write to its controlling terminal, **TOSTOP** is set, the process is neither ignoring nor blocking the **SIGTTOU** signal, and the process group of the process is orphaned.

[ENOSPC] There was no free space remaining on the device containing the file.

[EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal is also sent to the process.

[EILSEQ] The wide character *wc* does not correspond to a valid character.

Additional **errno** values can be set by the underlying **write()** function (see *write*(2)).

WARNINGS

Line buffering may cause confusion or malfunctioning of programs that use wide character I/O routines but use **read()** themselves to read from standard input. When a large amount of computation is done after printing part of a line on an output terminal, it is necessary to **fflush()** (see *fclose*(3S)) the standard output before beginning the computation.

putwc_unlocked(), **putwchar_unlocked()** and **fputwc_unlocked()** are obsolescent interfaces supported only for compatibility with existing DCE applications. New multithreaded applications should use **putwc()**, **putwchar()** and **fputwc()**.

AUTHOR

putwc() was developed by OSF and HP.

SEE ALSO

fclose(3S), ferror(3S), flockfile(3S), fopen(3S), getwc(3C), fread(3S), printf(3S), putws(3C), setbuf(3S).

STANDARDS CONFORMANCE

putwc(): XPG4

fputwc(): XPG4

putwchar(): XPG4

p

NAME

putws(), fputws() - put a wide character string on a stream file

SYNOPSIS

```
#include <wchar.h>
int putws(const wchar_t *ws);
int fputws(const wchar_t *ws, FILE *stream);
```

Obsolescent Interfaces

```
int putws_unlocked(const wchar_t *ws);
int fputws_unlocked(const wchar_t *ws, FILE *stream);
```

Remarks:

fputws is compliant with the XPG4 Worldwide Portability Interface wide-character I/O functions. These functions parallel the 8 bit character I/O functions defined in *puts*(3S).

DESCRIPTION

putws() writes a character string corresponding to the null-terminated wide-character string pointed to by *ws* followed by a new-line character, to the standard output stream **stdout**.

fputws() writes a character string corresponding to the null-terminated wide-character string pointed to by *ws* to the named output *stream*, but does *not* append a new-line character or a terminating null character.

Neither function writes a terminating null character.

The definition for these functions, the type **wchar_t** and the value **WEOF** are provided in the <**wchar.h**> header.

Obsolescent Interfaces

putws_unlocked() and **fputws_unlocked()** put a wide character string on a stream file.

APPLICATION USAGE

putws() and **fputws()** are thread-safe interfaces. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing **putws()** or **fputws()**.

p

EXTERNAL INFLUENCES

Locale

The **LC_CTYPE** category determines how wide character conversions are done.

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

Upon successful completion, **putws()**, **putws_unlocked()**, **fputws()**, and **fputws_unlocked()** return a non-negative number. Otherwise they return **WEOF**, set the error indicator for the stream, and set **errno** to indicate the error.

ERRORS

putws(), **putws_unlocked()**, **fputws()**, and **fputws_unlocked()** fail if either the *stream* is unbuffered, or *stream*'s buffer needed to be flushed causing an underlying **write()** call to be invoked, and:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.

[EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for writing.

[EFBIG] An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size (see *ulimit*(2)).

[EINTR] A signal was caught during the **write()** system call.

[EIO] The process is in a background process group and is attempting to write to its controlling terminal, **TOSTOP** is set, the process is neither ignoring nor blocking the

SIGTTOU signal, and the process group of the process is orphaned.

[ENOSPC] There was no free space remaining on the device containing the file.

[EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal is also sent to the process.

[EILSEQ] A wide character in *ws* does not correspond to a valid character.

Additional **errno** values may be set by the underlying **write()** function (see *write*(2)).

WARNINGS

putws_unlocked() and **fputws_unlocked()** are obsolescent interfaces supported only for compatibility with existing DCE applications. New multithreaded applications should use **putws()** and **fputws()**.

AUTHOR

putws() and **fputws()** were developed by OSF and HP.

SEE ALSO

ferror(3S), flockfile(3S), fopen(3S), fread(3S), printf(3S), putwc(3C).

STANDARDS CONFORMANCE

fputws(): XPG4

NAME

qsort() - quicker sort

SYNOPSIS

```
#include <stdlib.h>

void qsort(
     void *base,
     size_t nel,
     size_t size,
     int (*compar)(const void *, const void *)
);
```

DESCRIPTION

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

base Pointer to the element at the base of the table.

nel Number of elements in the table.

size Size of each element in the table.

compar Name of the comparison function, which is called with two arguments that point to the elements being compared. The function passed as *compar* must return an integer less than, equal to, or greater than zero, according to whether its first argument is to be considered less than, equal to, or greater than the second. **strcmp()** uses this same return convention (see *string*(3C)).

APPLICATION USAGE

qsort() is thread-safe. It is not async-cancel-safe.

NOTES

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

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

The order in the output of two items which compare as equal is unpredictable.

WARNINGS

If *size* is zero, a divide-by-zero error might be generated.

q

SEE ALSO

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

STANDARDS CONFORMANCE

qsort(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

rand(), rand_r(), srand() - simple random-number generator

SYNOPSIS

```
#include <stdlib.h>
int rand(void);
int rand_r(unsigned int *seed);
void srand(unsigned int 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.

rand_r() returns a random number at the address pointed to by the *randval* parameter. The *seed* parameter can be set at any time to start the random-number generator at an arbitrary point.

APPLICATION USAGE

Both **rand()** and **rand_r()** are thread-safe. The **rand_r()** interface has been provided to allow multiple threads to generate the same sequence of random numbers concurrently. These interfaces are not async-cancel-safe.

RETURN VALUE

If *seed* or *randval* is NULL, **rand_r()** returns 0. Otherwise, **rand_r()** returns a psuedo-random integer.

EXAMPLE

The following:

```
int x, y;
srand(10);
x = rand();
y = rand();
```

would produce the same results as:

```
int x, y, s = 10;
x=rand_r(&s);
y=rand_r(&s);
```

NOTE

The spectral properties of **rand()** leave a great deal to be desired. **drand48()** provides a much better, though more elaborate, random-number generator (see *drand48*(3C)).

WARNINGS

Users of **rand_r()** should note that rand_r() now conforms with POSIX.1c. The old prototype of **rand_r()** is supported for compatibility with existing DCE applications only.

SEE ALSO

drand48(3C), random(3M).

STANDARDS CONFORMANCE

rand(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

rand_r(): POSIX.1c

srand(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

random(), srandom(), initstate(), setstate() - generate a pseudorandom number

SYNOPSIS

```
#include <stdlib.h>
long random(void);
void srandom(unsigned seed);
char *initstate(unsigned seed, char *state, size_t size);
char *setstate(char *state);
```

DESCRIPTION

The **random()** and **srandom()** functions are random-number generators that have virtually the same calling sequence and initialization properties as the **rand()** and **srand()** functions, but produce sequences that are more random. The low 12 bits generated by the **rand()** function go through a cyclic pattern, while all the bits generated by the **random()** function are usable. For example, **random() & 01** produces a random binary value.

The **random()** function uses a nonlinear additive feedback random-number generator employing a default state array size of 31 long integers to return successive pseudorandom numbers in the range from 0 to 2^{31}-1. The period of this random-number generator is approximately 16 x (2^{31}-1). The size of the state array determines the period of the random-number generator. Increasing the state array size increases the period.

With 256 bytes of state information, the period of the random-number generator is greater than 2^{69}.

Like the **rand()** function, the **random()** function produces by default a sequence of numbers that can be duplicated by calling the **srandom()** function with a value of 1 as the seed.

The **srandom()** function initializes the current state array using the value of *seed*.

The **initstate()** and **setstate()** functions handle restarting and changing random-number generators. The **initstate()** function allows a state array, pointed to by the *state* argument, to be initialized for future use. The *size* argument, which specifies the size in bytes of the state array, is used by the **initstate()** function to decide how sophisticated a random-number generator to use; the larger the state array, the more random the numbers. Values for the amount of state information are 8, 32, 64, 128, and 256 bytes. Amounts less than 8 bytes generate an error, while other amounts are rounded down to the nearest known value. The *seed* argument specifies a starting point for the random-number sequence and provides for restarting at the same point. The **initstate()** function returns a pointer to the previous state information array.

Once a state has been initialized, the **setstate()** function allows switching between state arrays. The array defined by the *state* argument is used for further random-number generation until the **initstate()** function is called or the **setstate()** function is called again. The **setstate()** function returns a pointer to the previous state array.

After initialization, a state array can be restarted at a different point in one of two ways:

The **initstate()** function can be used, with the desired seed, state array, and size of the array.

The **setstate()** function, with the desired state, can be used, followed by the **srandom()** function with the desired seed. The advantage of using both of these functions is that the size of the state array does not have to be saved once it is initialized.

RETURN VALUE

The **random()** function returns the generated pseudorandom number.

The **srandom()** function returns no value.

Upon successful completion, the **initstate()** and **setstate()** functions return a pointer to the previous state array. Otherwise, a NULL pointer is returned.

ERRORS

If the **initstate()** function is called with *size* less than 8, or if the **setstate()** function detects that the state information has been damaged, error messages are written to standard error.

SEE ALSO
drand48(3C), rand(3C).

STANDARDS COMPLIANCE
random() COSE API, XPG 4.2
srandom() COSE API, XPG 4.2
initstate() COSE API, XPG 4.2
setstate() COSE API, XPG 4.2

NAME

rcmd(), rresvport(), ruserok() - return a stream to a remote command

SYNOPSIS

```
int rcmd(
     char **ahost,
     int remport,
     const char *locuser,
     const char *remuser,
     const char *cmd,
     int *fd2p);

int rresvport(int *port);

int ruserok(
     const char *rhost,
     int superuser,
     const char *ruser,
     const char *luser);
```

DESCRIPTION

rcmd()

The **rcmd()** function is used by privileged programs to execute a command on a remote host. **rcmd()** returns a file descriptor for the socket to which the standard input and standard output of the command are attached. A command level interface to **rcmd()** is provided by **remsh** (see *remsh*(1)), which is the same as the BSD **rsh** command.

ahost is a pointer to the address of the remote host name. The name of the remote host can be either an official host name or an alias as understood by **gethostbyname()** (see *gethostent*(3N), *named*(1M), and *hosts*(4)).

remport is the Internet port on the remote system, which **rcmd()** will try to connect to.

locuser and *remuser* point to the user login name on the local host and on the remote host, respectively. The names are used by the server on the remote host to authenticate the user (see **ruserok()** below).

cmd points to a string that specifies the command to be executed on the remote host.

fd2p is a pointer to an integer (it can be a NULL pointer).

rcmd() looks up the host ***ahost* using **gethostbyname()**, returning –1 if the host does not exist. Otherwise, ***ahost* is set to the standard name of the host and a connection is established to a server accepting requests at the port *remport*. If the connection is refused after five tries, or if it was refused for a reason other than the port being in use, **rcmd()** returns –1.

If the call succeeds, a socket of type SOCK_STREAM is returned to the caller and given to the remote command as **stdin** and **stdout**. If *fd2p* is non-NULL, **rcmd()** opens a second socket between the calling process (local) and the control process (remote), and places its descriptor in ***fd2p*. On this connection, the control process sends the diagnostic output (**stderr**) from *cmd* to the calling process, and receives UNIX signals from the calling process, to be forwarded to *cmd*. If the auxiliary port cannot be set up, **rcmd()** returns –1. If *fd2p* is NULL, **stderr** of the remote command is made the same as **stdout**, and no provision is made for sending arbitrary signals to the remote process.

The protocol is described in detail in *remshd*(1M).

Any program using **rcmd()** must be run as superuser.

rresvport()

The **rresvport()** function creates a socket and binds it to a reserved port. This socket is suitable for use by **rcmd()** and several other routines.

The caller is expected to set the initial value of ***port* to a number between 512 and **IPPORT_RESERVED**–1. (The value of **IPPORT_RESERVED** is defined in **netinet/in.h** and is 1024.) Typically, the initial value of ***port* is set to **IPPORT_RESERVED**–1. If the value is outside the valid range, **rresvport()** resets it silently to **IPPORT_RESERVED**–1. The function uses the inital value of ***port* as the first port number that it tries to *bind* to the created socket. If the operation fails, **rresvport()** decrements ***port* and attempts to *bind* the new port number to the socket. The process is repeated until either the operation succeeds, or the port numbers between 512 and

IPPORT_RESERVED–1 are exhausted.

If the call succeeds, the socket descriptor is returned to the caller and the port number is returned in the location pointed to by *port*. If the call fails, –1 is returned to the caller.

The socket returned by **rresvport()** has the **SO_KEEPALIVE** option on.

Only the superuser is permitted to bind a privileged address to a socket. Therefore, any program using **rresvport()** must be run as superuser.

ruserok()

The **ruserok()** function is used by servers to authenticate clients requesting service with **rcmd()**. **ruserok()** verifies that *ruser* on *rhost* is authorized to act as *luser* on the local host.

superuser is an integer flag that should be nonzero if the local user name corresponds to a superuser. If the *superuser* flag is not set, **ruserok()** first checks the file **/etc/hosts.equiv** to authenticate the request for service. If this check succeeds, **ruserok()** returns 0. If the *superuser* flag is set, or if there is no file **/etc/hosts.equiv**, or if the check fails, **ruserok()** then checks the file **.rhosts** (if there is one) in the local user's home directory. **ruserok()** returns 0 if this check succeeds. Otherwise, it returns –1.

Typically, the file **/etc/hosts.equiv** contains a list of host names, and users' **.rhosts** files contain host-name/user-name pairs. A remote user is authenticated by **ruserok()** if the remote host name appears in **/etc/hosts.equiv** and the remote user name and local user name are the same, or if the remote host name and the remote user name appear together in **.rhosts** in the home directory of the local user.

For a complete explanation of the syntax understood by **ruserok()**, see *hosts.equiv*(4).

MULTITHREADED USAGE

Thread Safe: Yes
Cancel Safe: Yes
Async-cancel Safe: No
Async-signal Safe: No

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancellation points.

DIAGNOSTICS

rcmd() Diagnostic Messages

rcmd() generates the following diagnostic messages.

hostname: **Unknown host**

gethostbyname was unable to find an entry in the hosts database matching the name of the server (see *gethostent*(3N) and *hosts*(4)).

Next step: Have the system administrator of your host check whether the remote host's entry is in the **hosts** database (see *hosts*(4)).

connect: *hostname*: ...

Unable to establish a connection to the reserved port. A message that specifies the reason for the failure is appended to this diagnostic message.

write: Setting up stderr

Error writing to the socket connection set up for error message transmission.

system call: ...

Error executing the system call. Appended to this error is a message specifying the reason for the failure.

socket: Protocol failure in circuit setup

Socket connection not established on a reserved port or socket address not of the Internet family type.

read: *hostname*: ...

Error in reading information from the standard socket connection. Appended to this error is a message explaining the reason for the error.

Connection timeout

The remote host did not connect within 30 seconds to the secondary socket set up as an error connection.

Lost connection

The program attempted to read from the socket and failed. This means the socket connection with the remote host was lost.

message...

An error message can be transmitted through the socket connection from the daemon. That message will be sent to **stderr**.

primary connection shutdown

While waiting for the secondary socket to be set up, **rcmd()** had its primary connection shut down. This may have been caused by an *inetd* security failure.

recv: ...

While trying to set up the secondary (**stderr**) socket, **rcmd()** had an error condition on its primary connection.

accept: Interrupted system call

While trying to set up its secondary socket, **rcmd()** ran out of some resource that caused the accept to be timed out.

Next step: Repeat the command.

rresvport() Diagnostic Messages

rresvport() generates the following diagnostic messages. These messages can also appear in **rcmd()** since **rcmd()** calls **rresvport()**.

*system call***:** ...

Error in executing the system call. The error message returned by the system call is appended to the message.

socket: All ports in use

All reserved ports in use. If a timeout occurs, check whether the Internet Services are installed and **inetd** is running.

EXAMPLES

To execute the **date** command on remote host **hpxzgy** using the remote account **chm**, call **rcmd()** as shown below. This program requires superuser privileges and the remote account must be equivalent (see *hosts.equiv*(4)) to the local account that runs the program.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <pwd.h>

struct passwd *getpwuid();
char *host[] = { "hpxzgy"  };
char *cmd = "date";
char *ruser = "chm";

main(argc,argv)
        int argc;
        char **argv;
{
        struct servent *sp;
        struct passwd *pwd;
```

```
        FILE *fp;
        char ch;
        int rem;

        sp = getservbyname("shell","tcp");
        pwd = getpwuid(getuid());
        rem = rcmd(host, sp->s_port, pwd->pw_name, ruser, cmd, 0);
        if (rem < 0)
                exit(1);   /* rcmd outputs its own error messages */
        fp = fdopen(rem, "r");
        while ((ch = getc(fp)) != EOF)
                putchar(ch);
}
```

WARNINGS

There is no way to specify options to the **socket()** call that **rcmd()** makes. Since **rcmd()** replaces the pointer to the host name (**ahost*) with a pointer to the standard name of the host in a static data area, this value must be copied into the user's data area if it is to be used later. Otherwise, unpredictable results will occur.

AUTHOR

rcmd() was developed by the University of California, Berkeley.

SEE ALSO

login(1), rlogin(1), remsh(1), named(1M), remshd(1M), rexecd(1M), gethostent(3N), rexec(3N), hosts.equiv(4).

r

NAME

re_comp, re_exec — compile and execute regular expressions (*TO BE WITHDRAWN*)

SYNOPSIS

```
#include <re_comp.h>
char *re_comp(const char *string);
int re_exec(const char *string);
```

DESCRIPTION

The **re_comp()** function converts a regular expression string (RE) into an internal form suitable for pattern matching. The **re_exec()** function compares the string pointed to by the *string* argument with the last regular expression passed to **re_comp()**.

If **re_comp()** is called with a null pointer argument, the current regular expression remains unchanged.

Strings passed to both **re_comp()** and **re_exec()** must be terminated by a null byte, and may include newline characters.

The **re_comp()** and **re_exec()** functions support **simple regular expressions**, which are defined below.

The following one-character REs match a single character:

1.1 An ordinary character (not one of those discussed in 1.2 below) is a one-character RE that matches itself.

1.2 A backslash (\) followed by any special character is a one-character RE that matches the special character itself. The special characters are:

a. **.**, *****, **[**, and \ (period, asterisk, left square bracket, and backslash, respectively), which are always special, except when they appear within square brackets (**[ ]**; see 1.4 below).

b. ^(caret or circumflex), which is special at the beginning of an entire RE (see 3.1 and 3.2 below), or when it immediately follows the left of a pair of square brackets (**[ ]**) (see 1.4 below).

c. **$** (dollar symbol), which is special at the end of an entire RE (see 3.2 below).

d. The character used to bound (delimit) an entire RE, which is special for that RE.

1.3 A period (**.**) is a one-character RE that matches any character except new-line.

1.4 A non-empty string of characters enclosed in square brackets (**[ ]**) is a one-character RE that matches any one character in that string. If, however, the first character of the string is a circumflex (^), the one-character RE matches any character except new-line and the remaining characters in the string. The ^ has this special meaning only if it occurs first in the string. The minus (**-**) may be used to indicate a range of consecutive ASCII characters; for example, **[0-9]** is equivalent to **[0123456789]**. The **-** loses this special meaning if it occurs first (after an initial ^, if any) or last in the string. The right square bracket (**]**) does not terminate such a string when it is the first character within it (after an initial ^, if any); for example, **[ ]a-f]** matches either a right square bracket (**]**) or one of the letters **a** through **f** inclusive. The four characters listed in 1.2.a above stand for themselves within such a string of characters.

The following rules may be used to construct REs from one-character REs:

2.1 A one-character RE is a RE that matches whatever the one-character RE matches.

2.2 A one-character RE followed by an asterisk (*****) is a RE that matches zero or more occurrences of the one-character RE. If there is any choice, the longest leftmost string that permits a match is chosen.

2.3 A one-character RE followed by **\{***m***\}**, **\{***m***,\}**, or **\{***m***,***n***\}** is a RE that matches a range of occurrences of the one-character RE. The values of *m* and *n* must be non-negative integers less than 256; **\{***m***\}** matches exactly *m* occurrences; **\{***m***,\}** matches at least *m* occurrences; **\{***m***,***n***\}** matches any number of occurrences between *m* and *n* inclusive. Whenever a choice exists, the RE matches as many occurrences as possible.

2.4 The concatenation of REs is a RE that matches the concatenation of the strings matched by each component of the RE.

2.5 A RE enclosed between the character sequences **\(** and **\)** is a RE that matches whatever the unadorned RE matches.

2.6 The expression *\n* matches the same string of characters as was matched by an expression enclosed between **\(** and **\)** earlier in the same RE. Here *n* is a digit; the sub-expression specified is that beginning with the *n -th* occurrence of **\(** counting from the left. For example, the expression ^\(.*\)\1$ matches a line consisting of two repeated appearances of the same string.

Finally, an entire RE may be constrained to match only an initial segment or final segment of a line (or both).

3.1 A circumflex (^) at the beginning of an entire RE constrains that RE to match an initial segment of a line.

3.2 A dollar symbol (**$**) at the end of an entire RE constrains that RE to match a final segment of a line. The construction *^entire RE*$ constrains the entire RE to match the entire line.

The null RE (that is, *//*) is equivalent to the last RE encountered.

The behaviour of **re_comp()** and **re_exec()** in locales other than the POSIX locale is unspecified.

RETURN VALUE

The **re_comp()** function returns a null pointer when the string pointed to by the *string* argument is successfully converted. Otherwise, a pointer to an unspecified error message string is returned.

Upon successful completion, **re_exec()** returns 1 if *string* matches the last compiled regular expression. Otherwise, **re_exec()** returns 0 if *string* fails to match the last compiled regular expression, and –1 if the compiled regular expression is invalid (indicating an internal error).

ERRORS

No errors are defined.

APPLICATION USAGE

For portability to implementations conforming to earlier versions of this document, **regcomp()** and **regexec()** are preferred to these functions.

SEE ALSO

regcomp(3C), <re_comp.h>.

CHANGE HISTORY

First released in Issue 4, Version 2.

NAME
realpath — resolve pathname

SYNOPSIS
```
#include <stdlib.h>
char *realpath(const char *file_name, char *resolved_name);
```

DESCRIPTION
The **realpath()** function derives, from the pathname pointed to by *file_name*, an absolute pathname that names the same file, whose resolution does not involve "**.**", "**..**", or symbolic links. The generated pathname is stored, up to a maximum of **{PATH_MAX}** bytes, in the buffer pointed to by *resolved_name*.

APPLICATION USAGE
realpath() is thread-safe. It is not async-cancel-safe. A cancellation point may occur when a thread is executing **realpath()**.

RETURN VALUE
On successful completion, **realpath()** returns a pointer to the resolved name. Otherwise, **realpath()** returns a null pointer and sets *errno* to indicate the error, and the contents of the buffer pointed to by *resolved_name* are undefined.

ERRORS
The **realpath()** function will fail if:

[EACCES]	Read or search permission was denied for a component of *file_name*.
[EINVAL]	Either the *file_name* or *resolved_name* argument is a null pointer.
[EIO]	An error occurred while reading from the file system.
[ELOOP]	Too many symbolic links were encountered in resolving *path*.
[ENAMETOOLONG]	The *file_name* argument is longer than **{PATH_MAX}** or a pathname component is longer than **{NAME_MAX}**.
[ENOENT]	A component of *file_name* does not name an existing file or *file_name* points to an empty string.
[ENOTDIR]	A component of the path prefix is not a directory.

The **realpath()** function may fail if:

[ENAMETOOLONG]	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds **{PATH_MAX}**.
[ENOMEM]	Insufficient storage space is available.

SEE ALSO
getcwd(3C), sysconf(2), <stdlib.h>.

CHANGE HISTORY
First released in Issue 4, Version 2.

NAME
redrawwin, wredrawln — line update status functions

SYNOPSIS
```
#include <curses.h>
int redrawwin(WINDOW *win);
int wredrawln(WINDOW *win, int beg_line, int num_lines);
```

DESCRIPTION
The **redrawwin()** and **wredrawln()** functions inform the implementation that some or all of the information physically displayed for the specified window may have been corrupted. The **redrawwin()** function marks the entire window; **wredrawln()** marks only *num_lines* lines starting at line number *beg_line*. The functions prevent the next refresh operation on that window from performing any optimisation based on assumptions about what is physically displayed there.

RETURN VALUE
Upon successful completion, these functions return OK. Otherwise they return ERR.

ERRORS
No errors are defined.

APPLICATION USAGE
The **redrawwin()** and **wredrawln()** functions could be used in a text editor to implement a command that redraws some or all of the screen.

SEE ALSO
clearok(3X), doupdate(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 4.

NAME

regcmp(), regex() - compile and execute regular expression

SYNOPSIS

```
#include <libgen.h>

char *regcmp(
     const char *string1,
     /* string2, */ ...
     /*, (char *)0 */
);

char *regex(const char *re, const char *subject, ...);

extern char *__loc1;
```

Remarks

The ANSI C "**, ...**" construct denotes a variable length argument list whose optional [or required] members are given in the associated comment (**/* */**).

Features documented in this manual entry are obsolescent and may be removed in a future HP-UX release. Use of *regcomp*(3C) instead is recommended.

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.

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 largely 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.

- Used within brackets the hyphen signifies a character range. For example, **[a-z]** is equivalent to **[abcd...xyz]**. The **-** can represent 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 can be applied. The value *m* is the minimum number and *u* is a maximum number, which must be no greater than 256. The syntax **{***m***}** indicates the exact number of times the regular expression can be applied. The syntax **{***m***,}** is analogous to {*m*,infinity}. The plus (**+**) and asterisk (*****) operations are equivalent to **{1,}** and **{0,}** respectively.

(...)$*n* The value of the enclosed regular expression is returned. The value is stored in the (*n*+1)th argument following the subject argument. A maximum of ten enclosed regular expressions are allowed. **regex()** makes its assignments unconditionally.

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

Since all of the above defined symbols are special characters, they must be escaped to be used as themselves.

EXAMPLES

Match a leading new-line in the subject string to which the *cursor* points.

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

Match through the string **Testing3** and return the address of the character after the last matched character (cursor+11). The string **Testing3** will be copied to the character array **ret0**.

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

WARNINGS

User programs that use **regcmp()** might run out of memory if **regcmp()** is called iteratively without freeing vectors that are no longer required.

SEE ALSO

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

NAME

regcomp(), regerror(), regexec(), regfree() - regular expression matching routines

SYNOPSIS

```
#include <regex.h>

int regcomp(regex_t *preg, const char *pattern, int cflags);

int regexec(
     const regex_t *preg,
     const char *string,
     size_t nmatch,
     regmatch_t pmatch[],
     int eflags
);

void regfree(regex_t *preg);

size_t regerror(
     int errcode,
     const regex_t *preg,
     char *errbuf,
     size_t errbuf_size
);
```

DESCRIPTION

These functions interpret regular expressions as described in *regexp*(5). They support both *basic* and *extended* regular expressions.

The structures **regex_t** and **regmatch_t** are defined in the header <**regex.h**>.

The **regex_t** structure contains at least the following member (use of other members results in non-portable code):

size_t re_nsub Number of parenthesized subexpressions.

The **regmatch_t** structure contains at least the following members:

regoff_t rm_so Byte offset from start of string to start of substring.

regoff_t rm_eo Byte offset from start of string to the first character after the end of the substring.

regcomp() compiles the regular expression specified by the *pattern* argument and places the results in the structure pointed to by *preg*. The *cflags* argument is the bit-wise logical OR of zero or more of the following flags (defined in <**regex.h**>):

REG_EXTENDED Use extended regular expressions.

REG_NEWLINE IF **REG_NEWLINE** is not set in *cflags*, a newline character in *pattern* or *string* is treated as an ordinary character. If **REG_NEWLINE** is set, newlines are treated as ordinary characters except as follows:

1. A newline in *string* is not matched by a period outside of a bracket expression or by any form of a nonmatching list.
2. A circumflex (**^**) in *pattern*, when used to specify expression anchoring, matches the zero-length string immediately after a newline in *string*, regardless of the setting of **REG_NOTBOL**.
3. A dollar-sign (**$**) in *pattern*, when used to specify expression anchoring, matches the zero-length string immediately before a newline in *string*, regardless of the setting of **REG_NOTEOL**.

REG_ICASE Ignore case in match. If a character in *pattern* is defined in the current **LC_CTYPE** locale as having one or more opposite-case counterpoints, both the character and any counterpoints match the pattern character. This applies to all portions of the pattern, including a string of characters specified to be matched via a back-reference expression (*****n*).

Within bracket expressions: Collation ranges, character classes, and equivalence classes are effectively expanded into equivalent lists of collation elements and characters. Opposite-case counterpoints are then generated for each collation element or character to form the complete matching list or non-matching list for the bracket expression. Opposite-case counterpoints for a multi-character collating element include all possible combinations of opposite-case counterpoints for each individual character comprising the collating element. These are then combined to form new valid multi-character collating elements. For example, the opposite-case counterpoints for **[.ch.]** could be **[.Ch.]**, **[.cH.]**, and **[.CH.]**.

The default regular expression type for *pattern* is Basic Regular Expression. The application can specify Extended Regular Expressions by using the **REG_EXTENDED** *cflags* value.

If the function **regcomp()** succeeds, it returns zero; otherwise it returns a non-zero value indicating the error.

If **regcomp()** succeeds, and if the **REG_NOSUB** flag was not set in *cflags*, **regcomp()** sets **re_nsub** to the number of parenthesized subexpressions (delimited by **\(** and **\)** in basic regular expressions or **(** and **)** in extended regular expressions) found in *pattern*.

regexec() matches the null-terminated string specified by *string* against the compiled regular expression *preg* initialized by a previous call to **regcomp()**. If it finds a match, **regexec()** returns zero; otherwise it returns non-zero indicating either no match or an error. The *eflags* argument is the bit-wise logical OR of the following flags:

REG_NOTBOL The first character of the string pointed to by *string* is not the beginning of the line. Therefore, the circumflex character (**^**), when taken as a special character, never matches.

REG_NOTEOL The last character of the string pointed to by *string* is not the end of the line. Therefore, the dollar sign (**$**), when taken as a special character, never matches.

If *nmatch* is not zero, and **REG_NOSUB** was not set in the *cflags* argument to **regcomp()**, then **regexec()** fills in the *pmatch* array with byte offsets to the substrings of *string* that correspond to the parenthesized subexpressions of *pattern*: *pmatch*[*i*].*rm_so* is the byte offset of the beginning and *pmatch*[*i*].*rm_eo* is the byte offset one byte past the end of the substring *i*. (Subexpression *i* begins at the *i*th matched left parenthesis, counting from 1). Offsets in *pmatch*[0] identify the substring that corresponds to the entire regular expression. Unused elements of *pmatch* are set to –1. If there are more than *nmatch* subexpressions in *pattern* (*pattern* itself counts as a subexpression), **regexec()** still does the match, but only records the first *nmatch* substrings.

When matching a regular expression, any given parenthesized subexpression of *pattern* might participate in the match of several different substrings of *string*, or it might not match any substring, even though the pattern as a whole did match. The following explains which substrings are reported in *pmatch* when matching regular expressions:

1. If subexpression *i* in a regular expression is not contained within another subexpression, and it participated in the match several times, the byte offsets in *pmatch*[*i*] delimit the last such match.
2. If subexpression *i* is not contained within another subexpression, and it did not participate in an otherwise successful match (because either *****, **?**, or **|** was used), then the byte offsets in *pmatch*[*i*] are –1.
3. If subexpression *i* is contained in subexpression *j*, and a match of subexpression *j* is reported in *pmatch*[*j*], the match or no-match reported in *pmatch*[*i*] is the last one that occurred within the substring in *pmatch*[*j*].
4. If subexpression *i* is contained in subexpression *j*, and the offsets in *pmatch*[*j*] are –1, the offsets in *pmatch*[*i*] will also be –1.
5. If subexpression *i* matched a zero-length string, both offsets in *pmatch*[*i*] refer to the character immediately following the zero-length substring.

If **REG_NOSUB** was set in *cflags* in the call to **regcomp()**, and *nmatch* is not zero in the call to **regexec()**, the content of the *pmatch* array is unspecified.

regfree() frees any memory allocated by **regcomp()** associated with *preg*.

If the *preg* argument to **regexec()** or **regfree()** is not a compiled regular expression returned by **regcomp()**, the result is undefined. A *preg* can no longer be treated as a compiled regular expression after it is given to **regfree()**.

regerror() provides a mapping from error codes returned by **regcomp()** and **regexec()** to printable strings. **regerror()** generates a string corresponding to the value of the *errcode* parameter, which was the last non-zero value returned by **regcomp()** or **regexec()** with the given value of *preg*. The *errcode* parameter can take on any of the error values defined in <**regex.h**>. If *errbuf_size* is not zero, **regerror()** copies an appropriate error message into the buffer specified by *errbuf*. If the error message (including the terminating null) cannot fit in the buffer, it is truncated to *errbuf_size* – 1 bytes and null terminated.

If *errbuf_size* is zero, the *errbuf* parameter is ignored, but the return value is as defined below.

regerror() returns the size of the buffer (including terminating null) that is required to hold the entire error message.

EXTERNAL INFLUENCES

Locale

The **LC_COLLATE** category determines the collating sequence used in compiling and executing regular expressions.

The **LC_CTYPE** category determines the interpretation of text as single and/or multi-byte characters, the characters matched by character-class expressions in regular expressions, and the opposite-case counterpart for each character.

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

regcomp() returns zero for success and non-zero for an invalid expression or other failure. **regexec()** returns zero if it finds a match and non-zero for no match or other failure.

ERRORS

If **regcomp()** or **regexec()** detects one of the error conditions listed below, it returns the corresponding non-zero error code. The error codes are defined in the header <**regex.h**>.

REG_BADBR	The contents within the pair **\{** (backslash left brace) and **\}** (backslash right brace) are unusable: not a number, number too large, more than two numbers, or first number larger than second.
REG_BADPAT	An invalid regular expression.
REG_BADRPT	The **?** (question mark), ***** (asterisk), or **+** (plus sign) symbols are not preceded by a valid regular expression.
REG_EBRACE	The use of a pair of **\{** (backslash left brace) and **\}** (backslash right brace) or **{}** (braces) is unbalanced.
REG_EBRACK	The use of **[]** (brackets) is unbalanced.
REG_EBOL	Using the **^** (caret) anchor and not beginning of line.
REG_ECHAR	There is an invalid multibyte character.
REG_ECOLLATE	There is an unusable collating element referenced.
REG_ECTYPE	There is an unusable character class type referenced.
REG_EEOL	Using the **$** (dollar) anchor and not end of line.
REG_EESCAPE	There is a trailing **** in the pattern.
REG_EPAREN	The use of a pair of **\(** (backslash left parenthesis) and **\)** (backslash right parenthesis) or **()** is unbalanced.
REG_ERANGE	There is an unusable endpoint in the range expression.
REG_ESPACE	There is insufficient memory space.
REG_ESUBREG	The number in *****digit* is invalid or in error.

REG_NOMATCH The **regexec()** function failed to match.

EXAMPLES

```
/* match string against the extended regular expression in pattern,
treating errors as no match.  Return 1 for match, 0 for no match.
Print an error message if an error occurs. */

int
match(string, pattern)
char *string;
char *pattern;
{
    int i;
    regex_t re;
    char buf[256];

    i=regcomp(&re, pattern, REG_EXTENDED|REG_NOSUB);
    if (i != 0) {
        (void)regerror(i,&re,buf,sizeof buf);
        printf("%s\n",buf);
        return(0);                          /* report error */
    }
    i = regexec(&re, string, (size_t) 0, NULL, 0);
    regfree(&re);
    if (i != 0) {
        (void)regerror(i,&re,buf,sizeof buf);
        printf("%s\n",buf);
        return(0);                          /* report error */
    }
    return(1);
}
```

The following demonstrates how the **REG_NOTBOL** flag could be used with **regexec()** to find all substrings in a line that match a pattern supplied by a user.

```
(void) regcomp(&re, pattern, 0);
/* look for first match at start of line */
error = regexec(&re, &buffer[0], 1, &pm, 0);
while (error == 0) {              /* while matches found */
    /* find next match on line */
    error = regexec(&re, &buffer[pm.rm_eo], 1, &pm, REG_NOTBOL);
}
```

AUTHOR

regcomp(), **regerror()**, **regexec()**, and **regfree()** were developed by OSF and HP.

SEE ALSO

regexp(5).

STANDARDS CONFORMANCE

regcomp(): XPG4, POSIX.2

regerror(): XPG4, POSIX.2

regexec(): XPG4, POSIX.2

regfree(): XPG4, POSIX.2

NAME

compile(), step(), advance() - regular expression compile and match routines

SYNOPSIS

```
#define INIT declarations
#define GETC() getc statements
#define PEEKC() peekc statements
#define UNGETC(c) ungetc statements
#define RETURN(pointer) return statements
#define ERROR(val) error statements

#include <regexp.h>

char *compile(
     char *instring,
     char *expbuf,
     const char *endbuf,
     int eof
);

int step(const char *string, const char *expbuf);

int advance(const char *string, const char *expbuf);

extern char *loc1, *loc2, *locs;

extern int circf, sed, nbra;
```

Remarks

Features documented in this manual entry are obsolescent and may be removed in a future HP-UX release. Use of *regcomp*(3C) functions instead is recommended.

DESCRIPTION

These functions are general-purpose regular expression matching routines to be used in programs that perform Basic Regular Expression (see *regexp*(5)) matching. These functions are defined in <**regexp.h**>.

The functions **step()** and **advance()** do pattern matching given a character string and a compiled regular expression as input. **compile()** takes a Basic Regular Expression as input and produces a compiled expression that can be used with **step()** and **advance()**.

The interface to this file is unpleasantly complex. Programs that include this file must have the following five macros declared before the **#include <regexp.h>** statement. These macros are used by the **compile()** routine.

GETC() Return the value of the next byte in the regular expression pattern. Successive calls to **GETC()** should return successive bytes of the regular expression.

PEEKC() Return the next byte in the regular expression. Successive calls to **PEEKC()** should return the same byte (which should also be the next byte returned by **GETC()**.

UNGETC(*c***)** Cause the argument *c* to be returned by the next call to **GETC()** (and **PEEKC()**). No more than one byte of pushback is ever needed, and this byte is guaranteed to be the last byte read by **GETC()**. The value of the macro **UNGETC(***c***)** is always ignored.

RETURN(*pointer***)**
This macro is used on normal exit of the **compile()** routine. The value of the argument *pointer* is a pointer to the character after the last character of the compiled regular expression. This is useful to programs that must manage memory allocation.

ERROR(*val***)** This is the abnormal return from the **compile()** routine. The argument *val* is an error number (see table below for meanings). This call should never return.

Error	*Meaning*
11	Range endpoint too large.
16	Bad number.
25	"*digit*" out of range.
36	Illegal or missing delimiter.
41	No remembered search string.
42	**\(** **\)** imbalance.

43 Too many **\(**.
44 More than 2 numbers given in **\{ \}**.
45 **}** expected after ****.
46 First number exceeds second in **\{ \}**.
49 **[ ]** imbalance.
50 Regular expression overflow.

The syntax of the **compile()** routine is as follows:

compile(*instring***,** *expbuf***,** *endbuf***,** *eof***)**

The first parameter *instring* is never used explicitly by the **compile()** routine, but is useful for programs that pass down different pointers to input characters. It is sometimes used in the **INIT** declaration (see below). Programs that call functions to input characters or have characters in an external array can pass down a value of **((char *) 0)** for this parameter.

The next parameter *expbuf* is a character pointer. It points to the location where the compiled regular expression will be placed.

The parameter *endbuf* is one more than the highest address where the compiled regular expression can be placed. If the compiled expression cannot fit in (*endbuf*–*expbuf*) bytes, a call to **ERROR(50)** is made.

The parameter *eof* is the character which marks the end of the regular expression. For example, in *ed*(1), this character is usually a **/**.

Each program that includes this file must have a **#define** statement for **INIT**. This definition is placed right after the declaration for the function **compile()** and the opening curly brace **{**. It is used for dependent declarations and initializations. Most often it is used to set a register variable to point to the beginning of the regular expression so that this register variable can be used in the declarations for **GETC()**, **PEEKC()**, and **UNGETC()**. Otherwise it can be used to declare external variables that might be used by **GETC()**, **PEEKC()**, and **UNGETC()**. See the example below of the declarations taken from *grep*(1).

step() also performs actual regular expression matching in this file. The call to *step* is as follows:

step(*string***,** *expbuf***)**

The first parameter to **step()** is a pointer to a string of characters to be checked for a match. This string should be null-terminated.

The second parameter *expbuf* is the compiled regular expression that was obtained by a call to **compile()**.

step() returns non-zero if the given string matches the regular expression, and zero if the expressions do not match. If there is a match, two external character pointers are set as a side effect to the call to **step()**. The variable set in **step()** is **loc1**. This is a pointer to the first character that matched the regular expression. The variable **loc2**, which is set by the function **advance()**, points to the character after the last character that matches the regular expression. Thus, if the regular expression matches the entire line, **loc1** points to the first character of *string* and **loc2** points to the null at the end of *string*.

step() uses the external variable **circf**, which is set by **compile()** if the regular expression begins with **^**. If this is set, **step()** tries to match the regular expression to the beginning of the string only. If more than one regular expression is to be compiled before the first is executed, the value of *circf* should be saved for each compiled expression and *circf* should be set to that saved value before each call to **step()**.

advance() is called from **step()** with the same arguments as **step()**. The purpose of **step()** is to step through the *string* argument and call **advance()** until **advance()** returns non-zero, which indicates a match, or until the end of *string* is reached. To constrain *string* to beginning-of-line in all cases, **step()** need not be called; simply call **advance()**.

When **advance()** encounters a ***** or **\{\}** sequence in the regular expression, it advances its pointer to the string to be matched as far as possible and recursively calls itself, trying to match the rest of the string to the rest of the regular expression. As long as there is no match, *advance* backs up along the string until it finds a match or reaches the point in the string that initially matched the ***** or **\{\}**. It is sometimes desirable to stop this backing up before the initial point in the string is reached. If the external character pointer **locs** is equal to the point in the string at sometime during the backing up process, **advance()** breaks out of the loop that backs up and returns zero. This is used by *ed*(1) and *sed*(1) for substitutions done globally (not just the first occurrence, but the whole line) so, for example, expressions such as **s/y*//g** do not loop forever.

The additional external variables **sed** and **nbra** are used for special purposes.

EXTERNAL INFLUENCES

Locale

The **LC_COLLATE** category determines the collating sequence used in compiling and executing regular expressions.

The **LC_CTYPE** category determines the interpretation of text as single and/or multi-byte characters, and the characters matched by character class expressions in regular expressions.

International Code Set Support

Single- and multi-byte character code sets are supported.

EXAMPLES

The following is an example of how the regular expression macros and calls look from *grep*(1):

```
#define INIT          register char *sp = instring;
#define GETC()        (*sp++)
#define PEEKC()       (*sp)
#define UNGETC(c)     (--sp)
#define RETURN(c)     return;
#define ERROR(c)      regerr()

#include <regexp.h>
    ...
(void) compile(*argv, expbuf, &expbuf[ESIZE], '\0');
    ...
if (step(linebuf, expbuf))
    succeed();
```

SEE ALSO

grep(1), regcomp(3C), setlocale(3C), regexp(5).

STANDARDS CONFORMANCE

advance(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

compile(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

loc1: AES, SVID2, SVID3, XPG2, XPG3, XPG4

loc2: AES, SVID2, SVID3, XPG2, XPG3, XPG4

locs: AES, SVID2, SVID3, XPG2, XPG3, XPG4

step(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

NAME

reltimer - relatively arm a per-process timer

SYNOPSIS

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

int reltimer(
     timer_t timerid,
     struct itimerspec *value,
     struct itimerspec *ovalue
);
```

DESCRIPTION

The **reltimer()** function sets the **it_value** of the specified timer to an offset from the current clock setting.

If **reltimer()** specifies a *value* argument with the **it_value** member equal to zero, the timer is disabled. **reltimer()** updates the *it_interval* value of the timer to the value specified. Time values smaller than the resolution of the specified timer are rounded up to its resolution; timer values larger than the maximum value of the specified timer are rounded down to the maximum value (see *mktimer*(3C)).

reltimer() returns in the *ovalue* parameter a value representing the previous amount of time before the timer would have expired or zero if the timer was disabled, together with the previous interval timer period. The members of *ovalue* are subject to the resolution of the timer, and are the same values that would be returned by a **gettimer()** call.

The behavior of this function is undefined if *value* is NULL.

APPLICATION USAGE

reltimer() is thread-safe. It is not async-cancel-safe.

RETURN VALUE

Upon successful completion, **reltimer()** returns zero; otherwise, it returns –1 and sets **errno** to indicate the error.

ERRORS

reltimer() fails if any of the following conditions are encountered:

[EINVAL] **timerid** does not correspond to an ID returned by **mktimer()** or the value structure specified a nanosecond value less than zero or greater than or equal to 1000 million.

[EIO] An error occurred while accessing the clock device.

FILES

/usr/include/sys/timers.h

SEE ALSO

timers(2), gettimer(3C), mktimer(3C).

STANDARDS CONFORMANCE

reltimer(): AES

NAME

remainder() - remainder function

SYNOPSIS

```
#include <math.h>
double remainder(double x, double y);
```

DESCRIPTION

The **remainder()** function returns the floating-point remainder $r = x - ny$ when y is a nonzero number. The value n is the integral value nearest the exact value x/y; when $| n - x/y | = ½$, the value n is chosen to be even.

The **remainder()** is independent of the rounding mode.

This function meets the requirement of the IEEE-754 standard for a remainder operation. The ISO/ANSI C committee has approved the **remainder()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

If y is ±INFINITY and x is not ±INFINITY, **remainder()** returns x.

If x is ±zero and y is a nonzero number, **remainder()** returns x.

If x or y is NaN, **remainder()** returns NaN.

If y is zero, **remainder()** returns NaN and sets **errno** to [EDOM].

If x is ±INFINITY, **remainder()** returns NaN and sets **errno** to [EDOM].

ERRORS

If **remainder()** fails, **errno** is set to one of the following values.

[EDOM]	y is zero.
[EDOM]	x is ±INFINITY.

SEE ALSO

fmod(3M), fabs(3M), remquo(3M), math(5).

STANDARDS CONFORMANCE

remainder(): SVID3, XPG4.2, IEEE-754

NAME
remove() - remove a file

SYNOPSIS
```
#include <stdio.h>
int remove(const char *path);
```

DESCRIPTION
remove() removes the file named by *path*. If *path* does not name a directory, **remove(***path***)** is equivalent to **unlink(***path***)**. If *path* names a directory, **remove(***path***)** is equivalent to **rmdir(***path***)**.

APPLICATION USAGE
remove() is thread-safe. It is not async-cancel-safe. A cancellation point may occur when a thread is executing **remove()**.

SEE ALSO
rmdir(2), unlink(2).

STANDARDS CONFORMANCE
remove(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

r

NAME

remquo() - remainder function with quotient

SYNOPSIS

```
#include <math.h>

double remquo(ouble x, double y, int *quo);
```

DESCRIPTION

The **remquo()** function computes the same remainder as the **remainder()** function. In the object pointed to by *quo* it stores a value whose sign is the sign of *x*/*y* and whose magnitude is congruent mod 2^n to the magnitude of the integral quotient of *x*/*y*, where *n* is an implementation-defined integer at least 3.

The ISO/ANSI C committee has approved the **remquo()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

The **remquo()** function returns *x* REM *y*.

ERRORS

No errors are defined.

SEE ALSO

fmod(3M), remainder(3M), math(5).

NAME
resetty, savetty — save/restore terminal mode

SYNOPSIS
```
#include <curses.h>
int resetty(void);
int savetty(void);
```

DESCRIPTION
The **resetty()** function restores the program mode as of the most recent call to **savetty()**.

The **savetty()** function saves the state that would be put in place by a call to **reset_prog_mode()**.

RETURN VALUE
Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS
No errors are defined.

SEE ALSO
def_prog_mode(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The entry is rewritten for clarity. The argument list for the **resetty()** and **savetty()** functions is explicitly declared as **void**.

r

NAME

res_query(), res_search(), res_mkquery(), res_send(), res_init(), dn_comp(), dn_expand(), herror() - resolver routines

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

ssize_t res_query(
     char *dname,
     int class,
     int type,
     u_char *answer,
     int anslen
);

ssize_t res_search(
     char *dname,
     int class,
     int type,
     u_char *answer,
     int anslen
);

ssize_t res_mkquery(
     int op,
     const char *dname,
     int class,
     int type,
     const char *data,
     int datalen,
     const char *newrr,
     char *buf,
     int buflen
);

ssize_t res_send(
     const char *msg,
     ssize_t msglen,
     char *answer,
     int anslen
);

int res_init();

ssize_t dn_comp(
     const char *exp_dn,
     u_char *comp_dn,
     ssize_t length,
     u_char **dnptrs,
     u_char **lastdnptr

);

ssize_t dn_expand(
     const u_char *msg,
     const u_char *eomorig,
     const u_char *comp_dn,
     u_char *exp_dn,
     int  length
);

void herror(char *s);
```

DESCRIPTION

These routines are used for making, sending, and interpreting query and reply messages with Internet domain name servers.

Global configuration and state information used by the resolver routines are kept in the structure **_res** and are defined in <**resolv.h**>. Most of the fields have reasonable defaults and can be ignored. The resolver options are stored in the **_res.options** field and are listed below. The options are stored as a simple bit mask containing the bitwise OR of the options enabled.

In a multithreaded environment, a thread specific **_res** structure is allocated for each thread.

RES_INIT True if the initial name server address and default domain name are initialized (i.e., **res_init()** has been called).

RES_DEBUG Print debugging messages.

RES_AAONLY Accept authoritative answers only. With this option, **res_send()** should continue until it finds an authoritative answer or finds an error. Currently this is not implemented.

RES_PRIMARY Query the primary server only. Currently this is not implemented.

RES_USEVC Use TCP connections for queries instead of UDP datagrams.

RES_STAYOPEN Used with **RES_USEVC** to keep the TCP connection open between queries. This is useful only in programs that regularly do many queries. UDP should be the normal mode used.

RES_IGNTC The name server will set the truncation bit if all of the data does not fit into the response datagram packet. If **RES_IGNTC** is set, **res_send()** will not retry the query with TCP (i.e., ignore truncation errors).

RES_RECURSE Set the recursion-desired bit in queries. This is the default. (**res_send()** does not do iterative queries and expects the name server to handle recursion.)

RES_DEFNAMES If set, **res_search()** appends the default domain name to single-component names (those that do not contain a dot). This option is enabled by default.

RES_DNSRCH If this option is set, **res_search()** searches for host names in the current domain and in parent domains; see *hostname*(5). This is used by the standard host lookup routine **gethostbyname()** (see *gethostent*(3N)). This option is enabled by default.

Initialization of the resolver structure normally occurs on the first call to one of the resolver routines below. If there are errors in the configuration file, they are silently ignored.

Primary Routines

res_init() Reads the configuration file, **/etc/resolv.conf**, to get the default domain name, search list, and the Internet address of the local name server(s). If no server is configured, the host running the resolver is tried. The current domain name is defined by the hostname if not specified in the configuration file; it can be overridden by the environment variable **LOCALDOMAIN**. This environment variable may contain several blank separated tokens and overrides the search list on a per process basis. This is similar to the **search** command in the configuration file. Another environment variable (**RES_OPTIONS**) can be set to override certain internal resolver options which are set by calling some of the configuration routines above, or by using the configuration file's **options** command. The syntax of the **RES_OPTIONS** environment variable is explained in *resolver*(4).

res_query() Provides an interface to the server query mechanism. It constructs a query, sends it to the local server, awaits a response, and makes preliminary checks on the reply. The query requests information of the specified *type* and *class* for the specified fully-qualified domain name *dname*. The reply message is left in the *answer* buffer with length *anslen* supplied by the caller.

res_search() Makes a query and awaits a response much like **res_query()**, but in addition, it implements the default and search rules controlled by the **RES_DEFNAMES** and **RES_DNSRCH** options. It returns the first successful reply.

Other Routines

Routines described here are lower-level routines used by **res_query()**.

res_mkquery() Constructs a standard query message and places it in *buf*. It returns the size of the query, or –1 if the query is larger than *buflen*. The query type *op* is usually **QUERY**, but can be any of the query types defined in <**arpa/nameser.h**>. The domain name for the query is given by *dname*. *class* can be any of the query classes defined in <**arpa/nameser.h**>. *type* can be any of the query types defined in <**arpa/nameser.h**>. *data* is the data for an inverse query (**IQUERY**). *newrr* is currently unused but is intended for making update messages.

res_send() Sends a pre-formatted query and returns an answer. It calls **res_init()** if **RES_INIT** is not set, sends the query to the local name server, and handles timeouts and retries. **res_send()** returns the length of the reply message, or –1 if there were errors.

dn_comp() Compresses the domain name *exp_dn* and stores it in *comp_dn*. The size of the compressed name is returned or –1 if there were errors. *length* is the size of the array pointed to by *comp_dn*. The compression uses an array of pointers *dnptrs* to previously compressed names in the current message. The first pointer points to to the beginning of the message and the list ends with NULL. The limit to the array is specified by *lastdnptr*. A side effect of **dn_comp()** is to update the list of pointers for labels inserted into the message as the name is compressed. If *dnptr* is NULL, names are not compressed. If *lastdnptr* is NULL, the list of labels is not updated.

dn_expand() Expands the compressed domain name *comp_dn* to a full domain name. The compressed name is contained in a query or reply message; *msg* is a pointer to the beginning of the message. The uncompressed name is placed in the buffer indicated by *exp_dn* which is of size *length*. The size of compressed name is returned or –1 if there was an error.

MULTITHREADED USAGE

Thread Safe: Yes
Cancel Safe: Yes
Fork Safe: No
Async-cancel Safe: No
Async-signal Safe: No

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancellation points.

RETURN VALUE

Error return status from **res_search()** is indicated by a return value of –1. The external integer **h_errno** can then be checked to see whether this is a temporary failure or an invalid or unknown host.

In a multithreaded application using kernel thread, a thread specific **h_errno** is allocated for each thread.

The routine **herror()** can be used to print an error message describing the failure. The argument string *s* is printed first, followed by a colon, a blank, the message, and a new-line.

ERRORS

h_errno can have the following values:

HOST_NOT_FOUND No such host is known.

TRY_AGAIN This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time may succeed.

NO_RECOVERY Some unexpected server failure was encountered. This is a non-recoverable error.

NO_DATA The name is known to the name server, but there is no data of the requested type associated with this name; this is not a temporary error. Another type of request to the name server using this domain name will result in an answer.

WARNINGS

h_errno is referenced as an **extern int** for non-threaded applications and is defined as function call macro for multithreaded applications in file **/usr/include/netdb.h**. Applications which reference **h_errno** should include **/usr/include/netdb.h**.

_res is referenced as an **extern struct _res_state** for non-threaded applications and is defined as function call macro for multithreaded application in file **/usr/include/resolv.h**. Applications which reference **_res** should include **/usr/include/resolv.h**.

AUTHOR

These resolver routines were developed by the University of California, Berkeley.

FILES

/etc/resolv.conf Resolver configuration file.

SEE ALSO

named(1M), gethostent(3N), resolver(4), hostname(5), RFC1034, RFC1035, RFC1535.

NAME

rexec() - return stream to a remote command

SYNOPSIS

```
int rexec(char **ahost,
          int inport,
          const char *user,
          const char *passwd,
          const char *cmd,
          int *fd2p);
```

DESCRIPTION

The **rexec()** routine performs the necessary tasks to arrange for the remote execution of *cmd* on the remote host ***ahost** as *user*, who is authenticated with *passwd*. Upon completion of authentication, a file descriptor is returned for the socket to which standard input and standard output of *cmd* are attached. A command-level interface to **rexec()** is provided by the **rexec** command (see *remsh*(1)).

When invoked, **rexec()** looks up host ***ahost** using **gethostbyname()** (see *gethostent*(3N)) and returns –1 if the host does not exist. The host name can be either the official name or an alias. If the **gethostbyname()** call succeeds, ***ahost** is set to the standard name of the host. Next, **rexec()** passes a user name and password to the remote host for authentication, as specified in the *user* and *passwd* parameters to **rexec()**. If either *user* or *passwd* are NULL, **rexec()** searches for the appropriate information in the **.netrc** file (see *netrc*(4)) in the user's home directory. If no *user* or *passwd* are found, **rexec()** prompts the user for the remote user name and password, defaulting to the local user name and a NULL password.

The *inport* variable specifies which TCP port to use for the connection; it is normally the value returned by the following call to **getservbyname :**

```
getservbyname("exec", "tcp")
```

(see *getservent*(3N)). The protocol used by **rexec()** is described in detail in *rexecd*(1M).

If the call succeeds, a socket of type **SOCK_STREAM** is returned to the caller, and given to the remote command as **stdin** and **stdout**. If connection to a socket is denied after five tries, or for some other reason (other than the port is in use), **rexec()** returns –1. If *fd2p* is non-zero, an auxiliary connection to a control process is set up and a file descriptor for it is placed in ***fd2p**. The control process returns diagnostic output from the command on this connection and accepts bytes on this connection, interpreting them as UNIX signal numbers to be forwarded to the process group of the command. If the auxiliary port cannot be set up, **rexec()** returns –1. If *fd2p* is 0, **stderr** of the remote command is made the same as **stdout** and no provision is made for sending arbitrary signals to the remote process.

DIAGNOSTICS

When invoked, **rexec()** produces the following diagnostic messages:

hostname: **Unknown host**

The remote host name was not found by **gethostbyname()**.

system call: *message*

Error in executing the *system call*. The *message* specifies the cause of the failure.

connect: *hostname*: *message*

Error in connecting to the socket obtained for **rexec()**. The *message* specifies the cause of the failure.

Secondary socket: *message*

Error in creating a secondary socket for error transmission to be used by **rexec()**.

read: *hostname*: *message*

Error in reading information transmitted over the socket. The *message* specifies the cause of the failure.

Connection timeout

The remote host did not connect within 30 seconds to the secondary socket set up as an error connection.

Lost connection

The program attempts to read from the socket and fails. This means the socket connection with the remote host was lost.

.netrc: *message*

Error in opening **.netrc** file in the home directory for a reason other than the file not existing.

Error - .netrc file not correct mode. Remove password or correct mode.

The **.netrc** file is readable, writable or executable by someone other than the owner.

Next step: Check whether **.netrc** has been modified by someone else and change the mode of **.netrc** (**chmod 400 .netrc**).

Unknown .netrc option *keyword*

An unrecognized keyword has been found in **.netrc** (see *netrc*(4)).

Next step: Correct the keyword in **.netrc**.

primary connection shutdown

While waiting for the secondary socket to be set up, **rexec()** had its primary connection shut down. This may have been caused by an **inetd** security failure (see *inetd*(1M)).

recv: *message*

While trying to set up the secondary (**stderr**) socket, **rexec()** had an error condition on its primary connection.

accept: Interrupted system call

While trying to set up a secondary socket, **rexec()** ran out of a resource, which caused the accept to time out.

Next step: Repeat the command. If a timeout occurs, check whether the Internet Services are installed and **inetd** is running.

EXAMPLE

To execute the **date** command on remote host **hpxzgy** using the remote account **chm**, **rexec()** is invoked as follows:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

char *host[] = { "hpxzgy" };
char *user = "chm";
char *passwd = "password";
char *cmd = "date";

main(argc, argv)
    char **argv;
    int argc;
{
    char ch;
    struct servent *servent;
    FILE *fp;
    int sd;

    servent = getservbyname("exec", "tcp");
    sd = rexec(host, servent->s_port, user, passwd, cmd, 0);
    fp = fdopen(sd, "r");
    while ((ch = getc(fp)) != EOF)
        putchar(ch);
}
```

WARNINGS

There is no way to specify options to the **socket()** call that **rexec()** makes.

A program using **rexec()** should not be put in the background when **rexec()** is expected to prompt for a password or user name. Putting **rexec()** in the background will cause it to compete with the current shell process for input.

Since **rexec()** replaces the pointer to the host name (*****_ahost_) with a pointer to the standard name of the host in a static data area, this value must be copied into the user's data area if it is to be used later.

The password is sent unencrypted through the socket connection.

AUTHOR

rexec() was developed by the University of California, Berkeley.

SEE ALSO

remsh(1), inetd(1M), rexecd(1M), gethostent(3N), getservent(3N), rcmd(3N), netrc(4).

r

NAME

rint(), nearbyint() - round to nearest int function

SYNOPSIS

```
#include <math.h>
double rint(double x);
double nearbyint(double x);
```

DESCRIPTION

rint() and **nearbyint()** return the integer (represented as a double precision number) nearest *x* according to the current rounding direction mode. These functions meet the requirement of the IEEE-754 standard for an operation that rounds to integer in floating-point format.

The two functions are identical except that **rint()** raises the inexact exception, while **nearbyint()** does not.

In the default rounding direction (round to nearest), **rint(***x***)** is the integer nearest *x* with the additional stipulation that if $|\mathbf{rint}(x) - x| = 1/2$, then **rint(***x***)** is even. (The same statement applies to **nearbyint(***x***).**)

If the current rounding direction is toward negative infinity, **rint()** and **nearbyint()** are identical to **floor()**. If the current rounding direction is toward positive infinity, **rint()** and **nearbyint()** are identical to **ceil()**.

The ISO/ANSI C committee has approved the **rint()** and **nearbyint()** functions for inclusion in the C9X draft standard.

To use these functions, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

If *x* is ±INFINITY, the **rint()** and **nearbyint()** functions return ±INFINITY respectively.

If *x* is NaN, the **rint()** and **nearbyint()** functions return NaN.

ERRORS

No errors are defined.

SEE ALSO

ceil(3M), floor(3M), fabs(3M), fmod(3M), fegetround(3M), fesetround(3M), lrint(3M), llrint(3M), lround(3M), llround(3M), round(3M), trunc(3M), math(5), fenv(5).

STANDARDS CONFORMANCE

rint(): XPG4.2, IEEE-754

NAME

ripoffline — reserve a line for a dedicated purpose

SYNOPSIS

```
#include <curses.h>
int ripoffline(int line, int (*init)(WINDOW *win, int columns));
```

DESCRIPTION

The **ripoffline()** function reserves a screen line for use by the application.

Any call to **ripoffline()** must precede the call to **initscr()** or **newterm()**. If *line* is positive, one line is removed from the beginning of *stdscr*; if *line* is negative, one line is removed from the end. Removal occurs during the subsequent call to **initscr()** or **newterm()**. When the subsequent call is made, the function pointed to by *init* is called with two arguments: a **WINDOW** pointer to the one-line window that has been allocated and an integer with the number of columns in the window. The initialisation function cannot use the **LINES** and **COLS** external variables and cannot call **wrefresh()** or **doupdate()**, but may call **wnoutrefresh()**.

Up to five lines can be ripped off. Calls to **ripoffline()** above this limit have no effect but report success.

RETURN VALUE

The **ripoffline()** function returns OK.

ERRORS

No errors are defined.

APPLICATION USAGE

Calling **slk_init()** reduces the size of the screen by one line if **initscr()** eventually uses a line from *stdscr* to emulate the soft labels. If **slk_init()** rips off a line, it thereby reduces by one the number of lines an application can reserve by subsequent calls to **ripoffline()**. Thus, portable applications that use soft label functions should not call **ripoffline()** more than four times.

When **initscr()** or **newterm()** calls the initialisation function pointed to by *init*, the implementation may pass NULL for the **WINDOW** pointer argument *win*. This indicates inability to allocate a one-line window for the line that the call to **ripoffline()** ripped off. Portable applications should verify that *win* is not NULL before performing any operation on the window it represents.

SEE ALSO

doupdate(3X), initscr(3X), slk_attroff(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME
rmtimer - free a per-process timer

SYNOPSIS
```
#include <sys/timers.h>
int rmtimer(timer_t timerid);
```

DESCRIPTION
The **rmtimer()** function is used to free a previously allocated timer (returned by **mktimer()**. Any pending timer event to be generated by this timer has been canceled when the call returns.

APPLICATION USAGE
rmtimer() is thread-safe. It is not async-cancel-safe.

RETURN VALUE
Upon successful completion, **rmtimer()** returns zero; otherwise, it returns –1 and sets **errno** to indicate the error.

ERRORS
rmtimer() fails if the following condition is encountered:

[EINVAL] **timerid** is not a valid timer ID.

FILES
/usr/include/sys/timers.h

SEE ALSO
timers(2), mktimer(3C), reltimer(3C).

STANDARDS CONFORMANCE
rmtimer(): AES

r

NAME

rnusers(), rusers() - return information about users on remote machines

SYNOPSIS

```
#include <utmp.h>
#include <rpcsvc/rusers.h>

int rnusers(char *host);

int rusers(char *host, struct utmpidlearr *up);
```

DESCRIPTION

rnusers() returns the number of users logged in on *host* or –1 if it cannot determine that number. The *host* string is either the official name of the host or an alias for it. See *hosts*(4) for more information regarding host names.

rusers() fills in the **utmpidlearr** structure with data about *host* and returns 0 if successful. The **ut_line** field is limited to eight characters on Berkeley systems, so the HP-UX XDR routine truncates from 12 characters to 8. The **nonuser()** macro does not exist in the HP-UX **utmp.h** file; therefore, HP-UX windows appear as separate users.

The relevant structures are:

```
struct utmparr {            /* RUSERSVERS_ORIG */
        struct utmp **uta_arr;
        int uta_cnt;
};

struct utmpidle {
        struct utmp ui_utmp;
        unsigned ui_idle;
};

struct utmpidlearr {        /* RUSERSVERS_IDLE */
        struct utmpidle **uia_arr;
        int uia_cnt;
};
```

RPC Information

Program number:

```
RUSERSPROG
```

XDR routines:

```
int xdr_utmp(xdrs, up)
        XDR *xdrs;
        struct utmp *up;
int xdr_utmpidle(xdrs, ui)
        XDR *xdrs;
        struct utmpidle *ui;
int xdr_utmpptr(xdrs, up)
        XDR *xdrs;
        struct utmp **up;
int xdr_utmpidleptr(xdrs, up)
        XDR *xdrs;
        struct utmpidle **up;
int xdr_utmparr(xdrs, up)
        XDR *xdrs;
        struct utmparr *up;
int xdr_utmpidlearr(xdrs, up)
        XDR *xdrs;
        struct utmpidlearr *up;
```

Procs:

RUSERSPROC_NUM
No arguments, returns number of users as an *unsigned long*.

RUSERSPROC_NAMES
No arguments, returns *utmparr* or *utmpidlearr*, depending on version number.

RUSERSPROC_ALLNAMES
No arguments, returns *utmparr* or *utmpidlearr*, depending on version number. Returns listing even for *utmp* entries satisfying *nonuser()* in *utmp.h*.

Versions:

RUSERSVERS_ORIG
RUSERSVERS_IDLE

Structures:

WARNINGS

User applications that call this routine must be linked with **/usr/lib/librpcsvc.a**. For example,

```
cc my_source.c -lrpcsvc
```

AUTHOR

rnusers() was developed by Sun Microsystems, Inc.

SEE ALSO

rusers(1).

r

NAME

round() - round function

SYNOPSIS

```
#include <math.h>
double round(double x);
```

DESCRIPTION

round() rounds its argument to the nearest integral value in floating-point format. An argument exactly halfway between two integers is rounded away from zero, regardless of the current rounding direction. Rounding away from zero also applies to the functions, **lround** and **llround**.

The ISO/ANSI C committee has approved the **round()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

The **round()** function returns the rounded integral value.

ERRORS

No errors are defined.

SEE ALSO

ceil(3M), floor(3M), fabs(3M), fmod(3M), fegetround(3M), fesetround(3M), lrint(3M), llrint(3M), lround(3M), llround(3M), rint(3M), trunc(3M), math(5), fenv(5).

r

NAME

rpc - library routines for remote procedure calls

SYNOPSIS

cc [*flag* ...] *file* ... **-lnsl** [*library* ...]

```
#include <rpc/rpc.h>
#include <netconfig.h>
```

DESCRIPTION

These routines allow C language programs to make procedure calls on other machines across a network. First, the client sends a request to the server. On receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends back a reply.

All RPC routines require the header **<rpc/rpc.h>**. Routines that take a **netconfig** structure also require that **<netconfig.h>** be included. Applications using RPC and XDR routines should be linked with the **libnsl** library.

Multithread Considerations

In the case of multithreaded applications, the **_REENTRANT** flag must be defined on the command line at compilation time (**-D_REENTRANT**). Defining this flag enables a thread-specific version of **rpc_createerr** (see *rpc_clnt_create*(3N)).

Client-side routines are MT-Safe. CLIENT handles (see *rpc_clnt_create*(3N)) can be shared between threads, however in this implementation requests by different threads are serialized (that is, the first request will receive its results before the second request is sent).

Server-side routines are mostly MT-Unsafe. In this implementation the service transport handle, SVCXPRT (see *rpc_svc_create*(3N)), contains a single data area for decoding arguments and encoding results. Therefore, this structure cannot be freely shared between threads that call functions that do this. Routines that are affected by this restriction are marked as unsafe for MT applications (see *rpc_svc_calls*(3N)).

Nettype

Some of the high-level RPC interface routines take a *nettype* string as one of the parameters (for example, **clnt_create()**, **svc_create()**, **rpc_reg()**, **rpc_call()**). This string defines a class of transports which can be used for a particular application.

nettype can be one of the following:

netpath Choose from the transports which have been indicated by their token names in the **NETPATH** environment variable. If **NETPATH** is unset or **NULL**, it defaults to **visible**. **netpath** is the default *nettype*.

visible Choose the transports which have the visible flag (**v**) set in the **/etc/netconfig** file.

circuit_v This is same as **visible** except that it chooses only the connection oriented transports (semantics **tpi_cots** or **tpi_cots_ord**) from the entries in the **/etc/netconfig** file.

datagram_v This is same as **visible** except that it chooses only the connectionless datagram transports (semantics **tpi_clts**) from the entries in the **/etc/netconfig** file.

circuit_n This is same as **netpath** except that it chooses only the connection oriented datagram transports (semantics **tpi_cots** or **tpi_cots_ord**).

datagram_n This is same as **netpath** except that it chooses only the connectionless datagram transports (semantics **tpi_clts**).

udp This refers to Internet UDP.

tcp This refers to Internet TCP.

If *nettype* is **NULL**, it defaults to **netpath**. The transports are tried in left to right order in the **NETPATH** variable or in top to down order in the **/etc/netconfig** file.

Data Structures

Some of the data structures used by the RPC package are shown below.

The AUTH Structure

```
union des_block {
   struct {
      u_int32 high;
      u_int32 low;
   } key;
   char c[8];
};
typedef union des_block des_block;
extern bool_t xdr_des_block( );

/*
* Authentication info. Opaque to client.
*/
struct opaque_auth {
   enum_t   oa_flavor;   /* flavor of auth */
   caddr_t  oa_base;     /* address of more auth stuff */
   u_int    oa_length;   /* not to exceed MAX_AUTH_BYTES */
};

/*
* Auth handle, interface to client side .B authenticators.
*/
typedef struct {
   struct    opaque_auth   ah_cred;
   struct    opaque_auth   ah_verf;
   union     des_block     ah_key;
   struct auth_ops {
      void   (*ah_nextverf)( );
      int    (*ah_marshal)( );   /* nextverf & serialize */
      int    (*ah_validate)( );  /* validate verifier */
      int    (*ah_refresh)( );   /* refresh credentials */
      void   (*ah_destroy)( );   /* destroy this structure */
   } *ah_ops;
   caddr_t ah_private;
} AUTH;
```

The CLIENT Structure

```
/*
* Client rpc handle.
* Created by individual implementations.
* Client is responsible for initializing auth.
*/
typedef struct {
   AUTH%*cl_auth;                       /* authenticator */
   struct clnt_ops {
     enum clnt_stat  (*cl_call)( );    /* call remote procedure */
     void            (*cl_abort)( );   /* abort a call */
     void            (*cl_geterr)( );  /* get specific error code */
     bool_t          (*cl_freeres)( ); /* frees results */
     void            (*cl_destroy)( ); /* destroy this structure */
     bool_t          (*cl_control)( ); /* the ioctl( ) of rpc */
   } *cl_ops;
   caddr_t           cl_private;        /* private stuff */
   char              *cl_netid;         /* network identifier */
   char              *cl_tp;            /* device name */
} CLIENT;
```

The SVCXPRT Structure

```
enum xprt_stat {
  XPRT_DIED,
  XPRT_MOREREQS,
  XPRT_IDLE
```

```
};
/*
* Server side transport handle
*/
typedef struct {
   int%xp_fd;                          /* file descriptor for the
                                                server handle */
   u_short            xp_port;         /* obsolete */
   struct xp_ops {
      bool_t          (*xp_recv)();    /* receive incoming requests */
      enum xprt_stat  (*xp_stat)();    /* get transport status */
      bool_t          (*xp_getargs)(); /* get arguments */
      bool_t          (*xp_reply)();   /* send reply */
      bool_t          (*xp_freeargs)(); /* free mem allocated
                                                for args */
      void            (*xp_destroy)(); /* destroy this struct */
   } *xp_ops;
   int                xp_addrlen;      /* length of remote addr.
                                                Obsolete */
   char               *xp_tp;          /* transport provider device
                                                name */
   char               *xp_netid;       /* network identifier */
   struct netbuf      xp_ltaddr;       /* local transport address */
   struct netbuf      xp_rtaddr;       /* remote transport address */
   char               xp_raddr[16];    /* remote address. Obsolete */
   struct opaque_auth xp_verf;         /* raw response verifier */
   caddr_t            xp_p1;           /* private: for use
                                                by svc ops */
   caddr_t            xp_p2;           /* private: for use
                                                by svc ops */
   caddr_t            xp_p3;           /* private: for use
                                                by svc lib */
   int                xp_type          /* transport type */
} SVCXPRT;
```

The svc_reg Structure

```
struct svc_req {
   u_long             rq_prog;         /* service program number */
   u_long             rq_vers;         /* service protocol version */
   u_long             rq_proc;         /* the desired procedure */
   struct opaque_auth rq_cred;         /* raw creds from the wire */
   caddr_t            rq_clntcred;     /* read only cooked cred */
   SVCXPRT            *rq_xprt;        /* associated transport */
};
```

The XDR Structure

```
/*
 * XDR operations.
 * XDR_ENCODE causes the type to be encoded into the stream.
 * XDR_DECODE causes the type to be extracted from the stream.
 * XDR_FREE can be used to release the space allocated by an XDR_DECODE
 * request.
 */
enum xdr_op {
    XDR_ENCODE=0,
    XDR_DECODE=1,
    XDR_FREE=2
};
/*
 * This is the number of bytes per unit of external data.
 */
```

```
#define BYTES_PER_XDR_UNIT    (4)
#define RNDUP(x)  ((((x) + BYTES_PER_XDR_UNIT - 1) /
                   BYTES_PER_XDR_UNIT) \ * BYTES_PER_XDR_UNIT)

/*
 * A xdrproc_t exists for each data type which is to be encoded or
 * decoded.  The second argument to the xdrproc_t is a pointer to
 * an opaque pointer.  The opaque pointer generally points to a
 * structure of the data type to be decoded.  If this points to 0,
 * then the type routines should allocate dynamic storage of the
 * appropriate size and return it.
 * bool_t  (*xdrproc_t)(XDR *, caddr_t *);
 */
typedef  bool_t (*xdrproc_t)();

/*
 * The XDR handle.
 * Contains operation which is being applied to the stream,
 * an operations vector for the particular implementation
 */
typedef struct {
   enum xdr_op      x_op;   /* operation; fast additional param */
   struct xdr_ops {
    bool_t  (*x_getlong)();  /* get a long from underlying stream */
    bool_t  (*x_putlong)();  /* put a long to underlying stream */
    bool_t  (*x_getbytes)(); /* get bytes from underlying stream */
    bool_t  (*x_putbytes)(); /* put bytes to underlying stream */
    u_int   (*x_getpostn)(); /* returns bytes off from beginning */
    bool_t  (*x_setpostn)(); /* lets you reposition the stream */
    long *  (*x_inline)();   /* buf quick ptr to buffered data */
    void    (*x_destroy)();  /* free privates of this xdr_stream */
   } *x_ops;
   caddr_t  x_public;        /* users' data */
   caddr_t  x_private;       /* pointer to private data */
   caddr_t  x_base;          /* private used for position info */
   int      x_handy;         /* extra private word */
} XDR;
```

Index to Routines

The following table lists RPC routines and the manual reference pages on which they are described:

RPC Routine	Manual Reference Page
auth_destroy	*rpc_clnt_auth*(3N)
authdes_create	*rpc_soc*(3N)
authdes_getucred	*secure_rpc*(3N)
authdes_seccreate	*secure_rpc*(3N)
authnone_create	*rpc_clnt_auth*(3N)
authsys_create	*rpc_clnt_auth*(3N)
authsys_create_default	*rpc_clnt_auth*(3N)
authunix_create	*rpc_soc*(3N)
authunix_create_default	*rpc_soc*(3N)
callrpc	*rpc_soc*(3N)
clnt_broadcast	*rpc_soc*(3N)
clnt_call	*rpc_clnt_calls*(3N)
clnt_control	*rpc_clnt_create*(3N)
clnt_create	*rpc_clnt_create*(3N)
clnt_destroy	*rpc_clnt_create*(3N)
clnt_dg_create	*rpc_clnt_create*(3N)
clnt_freeres	*rpc_clnt_calls*(3N)
clnt_geterr	*rpc_clnt_calls*(3N)
clnt_pcreateerror	*rpc_clnt_create*(3N)
clnt_perrno	*rpc_clnt_calls*(3N)

clnt_perror	*rpc_clnt_calls*(3N)
clnt_raw_create	*rpc_clnt_create*(3N)
clnt_spcreateerror	*rpc_clnt_create*(3N)
clnt_sperrno	*rpc_clnt_calls*(3N)
clnt_sperror	*rpc_clnt_calls*(3N)
clnt_tli_create	*rpc_clnt_create*(3N)
clnt_tp_create	*rpc_clnt_create*(3N)
clnt_udpcreate	*rpc_soc*(3N)
clnt_vc_create	*rpc_clnt_create*(3N)
clntraw_create	*rpc_soc*(3N)
clnttcp_create	*rpc_soc*(3N)
clntudp_bufcreate	*rpc_soc*(3N)
get_myaddress	*rpc_soc*(3N)
getnetname	*secure_rpc*(3N)
host2netname	*secure_rpc*(3N)
key_decryptsession	*secure_rpc*(3N)
key_encryptsession	*secure_rpc*(3N)
key_gendes	*secure_rpc*(3N)
key_setsecret	*secure_rpc*(3N)
netname2host	*secure_rpc*(3N)
netname2user	*secure_rpc*(3N)
pmap_getmaps	*rpc_soc*(3N)
pmap_getport	*rpc_soc*(3N)
pmap_rmtcall	*rpc_soc*(3N)
pmap_set	*rpc_soc*(3N)
pmap_unset	*rpc_soc*(3N)
registerrpc	*rpc_soc*(3N)
rpc_broadcast	*rpc_clnt_calls*(3N)
rpc_broadcast_exp	*rpc_clnt_calls*(3N)
rpc_call	*rpc_clnt_calls*(3N)
rpc_reg	*rpc_svc_calls*(3N)
svc_create	*rpc_svc_create*(3N)
svc_destroy	*rpc_svc_create*(3N)
svc_dg_create	*rpc_svc_create*(3N)
svc_dg_enablecache	*rpc_svc_calls*(3N)
svc_fd_create	*rpc_svc_create*(3N)
svc_fds	*rpc_soc*(3N)
svc_freeargs	*rpc_svc_reg*(3N)
svc_getargs	*rpc_svc_reg*(3N)
svc_getcaller	*rpc_soc*(3N)
svc_getreq	*rpc_soc*(3N)
svc_getreqset	*rpc_svc_calls*(3N)
svc_getrpccaller	*rpc_svc_calls*(3N)
svc_raw_create	*rpc_svc_create*(3N)
svc_reg	*rpc_svc_calls*(3N)
svc_register	*rpc_soc*(3N)
svc_run	*rpc_svc_reg*(3N)
svc_sendreply	*rpc_svc_reg*(3N)
svc_tli_create	*rpc_svc_create*(3N)
svc_tp_create	*rpc_svc_create*(3N)
svc_unreg	*rpc_svc_calls*(3N)
svc_unregister	*rpc_soc*(3N)
svc_vc_create	*rpc_svc_create*(3N)
svcerr_auth	*rpc_svc_err*(3N)
svcerr_decode	*rpc_svc_err*(3N)
svcerr_noproc	*rpc_svc_err*(3N)
svcerr_noprog	*rpc_svc_err*(3N)
svcerr_progvers	*rpc_svc_err*(3N)
svcerr_systemerr	*rpc_svc_err*(3N)
svcerr_weakauth	*rpc_svc_err*(3N)
svcfd_create	*rpc_soc*(3N)

r

svcraw_create	*rpc_soc*(3N)
svctcp_create	*rpc_soc*(3N)
svcudp_bufcreate	*rpc_soc*(3N)
svcudp_create	*rpc_soc*(3N)
user2netname	*secure_rpc*(3N)
xdr_accepted_reply	*rpc_xdr*(3N)
xdr_authsys_parms	*rpc_xdr*(3N)
xdr_authunix_parms	*rpc_soc*(3N)
xdr_callhdr	*rpc_xdr*(3N)
xdr_callmsg	*rpc_xdr*(3N)
xdr_opaque_auth	*rpc_xdr*(3N)
xdr_rejected_reply	*rpc_xdr*(3N)
xdr_replymsg	*rpc_xdr*(3N)
xprt_register	*rpc_svc_calls*(3N)
xprt_unregister	*rpc_svc_calls*(3N)

MULTITHREAD USAGE

See the MULTITHREAD USAGE section of the routines respective manual pages.

FILES

/etc/netconfig

SEE ALSO

getnetconfig(3N), getnetpath(3N), rpc_clnt_auth(3N), rpc_clnt_calls(3N), rpc_clnt_create(3N), rpc_svc_calls(3N), rpc_svc_create(3N), rpc_svc_err(3N), rpc_svc_reg(3N), rpc_xdr(3N), rpcbind(3N), secure_rpc(3N), xdr(3N), netconfig(4), rpc(4), environ(5).

r

NAME

rpc_clnt_auth, auth_destroy, authnone_create, authsys_create, authsys_create_default - library routines for client side remote procedure call authentication

SYNOPSIS

```
#include <rpc/rpc.h>

void auth_destroy(AUTH *auth);

AUTH *authnone_create(void);

AUTH *authsys_create(const char *host, const uid_t uid, const gid_t gid,
    const int len, const gid_t *aup_gids);

AUTH *authsys_create_default(void);
```

DESCRIPTION

These routines are part of the RPC library that allows C language programs to make procedure calls on other machines across the network, with desired authentication.

These routines are normally called after creating the **CLIENT** handle. The **cl_auth** field of the **CLIENT** structure should be initialized by the **AUTH** structure returned by some of the following routines. The client's authentication information is passed to the server when the RPC call is made.

Only the **NULL** and the **SYS** style of authentication is discussed here. For the **DES** style authentication, please refer to *secure_rpc*(3N).

The **NULL** and **SYS** style of authentication are safe in multithreaded applications. For the MT-level of the **DES** see it respective manual page.

Routines

The following routines require that the header <**rpc/rpc.h**> be included (see *rpc*(3N) for the definition of the **AUTH** data structure).

void auth_destroy()
A function macro that destroys the authentication information associated with *auth*. Destruction usually involves deallocation of private data structures. The use of *auth* is undefined after calling **auth_destroy()**.

AUTH *authnone_create()
Create and return an RPC authentication handle that passes nonusable authentication information with each remote procedure call. This is the default authentication used by RPC.

AUTH *authsys_create()
Create and return an RPC authentication handle that contains **AUTH_SYS** authentication information. The parameter *host* is the name of the machine on which the information was created; *uid* is the user's user ID; *gid* is the user's current group ID; *len* and *aup_gids* refer to a counted array of groups to which the user belongs.

AUTH *authsys_create_default()
Call **authsys_create()** with the appropriate parameters.

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

SEE ALSO

rpc(3N), rpc_clnt_calls(3N), rpc_clnt_create(3N), secure_rpc(3N).

NAME

rpc_clnt_calls, clnt_call, clnt_freeres, clnt_geterr, clnt_perrno, clnt_perror, clnt_sperrno, clnt_sperror, rpc_broadcast, rpc_broadcast_exp, rpc_call - library routines for client side calls

SYNOPSIS

```
#include <rpc/rpc.h>

enum clnt_stat clnt_call(CLIENT *clnt, const u_long procnum,
    const xdrproc_t inproc, const caddr_t in, const xdrproc_t outproc,
    caddr_t out, const struct timeval tout);

bool_t clnt_freeres(CLIENT *clnt, const xdrproc_t outproc, caddr_t out);

void clnt_geterr(const CLIENT *clnt, struct rpc_err *errp);

void clnt_perrno(const enum clnt_stat stat);

void clnt_perror(const CLIENT *clnt, const char *s);

char *clnt_sperrno(const enum clnt_stat stat);

char *clnt_sperror(const CLIENT *clnt, const char *s);

enum clnt_stat rpc_broadcast(const u_long progrnum, const u_long versnum,
    const u_long procnum, const xdrproc_t inproc, const caddr_t in,
    const xdrproc_t outproc, caddr_t out, const resultproc_t eachresult,
    const char *nettype);

enum clnt_stat rpc_broadcast_exp(const u_long progrnum,
    const u_long versnum, const u_long procnum, const xdrproc_t xargs,
    caddr_t argsp, const xdrproc_t xresults, caddr_t resultsp,
    const int inittime, const int waittime,
    const resultproc_t eachresult, const char *nettype);

enum clnt_stat rpc_call(const char *host, const u_long progrnum,
    const u_long versnum, const u_long procnum, const xdrproc_t inproc,
    const char *in, const xdrproc_t outproc, char *out, const char *nettype);
```

DESCRIPTION

RPC library routines allow C language programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a request to the server. Upon receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends back a reply.

The **clnt_call()**, **rpc_call()**, and **rpc_broadcast()** routines handle the client side of the procedure call. The remaining routines deal with error handling in the case of errors.

Some of the routines take a **CLIENT** handle as one of the parameters. A **CLIENT** handle can be created by an RPC creation routine such as **clnt_create()** (see *rpc_clnt_create*(3N)).

These routines are safe for use in multithreaded applications. **CLIENT** handles can be shared between threads, however in this implementation requests by different threads are serialized (that is, the first request will receive its results before the second request is sent).

Routines

See *rpc*(3N) for the definition of the **CLIENT** data structure.

enum clnt_stat clnt_call()

A function macro that calls the remote procedure *procnum* associated with the client handle, *clnt*, which is obtained with an RPC client creation routine such as **clnt_create()** (see *rpc_clnt_create*(3N)). The parameter *inproc* is the XDR function used to encode the procedure's parameters, and *outproc* is the XDR function used to decode the procedure's results; *in* is the address of the procedure's argument(s), and *out* is the address of where to place the result(s). *tout* is the time allowed for results to be returned, which is overridden by a time-out set explicitly through **clnt_control()**, see *rpc_clnt_create*(3N).

If the remote call succeeds, the status returned is **RPC_SUCCESS**, otherwise an appropriate status is returned.

bool_t clnt_freeres()

A function macro that frees any data allocated by the RPC/XDR system when it decoded the results of

an RPC call. The parameter *out* is the address of the results, and *outproc* is the XDR routine describing the results. This routine returns **1** if the results were successfully freed, and **0** otherwise.

void clnt_geterr()
A function macro that copies the error structure out of the client handle to the structure at address *errp*.

void clnt_perrno()
Print a message to standard error corresponding to the condition indicated by *stat*. A newline is appended. Normally used after a procedure call fails for a routine for which a client handle is not needed, for instance **rpc_call()**.

void clnt_perror()
Print a message to the standard error indicating why an RPC call failed; *clnt* is the handle used to do the call. The message is prepended with string *s* and a colon. A newline is appended. Normally used after a remote procedure call fails for a routine which requires a client handle, for instance **clnt_call()**.

char *clnt_sperrno()
Take the same arguments as **clnt_perrno()**, but instead of sending a message to the standard error indicating why an RPC call failed, return a pointer to a string which contains the message.

clnt_sperrno() is normally used instead of **clnt_perrno()** when the program does not have a standard error (as a program running as a server quite likely does not), or if the programmer does not want the message to be output with **printf()** (see *printf*(3S)), or if a message format different than that supported by **clnt_perrno()** is to be used.

Note: unlike **clnt_sperror()** and **clnt_spcreaterror()** (see *rpc_clnt_create*(3N)), **clnt_sperrno()** does not return pointer to static data so the result will not get overwritten on each call.

char *clnt_sperror()
Like **clnt_perror()**, except that (like **clnt_sperrno()**) it returns a string instead of printing to standard error. However, **clnt_sperror()** does not append a newline at the end of the message.

Warning: returns pointer to a buffer that is overwritten on each call. In multithread applications, this buffer is implemented as thread-specific data.

enum clnt_stat rpc_broadcast()
Like **rpc_call()**, except the call message is broadcast to all the connectionless transports specified by *nettype*. If *nettype* is **NULL**, it defaults to **netpath**. Each time it receives a response, this routine calls **eachresult()**, whose form is:

```
bool_t eachresult(caddr_t out,const struct netbuf *addr,
const struct netconfig *netconf);
```

where *out* is the same as *out* passed to **rpc_broadcast()**, except that the remote procedure's output is decoded there; *addr* points to the address of the machine that sent the results, and *netconf* is the netconfig structure of the transport on which the remote server responded. If **eachresult()** returns **0**, **rpc_broadcast()** waits for more replies; otherwise it returns with appropriate status.

Warning: broadcast file descriptors are limited in size to the maximum transfer size of that transport. For Ethernet, this value is 1500 bytes. **rpc_broadcast()** uses **AUTH_SYS** credentials by default (see *rpc_clnt_auth*(3N)).

enum clnt_stat rpc_broadcast_exp()
Like **rpc_broadcast()**, except that the initial timeout, *inittime* and the maximum timeout, *waittime* are specified in milliseconds.

inittime is the initial time that **rpc_broadcast_exp()** waits before resending the request. After the first resend, the re-transmission interval increases exponentially until it exceeds *waittime*.

enum clnt_stat rpc_call()
Call the remote procedure associated with *prognum*, *versnum*, and *procnum* on the machine, *host*. The parameter *inproc* is used to encode the procedure's parameters, and *outproc* is used to decode the procedure's results; *in* is the address of the procedure's argument(s), and *out* is the address of where to place the result(s). *nettype* can be any of the values listed on *rpc*(3N). This routine returns RPC_SUCCESS if it succeeds, or an appropriate status is returned. Use the **clnt_perrno()**

routine to translate failure status into error messages.

Warning: **rpc_call()** uses the first available transport belonging to the class *nettype*, on which it can create a connection. You do not have control of timeouts or authentication using this routine.

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

SEE ALSO

printf(3S), rpc(3N), rpc_clnt_auth(3N), rpc_clnt_create(3N).

NAME

rpc_clnt_create, clnt_control, clnt_create, clnt_create_vers, clnt_destroy, clnt_dg_create, clnt_pcreateerror, clnt_raw_create, clnt_spcreateerror, clnt_tli_create, clnt_tp_create, clnt_vc_create, rpc_createerr - library routines for dealing with creation and manipulation of CLIENT handles

SYNOPSIS

```
#include <rpc/rpc.h>

bool_t clnt_control(CLIENT *clnt, const u_int req, char *info);

CLIENT *clnt_create(const char *host, const u_long prognum,
    const u_long versnum, const char *nettype);

CLIENT *clnt_create_vers(const char *host, const u_long prognum,
    u_long *vers_outp, const u_long vers_low, const u_long vers_high,
    char *nettype);

void clnt_destroy(CLIENT *clnt);

CLIENT *clnt_dg_create(const int fildes, const struct netbuf *svcaddr,
    const u_long prognum, const u_long versnum, const u_int sendsz,
    const u_int recvsz);

void clnt_pcreateerror(const char *s);

CLIENT *clnt_raw_create(const u_long prognum, const u_long versnum);

char *clnt_spcreateerror(const char *s);

CLIENT *clnt_tli_create(const int fildes, const struct netconfig *netconf,
    const struct netbuf *svcaddr, const_long prognum, const u_long versnum,
    const u_int sendsz, const u_int recvsz);

CLIENT *clnt_tp_create(const char *host, const u_long prognum,
    const u_long versnum, const struct netconfig *netconf);

CLIENT *clnt_vc_create(const int fildes, const struct netbuf *svcaddr,
    const u_long prognum, const u_long versnum, const u_int sendsz,
    const u_int recvsz);

struct rpc_createerr rpc_createerr;
```

DESCRIPTION

RPC library routines allow C language programs to make procedure calls on other machines across the network. First a **CLIENT** handle is created and then the client calls a procedure to send a request to the server. On receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends a reply.

These routines are MT-Safe. In the case of multithreaded applications, the **_REENTRANT** flag must be defined on the command line at compilation time (**-D_REENTRANT**). When the **_REENTRANT** flag is defined, **rpc_createerr** becomes a macro which enables each thread to have its own **rpc_createerr**.

Routines

See *rpc*(3N) for the definition of the **CLIENT** data structure.

bool_t clnt_control();

A function macro to change or retrieve various information about a client object. *req* indicates the type of operation, and *info* is a pointer to the information. For both connectionless and connection-oriented transports, the supported values of *req* and their argument types and what they do are:

```
CLSET_TIMEOUT     struct timeval  * set total timeout
CLGET_TIMEOUT     struct timeval  * get total timeout
```

Note: if you set the timeout using **clnt_control()**, the timeout argument passed by **clnt_call()** is ignored in all subsequent calls.

Note: If you set the timeout value to **0** **clnt_call()** immediately returns an error (**RPC_TIMEDOUT**). Set the timeout parameter to **0** for batching calls.

```
CLGET_FD          int             * get the associated file descriptor
```

```
CLGET_SVC_ADDR   struct netbuf   * get servers address
CLSET_FD_CLOSE   void            * close the file descriptor
                                   when destroying the client handle
                                   (see clnt_destroy())
CLSET_FD_NCLOSE  void              do not close the file descriptor
                                   when destroying the client handle
CLGET_VERS       unsigned long   * get the RPC program's
                                   version number associated with the
                                   client handle
CLSET_VERS       unsigned long   * set the RPC program's version
                                   number associated with the
                                   client handle.  This assumes
                                   that the RPC server for
                                   this new version is still
                                   listening at the address of the
                                   previous version.
CLGET_XID        unsigned long   * get the XID of the previous
                                   remote procedure call
CLSET_XID        unsigned long   * set the XID of the next
                                   remote procedure call
```

The following operations are valid for connectionless transports only:

```
CLSET_RETRY_TIMEOUT  struct timeval  * set the retry timeout
CLGET_RETRY_TIMEOUT  struct timeval  * get the retry timeout
```

The retry timeout is the time that RPC waits for the server to reply before retransmitting the request.

clnt_control() returns **TRUE** on success and **FALSE** on failure.

CLIENT *clnt_create()

Generic client creation routine for program *prognum* and version *versnum*. *host* identifies the name of the remote host where the server is located. *nettype* indicates the class of transport protocol to use. The transports are tried in left to right order in **NETPATH** variable or in top to bottom order in the netconfig database.

clnt_create() tries all the transports of the *nettype* class available from the **NETPATH** environment variable and the netconfig database, and chooses the first successful one. A default timeout is set and can be modified using **clnt_control()**. This routine returns **NULL** if it fails. The **clnt_pcreateerror()** routine can be used to print the reason for failure.

Note: **clnt_create()** returns a valid client handle even if the particular version number supplied to **clnt_create()** is not registered with the **rpcbind** service. This mismatch will be discovered by a **clnt_call** later (see *rpc_clnt_calls*(3N)).

CLIENT *clnt_create_vers()

Generic client creation routine which is similar to **clnt_create()** but which also checks for the version availability. *host* identifies the name of the remote host where the server is located. *nettype* indicates the class transport protocols to be used. If the routine is successful it returns a client handle created for the highest version between *vers_low* and *vers_high* that is supported by the server. *vers_outp* is set to this value. That is, after a successful return *vers_low* <= **vers_outp* <= *vers_high*. If no version between *vers_low* and *vers_high* is supported by the server then the routine fails and returns **NULL**. A default timeout is set and can be modified using **clnt_control()**. This routine returns **NULL** if it fails. The **clnt_pcreateerror()** routine can be used to print the reason for failure.

Note: **clnt_create()** returns a valid client handle even if the particular version number supplied to **clnt_create()** is not registered with the **rpcbind** service. This mismatch will be discovered by a **clnt_call** later (see *rpc_clnt_calls*(3N)). However, **clnt_create_vers()** does this for you and returns a valid handle only if a version within the range supplied is supported by the server.

void clnt_destroy()

A function macro that destroys the client's RPC handle. Destruction usually involves deallocation of private data structures, including *clnt* itself. Use of *clnt* is undefined after calling **clnt_destroy()**. If the RPC library opened the associated file descriptor, or **CLSET_FD_CLOSE** was set using **clnt_control()**, the file descriptor will be closed.

The caller should call **auth_destroy(***clnt*→**cl_auth)** (before calling **clnt_destroy()**) to destroy the associated AUTH structure (see *rpc_clnt_auth*(3N)).

CLIENT *clnt_dg_create()
This routine creates an RPC client for the remote program *prognum* and version *versnum*; the client uses a connectionless transport. The remote program is located at address *svcaddr*. The parameter *fildes* is an open and bound file descriptor. This routine will resend the call message in intervals of 15 seconds until a response is received or until the call times out. The total time for the call to time out is specified by **clnt_call()** (see **clnt_call()** in *rpc_clnt_calls*(3N)). The retry time out and the total time out periods can be changed using **clnt_control()**. The user may set the size of the send and receive buffers with the parameters *sendsz* and *recvsz*; values of **0** choose suitable defaults. This routine returns **NULL** if it fails.

void clnt_pcreateerror()
Print a message to standard error indicating why a client RPC handle could not be created. The message is prepended with the string *s* and a colon, and appended with a newline.

CLIENT *clnt_raw_create()
This routine creates an RPC client handle for the remote program *prognum* and version *versnum*. The transport used to pass messages to the service is a buffer within the process's address space, so the corresponding RPC server should live in the same address space; (see **svc_raw_create()** in *rpc_svc_create*(3N)). This allows simulation of RPC and measurement of RPC overheads, such as round trip times, without any kernel or networking interference. This routine returns **NULL** if it fails. **clnt_raw_create()** should be called after **svc_raw_create()**.

char *clnt_spcreateerror()
Like **clnt_pcreateerror()**, except that it returns a string instead of printing to the standard error. A newline is not appended to the message in this case.

Warning: returns a pointer to a buffer that is overwritten on each call. In multithread applications, this buffer is implemented as thread-specific data.

CLIENT *clnt_tli_create()
This routine creates an RPC client handle for the remote program *prognum* and version *versnum*. The remote program is located at address *svcaddr*. If *svcaddr* is **NULL** and it is connection-oriented, it is assumed that the file descriptor is connected. For connectionless transports, if *svcaddr* is **NULL**, **RPC_UNKNOWNADDR** error is set. *fildes* is a file descriptor which may be open, bound and connected. If it is **RPC_ANYFD**, it opens a file descriptor on the transport specified by *netconf*. If *fildes* is **RPC_ANYFD** and *netconf* is **NULL**, a **RPC_UNKNOWNPROTO** error is set. If *fildes* is unbound, then it will attempt to bind the descriptor. The user may specify the size of the buffers with the parameters *sendsz* and *recvsz*; values of **0**choose**suitable**defaults. Depending upon the type of the transport (connection-oriented or connectionless), **clnt_tli_create()** calls appropriate client creation routines. This routine returns **NULL** if it fails. The **clnt_pcreateerror()** routine can be used to print the reason for failure. The remote **rpcbind** service (see *rpcbind*(1M)) is not consulted for the address of the remote service.

CLIENT *clnt_tp_create()
Like **clnt_create()** except **clnt_tp_create()** tries only one transport specified through *netconf*.

clnt_tp_create() creates a client handle for the program *prognum*, the version *versnum*, and for the transport specified by *netconf*. Default options are set, which can be changed using **clnt_control()** calls. The remote **rpcbind** service on the host *host* is consulted for the address of the remote service. This routine returns **NULL** if it fails. The **clnt_pcreateerror()** routine can be used to print the reason for failure.

CLIENT *clnt_vc_create()
This routine creates an RPC client for the remote program *prognum* and version *versnum*; the client uses a connection-oriented transport. The remote program is located at address *svcaddr*. The parameter *fildes* is an open and bound file descriptor. The user may specify the size of the send and receive buffers with the parameters *sendsz* and *recvsz*; values of **0** choose suitable defaults. This routine returns **NULL** if it fails.

The address *svcaddr* should not be **NULL** and should point to the actual address of the remote program. **clnt_vc_create()** does not consult the remote **rpcbind** service for this information.

struct rpc_createerr rpc_createerr;
A global variable whose value is set by any RPC client handle creation routine that fails. It is used by

the routine **clnt_pcreateerror()** to print the reason for the failure.

In multithreaded applications, **rpc_createerr** becomes a macro which enables each thread to have its own **rpc_createerr**.

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

SEE ALSO

rpc(3N), rpc_clnt_auth(3N), rpc_clnt_calls(3N), rpcbind(1M).

NAME

rpc_control - library routine for manipulating global RPC attributes for client and server applications

SYNOPSIS

```
bool_t rpc_control(int op, void *info);
```

DESCRIPTION

This RPC library routine allows applications to set and modify global RPC attributes that apply to clients as well as servers. At present, it supports only server side operations. *op* indicates the type of operation, and *info* is a pointer to the operation specific information. The supported values of *op* and their argument types, and what they do are:

```
RPC_SVC_MTMODE_SET     int   *      set multithread mode
RPC_SVC_MTMODE_GET     int   *      get multithread mode
```

There are three multithread (MT) modes. These are:

```
RPC_SVC_MT_NONE        Single threaded mode    (default)
RPC_SVC_MT_USER        User MT mode
```

Unless the application sets the User MT modes, it will stay in the default (single threaded) mode. Once a mode is set, it cannot be changed.

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

RETURN VALUES

This routine returns **TRUE** if the operation was successful, and **FALSE** otherwise.

SEE ALSO

rpc(3N), rpc_svc_calls(3N), rpcbind(1M).

NAME

rpc_soc, authdes_create, authunix_create, authunix_create_default, callrpc, clnt_broadcast, clntraw_create, clnttcp_create, clntudp_bufcreate, clntudp_create, get_myaddress, pmap_getmaps, pmap_getport, pmap_rmtcall, pmap_set, pmap_unset, registerrpc, svc_fds, svc_getcaller, svc_getreq, svc_register, svc_unregister, svcfd_create, svcraw_create, svctcp_create, svcudp_bufcreate, svcudp_create, xdr_authunix_parms - obsolete library routines for RPC

SYNOPSIS

```
#define PORTMAP
#include <rpc/rpc.h>

AUTH * authdes_create(char *name, unsigned window,
    struct sockaddr *syncaddr, des_block *ckey);

AUTH * authunix_create(char *host, int uid , int gid, int grouplen, int gidlistp);

AUTH * authunix_create_default(void)

callrpc(char *host, u_long prognum, u_long versnum, u_long procnum,
    xdrproc_t inproc, char *in, xdrproc_t outproc, char *out);

enum clnt_stat clnt_broadcast(u_long prognum, u_long versnum,
    u_long procnum, xdrproc_t inproc, char *in, xdrproc_t outproc, char *out,
    resultproc_t eachresult);

CLIENT * clntraw_create(u_long prognum, u_long versnum);

CLIENT * clnttcp_create(struct sockaddr_in *addr, u_long prognum,
    u_long versnum, int *fdp, u_int sendsz, u_int recvsz);

CLIENT * clntudp_bufcreate(struct sockaddr_in *addr, u_long prognum,
    u_long versnum, struct timeval wait, int *fdp, u_int sendsz, u_int recvsz);

CLIENT * clntudp_create(struct sockaddr_in *addr, u_long prognum,
    u_long versnum, struct timeval wait, int *fdp);

void get_myaddress(struct sockaddr_in *addr);

struct pmaplist * pmap_getmaps(struct sockaddr_in *addr);

u_short pmap_getport(struct sockaddr_in *addr, u_long prognum,
    u_long versnum, u_long protocol);

enum clnt_stat pmap_rmtcall(struct sockaddr_in *addr, u_long prognum,
    u_long versnum, u_long procnum, char *in, xdrproct_t inproc, char *out,
    xdrproct_t outproc, struct timeval tout, u_long *portp);

bool_t pmap_set(u_long prognum, u_long versnum, u_long protocol, u_short port);

bool_t pmap_unset(u_long prognum, u_long versnum);

int svc_fds;

struct sockaddr_in * svc_getcaller(SVCXPRT *xprt);

void svc_getreq(int rdfds);

SVCXPRT * svcfd_create(int fd, u_int sendsz, u_int recvsz);

SVCXPRT * svcraw_create(void);

SVCXPRT * svctcp_create(int fd, u_int sendsz, u_int recvsz);

SVCXPRT * svcudp_bufcreate(int fd, u_int sendsz, u_int recvsz);

SVCXPRT * svcudp_create(int fd);

registerrpc(u_long prognum, u_long versnum, u_long procnum,
    char *(*procname)(), xdrproc_t inproc, xdrproc_t outproc);

svc_register(SVCXPRT *xprt, u_long prognum,
    u_long versnum, void (*dispatch)(), u_long protocol);

void svc_unregister(u_long prognum, u_long versnum);
```

xdr_authunix_parms(XDR *_xdrs_, struct authunix_parms *_aupp_);

DESCRIPTION

RPC routines allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a request to the server. Upon receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends back a reply. Finally, the procedure call returns to the client.

The routines described in this manual page have been superseded by other routines. The preferred routine is given after the description of the routine. New programs should use the preferred routines, as support for the older interfaces may be dropped in future releases.

File Descriptors

Transport independent RPC uses XTI as its transport interface instead of sockets.

Some of the routines described in this section (such as **clnttcp_create()**) take a pointer to a file descriptor as one of the parameters. If the user passed **RPC_ANYSOCK** as the file descriptor, then the routine will return a TLI file descriptor and not a socket.

Routines

The following routines require that the header **<rpc/rpc.h>** be included. The symbol **PORTMAP** should be defined so that the appropriate function declarations for the old interfaces are included through the header files.

AUTH * authdes_create()

authdes_create() is the first of two routines which interface to the RPC secure authentication system, known as DES authentication. The second is **authdes_getucred()**, below. Note: the keyserver daemon *keyserv*(1M) must be running for the DES authentication system to work.

authdes_create(), used on the client side, returns an authentication handle that will enable the use of the secure authentication system. The first parameter *name* is the network name, or *netname*, of the owner of the server process. This field usually represents a hostname derived from the utility routine **host2netname()**, but could also represent a user name using **user2netname()** (see *secure_rpc*(3N)). The second field is window on the validity of the client credential, given in seconds. A small window is more secure than a large one, but choosing too small of a window will increase the frequency of resynchronizations because of clock drift. The third parameter *syncaddr* is optional. If it is NULL, then the authentication system will assume that the local clock is always in sync with the server's clock, and will not attempt resynchronizations. If an address is supplied, however, then the system will use the address for consulting the remote time service whenever resynchronization is required. This parameter is usually the address of the RPC server itself. The final parameter *ckey* is also optional. If it is NULL, then the authentication system will generate a random DES key to be used for the encryption of credentials. If it is supplied, however, then it will be used instead.

Warning: this routine exists for backward compatibility only, and is obsoleted by **authdes_seccreate()** (see *secure_rpc*(3N)).

AUTH * authunix_create()

Create and return an RPC authentication handle that contains UX authentication information. The parameter *host* is the name of the machine on which the information was created; *uid* is the user's user ID; *gid* is the user's current group ID; *grouplen* and *gidlistp* refer to a counted array of groups to which the user belongs.

Warning: it is not very difficult to impersonate a user.

Warning: this routine exists for backward compatibility only, and is obsoleted by **authsys_create()** (see *rpc_clnt_auth*(3N)).

AUTH * authunix_create_default(void)

Call **authunix_create()** with the appropriate parameters.

Warning: this routine exists for backward compatibility only, and is obsoleted by **authsys_create_default()** (see *rpc_clnt_auth*(3N)).

callrpc()

Call the remote procedure associated with *prognum*, *versnum*, and *procnum* on the machine, *host*. The parameter *inproc* is used to encode the procedure's parameters, and *outproc* is used to

decode the procedure's results; *in* is the address of the procedure's argument, and *out* is the address of where to place the result(s). This routine returns **0** if it succeeds, or the value of **enum clnt_stat** cast to an integer if it fails. The routine **clnt_perrno()** (see *rpc_clnt_calls*(3N)) is handy for translating failure statuses into messages.

Warning: you do not have control of timeouts or authentication using this routine. This routine exists for backward compatibility only, and is obsoleted by **rpc_call()** (see *rpc_clnt_calls*(3N)).

enum clnt_stat clnt_broadcast()

Like **callrpc()**, except the call message is broadcast to all locally connected broadcast nets. Each time the caller receives a response, this routine calls **eachresult()**, whose form is:

eachresult(char ****out*,struct sockaddr_in ****addr***);**

where *out* is the same as *out* passed to **clnt_broadcast()**, except that the remote procedure's output is decoded there; *addr* points to the address of the machine that sent the results. If **eachresult()** returns **0** **clnt_broadcast()** waits for more replies; otherwise it returns with appropriate status. If **eachresult()** is NULL, **clnt_broadcast()** returns without waiting for any replies.

Warning: broadcast packets are limited in size to the maximum transfer unit of the transports involved. For Ethernet, the callers argument size is approximately 1500 bytes. Since the call message is sent to all connected networks, it may potentially lead to broadcast storms. **clnt_broadcast()** uses **AUTH_SYS** credentials by default (see *rpc_clnt_auth*(3N)).

Warning: this routine exists for backward compatibility only, and is obsoleted by **rpc_broadcast()** (see *rpc_clnt_calls*(3N)).

CLIENT * clntraw_create()

This routine creates an internal, memory-based RPC client for the remote program *prognum*, version *versnum*. The transport used to pass messages to the service is actually a buffer within the process's address space, so the corresponding RPC server should live in the same address space; see **svcraw_create()**. This allows simulation of RPC and acquisition of RPC overheads, such as round trip times, without any kernel interference. This routine returns NULL if it fails.

Warning: this routine exists for backward compatibility only, and has the same functionality as **clnt_raw_create()** (see *rpc_clnt_create*(3N)), which obsoletes it.

CLIENT * clnttcp_create()

This routine creates an RPC client for the remote program *prognum*, version *versnum*; the client uses TCP/IP as a transport. The remote program is located at Internet address *addr*. If *addr*→*sin_port* is **0**, then it is set to the actual port that the remote program is listening on (the remote **rpcbind** service is consulted for this information). The parameter **fdp* is a file descriptor, which may be open and bound; if it is **RPC_ANYSOCK**, then this routine opens a new one and sets **fdp*. Refer to the *File Descriptor* section for more information. Since TCP-based RPC uses buffered I/O, the user may specify the size of the send and receive buffers with the parameters *sendsz* and *recvsz*; values of **0** choose suitable defaults. This routine returns NULL if it fails.

Warning: this routine exists for backward compatibility only. **clnt_create()**, **clnt_tli_create()**, or **clnt_vc_create()** (see *rpc_clnt_create*(3N)) should be used instead.

CLIENT * clntudp_bufcreate()

Create a client handle for the remote program *prognum*, on *versnum*; the client uses UDP/IP as the transport. The remote program is located at the Internet address *addr*. If *addr*→*sin_port* is **0**, it is set to port on which the remote program is listening on (the remote **rpcbind** service is consulted for this information). The parameter **fdp* is a file descriptor, which may be open and bound; if it is **RPC_ANYSOCK**, then this routine opens a new one and sets **fdp*. Refer to the *File Descriptor* section for more information. The UDP transport resends the call message in intervals of *wait* time until a response is received or until the call times out. The total time for the call to time out is specified by **clnt_call()** (see *rpc_clnt_calls*(3N)). If successful it returns a client handle, otherwise it returns NULL. The error can be printed using the **clnt_pcreateerror()** (see *rpc_clnt_create*(3N)) routine.

The user can specify the maximum packet size for sending and receiving by using *sendsz* and *recvsz* arguments for UDP-based RPC messages.

Warning: if *addr→sin_port* is **0** and the requested version number *versnum* is not registered with the remote portmap service, it returns a handle if at least a version number for the given program number is registered. The version mismatch is discovered by a **clnt_call()** later (see *rpc_clnt_calls*(3N)).

Warning: this routine exists for backward compatibility only. **clnt_tli_create()** or **clnt_dg_create()** (see *rpc_clnt_create*(3N)) should be used instead.

CLIENT * clntudp_create()

This routine creates an RPC client handle for the remote program *prognum*, version *versnum*; the client uses UDP/IP as a transport. The remote program is located at Internet address *addr*. If *addr→sin_port* is **0**, then it is set to actual port that the remote program is listening on (the remote **rpcbind** service is consulted for this information). The parameter **fdp* is a file descriptor, which may be open and bound; if it is **RPC_ANYSOCK**, then this routine opens a new one and sets **fdp*. Refer to the *File Descriptor* section for more information. The UDP transport resends the call message in intervals of **wait** time until a response is received or until the call times out. The total time for the call to time out is specified by **clnt_call()** (see *rpc_clnt_calls*(3N)). **clntudp_create()** returns a client handle on success, otherwise it returns NULL. The error can be printed using the **clnt_pcreateerror()** (see *rpc_clnt_create*(3N)) routine.

Warning: since UDP-based RPC messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

Warning: this routine exists for backward compatibility only. **clnt_create()**, **clnt_tli_create()**, or **clnt_dg_create()** (see *rpc_clnt_create*(3N)) should be used instead.

void get_myaddress()

Places the local system's IP address into **addr*, without consulting the library routines that deal with **/etc/hosts**. The port number is always set to **htons(PMAPPORT)**.

Warning: this routine is only intended for use with the RPC library. It returns the local system's address in a form compatible with the RPC library, and should not be taken as the system's actual IP address. In fact, the **addr* buffer's host address part is actually zeroed. This address may have only local significance and should **NOT** be assumed to be an address that can be used to connect to the local system by remote systems or processes.

Warning: this routine remains for backward compatibility only. The routine **netdir_getbyname()** (see *netdir*(3N)) should be used with the name **HOST_SELF** to retrieve the local system's network address as a *netbuf* structure.

struct pmaplist * pmap_getmaps()

A user interface to the **portmap** service, which returns a list of the current RPC program-to-port mappings on the host located at IP address *addr*. This routine can return NULL . The command '**rpcinfo -p**' uses this routine.

Warning: this routine exists for backward compatibility only, enhanced functionality is provided by **rpcb_getmaps()** (see *rpcbind*(3N)).

u_short pmap_getport()

A user interface to the **portmap** service, which returns the port number on which waits a service that supports program *prognum*, version *versnum*, and speaks the transport protocol associated with *protocol*. The value of *protocol* is most likely **IPPROTO_UDP** or **IPPROTO_TCP**. A return value of **0** means that the mapping does not exist or that the RPC system failured to contact the remote **portmap** service. In the latter case, the global variable **rpc_createerr** contains the RPC status.

Warning: this routine exists for backward compatibility only, enhanced functionality is provided by **rpcb_getaddr()** (see *rpcbind*(3N)).

enum clnt_stat pmap_rmtcall()

Request that the **portmap** on the host at IP address **addr* make an RPC on the behalf of the caller to a procedure on that host. **portp* is modified to the program's port number if the procedure succeeds. The definitions of other parameters are discussed in **callrpc()** and **clnt_call()** (see *rpc_clnt_calls*(3N)).

Note: this procedure is only available for the UDP transport.

Warning: if the requested remote procedure is not registered with the remote **portmap** then no error response is returned and the call times out. Also, no authentication is done.

Warning: this routine exists for backward compatibility only, enhanced functionality is provided by **rpcb_rmtcall()** (see *rpcbind*(3N)).

bool_t pmap_set()

A user interface to the **portmap** service, that establishes a mapping between the triple [*prognum*, *versnum*, *protocol*] and *port* on the machine's **portmap** service. The value of *protocol* may be **IPPROTO_UDP** or **IPPROTO_TCP**. Formerly, the routine failed if the requested *port* was found to be in use. Now, the routine only fails if it finds that *port* is still bound. If *port* is not bound, the routine removes the old registration and completes the requested registration. This routine returns **1** if it succeeds, **0** otherwise. Automatically done by **svc_register()**.

Warning: this routine exists for backward compatibility only, enhanced functionality is provided by **rpcb_set()** (see *rpcbind*(3N)).

bool_t pmap_unset()

A user interface to the **portmap** service, which destroys all mapping between the triple [*prognum*, *versnum*, *all-protocols*] and *port* on the machine's **portmap** service. This routine returns one if it succeeds, **0** otherwise.

Warning: this routine exists for backward compatibility only, enhanced functionality is provided by **rpcb_unset()** (see *rpcbind*(3N)).

int svc_fds;

A global variable reflecting the RPC service side's read file descriptor bit mask; it is suitable as a parameter to the **select()** call. This is only of interest if a service implementor does not call **svc_run()**, but rather does his own asynchronous event processing. This variable is read-only (do not pass its address to **select()**!), yet it may change after calls to **svc_getreq()** or any creation routines. Similar to **svc_fdset**, but limited to 32 descriptors.

Warning: this interface is obsoleted by **svc_fdset** (see *rpc_svc_calls*(3N)).

struct sockaddr_in * svc_getcaller()

This routine returns the network address, represented as a **struct sockaddr_in**, of the caller of a procedure associated with the RPC service transport handle, *xprt*.

Warning: this routine exists for backward compatibility only, and is obsolete. The preferred interface is **svc_getrpccaller()** (see *rpc_svc_reg*(3N)), which returns the address as a **struct netbuf**.

void svc_getreq()

This routine is only of interest if a service implementor does not call **svc_run()**, but instead implements custom asynchronous event processing. It is called when the **select()** call has determined that an RPC request has arrived on some RPC file descriptors; *rdfds* is the resultant read file descriptor bit mask. The routine returns when all file descriptors associated with the value of *rdfds* have been serviced. This routine is similar to **svc_getreqset()** but is limited to 32 descriptors.

Warning: this interface is obsoleted by **svc_getreqset()**.

SVCXPRT * svcfd_create()

Create a service on top of any open and bound descriptor. Typically, this descriptor is a connected file descriptor for a stream protocol. Refer to the *File Descriptor* section for more information. *sendsz* and *recvsz* indicate sizes for the send and receive buffers. If they are **0**, a reasonable default is chosen.

Warning: this interface is obsoleted by **svc_fd_create()** (see *rpc_svc_create*(3N)).

SVCXPRT * svcraw_create(void);

This routine creates an internal, memory-based RPC service transport, to which it returns a pointer. The transport is really a buffer within the process's address space, so the corresponding RPC client should live in the same address space; see **clntraw_create()**. This routine allows simulation of RPC and acquisition of RPC overheads (such as round trip times), without any kernel interference. This routine returns NULL if it fails.

Warning: this routine exists for backward compatibility only, and has the same functionality of **svc_raw_create()** (see *rpc_svc_create*(3N)), which obsoletes it.

SVCXPRT * svctcp_create()

This routine creates a TCP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the file descriptor *fd*, which may be **RPC_ANYSOCK**, in which case a new file descriptor is created. If the file descriptor is not bound to a local TCP port, then this routine binds it to an arbitrary port. Refer to the *File Descriptor* section for more information. Upon completion, *xprt→xp_fd* is the transport's file descriptor, and *xprt→xp_port* is the transport's port number. This routine returns NULL if it fails. Since TCP-based RPC uses buffered I/O, users may specify the size of buffers; values of **0** choose suitable defaults.

Warning: this routine exists for backward compatibility only. **svc_create()**, **svc_tli_create()**, or **svc_vc_create()** (see *rpc_svc_create*(3N)) should be used instead.

SVCXPRT * svcudp_bufcreate()

This routine creates a UDP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the file descriptor *fd*. If *fd* is **RPC_ANYSOCK**, then a new file descriptor is created. If the file descriptor is not bound to a local UDP port, then this routine binds it to an arbitrary port. Upon completion, *xprt→xp_fd* is the transport's file descriptor, and *xprt→xp_port* is the transport's port number. Refer to the *File Descriptor* section for more information. This routine returns NULL if it fails.

The user specifies the maximum packet size for sending and receiving UDP-based RPC messages by using the *sendsz* and *recvsz* parameters.

Warning: this routine exists for backward compatibility only. **svc_tli_create()**, or **svc_dg_create()** (see *rpc_svc_create*(3N)) should be used instead.

SVCXPRT * svcudp_create()

This routine creates a UDP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the file descriptor *fd*, which may be **RPC_ANYSOCK**, in which case a new file descriptor is created. If the file descriptor is not bound to a local UDP port, then this routine binds it to an arbitrary port. Upon completion, *xprt→xp_fd* is the transport's file descriptor, and *xprt→xp_port* is the transport's port number. This routine returns NULL if it fails.

Warning: since UDP-based RPC messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

Warning: this routine exists for backward compatibility only. **svc_create()**, **svc_tli_create()**, or **svc_dg_create()** (see *rpc_svc_create*(3N)) should be used instead.

registerrpc()

Register program *prognum*, procedure *procname*, and version *versnum* with the RPC service package. If a request arrives for program *prognum*, version *versnum*, and procedure *procnum*, *procname* is called with a pointer to its parameter(s); *procname* should return a pointer to its static result(s); *inproc* is used to decode the parameters while *outproc* is used to encode the results. This routine returns **0** if the registration succeeded, **-1** otherwise.

svc_run() must be called after all the services are registered.

Warning: this routine exists for backward compatibility only, and is obsoleted by **rpc_reg()** (see *rpc_svc_calls*(3N)).

svc_register()

Associates *prognum* and *versnum* with the service dispatch procedure, *dispatch*. If *protocol* is **0**, the service is not registered with the **portmap** service. If *protocol* is non-zero, then a mapping of the triple [*prognum*, *versnum*, *protocol*] to *xprt→xp_port* is established with the local **portmap** service (generally *protocol* is **0**, **IPPROTO_UDP** or **IPPROTO_TCP**). The procedure *dispatch* has the following form:

dispatch(struct svc_req ****request*,SVCXPRT *****xprt**);

The **svc_register()** routine returns one if it succeeds, and **0** otherwise.

Warning: this routine exists for backward compatibility only; enhanced functionality is provided by **svc_reg()** (see *rpc_svc_calls*(3N)).

void svc_unregister()

Remove all mapping of the double [*prognum*, *versnum*] to dispatch routines, and of the triple

[*prognum*, *versnum*, *all-protocols*] to port number from **portmap**.

Warning: this routine exists for backward compatibility, enhanced functionality is provided by **svc_unreg()** (see *rpc_svc_calls*(3N)).

xdr_authunix_parms()

Used for describing UNIX credentials. This routine is useful for users who wish to generate these credentials without using the RPC authentication package.

Warning: this routine exists for backward compatibility only, and is obsoleted by **xdr_authsys_parms()** (see *rpc_xdr*(3N)).

MT-LEVEL

Unsafe

NOTES

These interfaces are unsafe in multithreaded applications. Unsafe interfaces should be called only from the main thread.

SEE ALSO

keyserv(1M), rpcbind(1M), rpcinfo(1M), rpc(3N), rpc_clnt_auth(3N), rpc_clnt_calls(3N), rpc_clnt_create(3N), rpc_svc_calls(3N), rpc_svc_create(3N), rpc_svc_err(3N), rpc_svc_reg(3N), rpcbind(3N), secure_rpc(3N), select(2).

NAME

rpc_svc_calls, svc_dg_enablecache, svc_done, svc_exit, svc_fdset, svc_freeargs, svc_getargs, svc_getreq_common, svc_getreq_poll, svc_getreqset, svc_getrpccaller, svc_pollset, svc_run, svc_sendreply - library routines for RPC servers

SYNOPSIS

```
#include <rpc/rpc.h>
int svc_dg_enablecache(SVCXPRT *xprt, const unsigned long cache_size);
int svc_done(SVCXPRT *xprt);
void svc_exit(void);
fd_set svc_fdset;
bool_t svc_freeargs(const SVCXPRT *xprt, const xdrproc_t inproc,
    caddr_t in);
bool_t svc_getargs(const SVCXPRT *xprt, const xdrproc_t inproc, caddr_t in);
void svc_getreq_common(const int fd);
void svc_getreq_poll(struct pollfd *pfdp, const int pollretval);
void svc_getreqset(fd_set *rdfds);
struct netbuf *svc_getrpccaller(const SVCXPRT *xprt);
void svc_run(void);
bool_t svc_sendreply(const SVCXPRT *xprt, const xdrproc_t outproc,
    const caddr_t out);
```

DESCRIPTION

These routines are part of the RPC library which allows C language programs to make procedure calls on other machines across the network.

These routines are associated with the server side of the RPC mechanism. Some of them are called by the server side dispatch function, while others (such as **svc_run()**) are called when the server is initiated.

Multithread Considerations

In the current implementation, the service transport handle **SVCXPRT** contains a single data area for decoding arguments and encoding results. Therefore, this structure cannot be freely shared between threads that call functions that do this. However, when a server is operating in the User MT modes, a copy of this structure is passed to the service dispatch procedure in order to enable concurrent request processing. Under these circumstances, some routines which would otherwise be unsafe, become safe. These are marked as such. Also marked are routines that are unsafe for MT applications, and are not to be used by such applications.

Routines

See *rpc*(3N) for the definition of the **SVCXPRT** data structure.

int svc_dg_enablecache()

This function allocates a duplicate request cache for the service endpoint *xprt*, large enough to hold *cache_size* entries. Once enabled, there is no way to disable caching. This routine returns **1** if space necessary for a cache of the given size was successfully allocated, and **0** otherwise.

This function is safe in MT applications.

int svc_done()

This function frees resources allocated to service a client request directed to the service endpoint *xprt*. This call pertains only to servers executing in the User MT mode. In the User MT mode, service procedures must invoke this call before returning, either after a client request has been serviced, or after an error or abnormal condition that prevents a reply from being sent. After **svc_done**() is invoked, the service endpoint *xprt* should not be referenced by the service procedure. Server multithreading modes and parameters can be set using the **rpc_control**() call.

This function is safe in MT applications. It will have no effect if invoked in modes other than the User MT mode.

void svc_exit(void)
This function when called by any of the RPC server procedure or otherwise, destroys all services registered by the server and causes **svc_run()** to return.

If RPC server activity is to be resumed, services must be reregistered with the RPC library either through one of the *rpc_svc_create*(3N) functions, or using **xprt_register()**.

svc_exit() has global scope and ends all RPC server activity.

fd_set svc_fdset
A global variable reflecting the RPC server's read file descriptor bit mask. This is only of interest if service implementors do not call **svc_run()**, but rather do their own asynchronous event processing. This variable is read-only, and it may change after calls to **svc_getreqset()** or any creation routines. Do not pass its address to *select*(2)! Instead, pass the address of a copy.

MT applications executing in the User MT mode should never read this variable. They should use auxiliary threads to do asynchronous event processing.

bool_t svc_freeargs()
A function macro that frees any data allocated by the **RPC/XDR** system when it decoded the arguments to a service procedure using **svc_getargs()**. This routine returns **TRUE** if the results were successfully freed, and **FALSE** otherwise.

This function macro is safe in MT applications utilizing the User MT modes.

bool_t svc_getargs()
A function macro that decodes the arguments of an RPC request associated with the RPC service transport handle *xprt*. The parameter *in* is the address where the arguments will be placed; *inproc* is the XDR routine used to decode the arguments. This routine returns **TRUE** if decoding succeeds, and **FALSE** otherwise.

This function macro is safe in MT applications utilizing the User MT modes.

void svc_getreq_common()
This routine is called to handle a request on the given file descriptor.

This function macro is unsafe in MT applications.

void svc_getreq_poll()
This routine is only of interest if a service implementor does not call **svc_run()**, but instead implements custom asynchronous event processing. It is called when *poll*(2) has determined that an RPC request has arrived on some RPC file descriptors; *pollretval* is the return value from *poll*(2) and *pfdp* is the array of *pollfd* structures on which the *poll*(2) was done. It is assumed to be an array large enough to contain the maximal number of descriptors allowed.

This function macro is unsafe in MT applications.

void svc_getreqset()
This routine is only of interest if a service implementor does not call **svc_run()**, but instead implements custom asynchronous event processing. It is called when *select*(2) has determined that an RPC request has arrived on some RPC file descriptors; *rdfds* is the resultant read file descriptor bit mask. The routine returns when all file descriptors associated with the value of *rdfds* have been serviced.

This function macro is unsafe in MT applications.

struct netbuf *svc_getrpccaller()
The approved way of getting the network address of the caller of a procedure associated with the RPC service transport handle *xprt*.

This function macro is safe in MT applications.

void svc_run(void)
This routine never returns. In single threaded mode, it waits for RPC requests to arrive, and calls the appropriate service procedure using **svc_getreq_poll()** when one arrives. This procedure is usually waiting for the *poll*(2) library call to return.

Applications executing in the User MT modes should invoke this function exactly once. In the User MT mode, it will provide a framework for service developers to create and manage their own threads for servicing client requests.

bool_t svc_sendreply()
Called by an RPC service's dispatch routine to send the results of a remote procedure call. The

parameter *xprt* is the request's associated transport handle; *outproc* is the XDR routine which is used to encode the results; and *out* is the address of the results. This routine returns **TRUE** if it succeeds, **FALSE** otherwise.

This function macro is safe in MT applications utilizing the User MT modes.

MULTITHREAD USAGE

Thread Safe: See the **NOTES** section of this page.
Cancel Safe: See the **NOTES** section of this page.
Fork Safe: **No**
Async-cancel Safe: **No**
Async-signal Safe: **No**

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

NOTES

svc_dg_enablecache() and **svc_getrpccaller()** are Thread Safe and Cancel Safe in multithreaded applications. **svc_freeargs()**, **svc_getargs()**, and **svc_sendreply()** are Thread safe and Cancel Safe in multithreaded applications utilizing the User MT modes. **svc_getreq_common()**, **svc_getreqset()**, and **svc_getreq_poll()** are unsafe in multithreaded applications and should be called only from the main thread.

SEE ALSO

rpcgen(1), poll(2), rpc(3N), rpc_control(3N), rpc_svc_create(3N), rpc_svc_err(3N), rpc_svc_reg(3N), select(2).

NAME

rpc_svc_create, svc_control, svc_create, svc_destroy, svc_dg_create, svc_fd_create, svc_raw_create, svc_tli_create, svc_tp_create, svc_vc_create - library routines for the creation of server handles

SYNOPSIS

```
#include <rpc/rpc.h>

bool_t svc_control(SVCXPRT *svc, const u_int req, void *info);

int svc_create(const void (*dispatch)(const struct svc_req *,
    const SVCXPRT *), const u_long prognum, const u_long versnum,
    const char *nettype);

void svc_destroy(SVCXPRT *xprt);

SVCXPRT1 *svc_dg_create(const int fildes, const u_int sendsz,
    const u_int recvsz);

SVCXPRT *svc_fd_create(const int fildes, const u_int sendsz,
    const u_int recvsz);

SVCXPRT *svc_raw_create(void);

SVCXPRT *svc_tli_create(const int fildes, const struct netconfig *netconf,
    const struct t_bind *bindaddr, const u_int sendsz, const u_int recvsz);

SVCXPRT *svc_tp_create(const void (*dispatch)(const struct svc_req *,
    const SVCXPRT *), const u_long prognum,
    const u_long versnum, const struct netconfig *netconf);

SVCXPRT *svc_vc_create(const int fildes, const u_int sendsz,
    const u_int recvsz);
```

DESCRIPTION

These routines are part of the RPC library which allows C language programs to make procedure calls on servers across the network. These routines deal with the creation of service handles. Once the handle is created, the server can be invoked by calling **svc_run()**.

Routines

See *rpc*(3N) for the definition of the **SVCXPRT** data structure.

bool_t svc_control()

A function to change or retrieve various information about a service object. *req* indicates the type of operation and *info* is a pointer to the information. The supported values of *req*, their argument types, and what they do are:

SVCGET_VERSQUIET

If a request is received for a program number served by this server but the version number is outside the range registered with the server, an **RPC_PROGVERSMISMATCH** error will normally be returned. *info* should be a pointer to an integer. Upon successful completion of the **SVCGET_VERSQUIET** request, **info* contains an integer which describes the server's current behavior: **0** indicates normal server behavior (that is, an **RPC_PROGVERSMISMATCH** error will be returned); **1** indicates that the out of range request will be silently ignored.

SVCSET_VERSQUIET

If a request is received for a program number served by this server but the version number is outside the range registered with the server, an **RPC_PROGVERSMISMATCH** error will normally be returned. It is sometimes desirable to change this behavior. *info* should be a pointer to an integer which is either **0** (indicating normal server behavior - an **RPC_PROGVERSMISMATCH** error will be returned), or **1** (indicating that the out of range request should be silently ignored).

int svc_create()

svc_create() creates server handles for all the transports belonging to the class *nettype*.

nettype defines a class of transports which can be used for a particular application. The transports are tried in left to right order in **NETPATH** variable or in top to bottom order in the netconfig database. If *nettype* is **NULL**, it defaults to **netpath**.

svc_create() registers itself with the **rpcbind** service (see *rpcbind*(1M)). *dispatch* is called when there is a remote procedure call for the given *prognum* and *versnum*; this requires calling **svc_run()** (see **svc_run()** in *rpc_svc_reg*(3N)). If **svc_create()** succeeds, it returns the number of server handles it created, otherwise it returns **0** and an error message is logged.

void svc_destroy()
A function macro that destroys the RPC service handle *xprt*. Destruction usually involves deallocation of private data structures, including *xprt* itself. Use of *xprt* is undefined after calling this routine.

SVCXPRT *svc_dg_create()
This routine creates a connectionless RPC service handle, and returns a pointer to it. This routine returns **NULL** if it fails, and an error message is logged. *sendsz* and *recvsz* are parameters used to specify the size of the buffers. If they are **0**, suitable defaults are chosen. The file descriptor *fildes* should be open and bound. The server is not registered with *rpcbind*(1M).

Warning: since connectionless-based RPC messages can only hold limited amount of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

SVCXPRT *svc_fd_create()
This routine creates a service on top of an open and bound file descriptor, and returns the handle to it. Typically, this descriptor is a connected file descriptor for a connection-oriented transport. *sendsz* and *recvsz* indicate sizes for the send and receive buffers. If they are **0**, reasonable defaults are chosen. This routine returns **NULL** if it fails, and an error message is logged.

SVCXPRT *svc_raw_create(void)
This routine creates an RPC service handle and returns a pointer to it. The transport is really a buffer within the process's address space, so the corresponding RPC client should live in the same address space; (see **clnt_raw_create()** in *rpc_clnt_create*(3N)). This routine allows simulation of RPC and acquisition of RPC overheads (such as round trip times), without any kernel and networking interference. This routine returns **NULL** if it fails, and an error message is logged.

Note: **svc_run()** should not be called when the raw interface is being used.

SVCXPRT *svc_tli_create()
This routine creates an RPC server handle, and returns a pointer to it. *fildes* is the file descriptor on which the service is listening. If *fildes* is **RPC_ANYFD**, it opens a file descriptor on the transport specified by *netconf*. If the file descriptor is unbound and *bindaddr* is non-null *fildes* is bound to the address specified by *bindaddr*, otherwise *fildes* is bound to a default address chosen by the transport. In the case where the default address is chosen, the number of outstanding connect requests is set to 8 for connection-oriented transports. The user may specify the size of the send and receive buffers with the parameters *sendsz* and *recvsz ;* values of **0** choose suitable defaults. This routine returns **NULL** if it fails, and an error message is logged. The server is not registered with the *rpcbind*(1M) service.

SVCXPRT *svc_tp_create()
svc_tp_create() creates a server handle for the network specified by *netconf*, and registers itself with the **rpcbind** service. *dispatch* is called when there is a remote procedure call for the given *prognum* and *versnum*; this requires calling **svc_run()**. **svc_tp_create()** returns the service handle if it succeeds, otherwise a **NULL** is returned and an error message is logged.

SVCXPRT *svc_vc_create()
This routine creates a connection-oriented RPC service and returns a pointer to it. This routine returns **NULL** if it fails, and an error message is logged. The users may specify the size of the send and receive buffers with the parameters *sendsz* and *recvsz*; values of **0** choose suitable defaults. The file descriptor *fildes* should be open and bound. The server is not registered with the *rpcbind*(1M) service.

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

SEE ALSO

rpcbind(1M), rpc(3N), rpc_svc_calls(3N), rpc_svc_err(3N), rpc_svc_reg(3N).

NAME

rpc_svc_err, svcerr_auth, svcerr_decode, svcerr_noproc, svcerr_noprog, svcerr_progvers, svcerr_systemerr, svcerr_weakauth - library routines for server side remote procedure call errors

SYNOPSIS

```
#include <rpc/rpc.h>
void svcerr_auth(const SVCXPRT *xprt, const enum auth_stat why);
void svcerr_decode(const SVCXPRT *xprt);
void svcerr_noproc(const SVCXPRT *xprt);
void svcerr_noprog(const SVCXPRT *xprt);
void svcerr_progvers(const SVCXPRT *xprt, u_long low_vers, u_long high_vers);
void svcerr_systemerr(const SVCXPRT *xprt);
void svcerr_weakauth(const SVCXPRT *xprt);
```

DESCRIPTION

These routines are part of the RPC library which allows C language programs to make procedure calls on other machines across the network.

These routines can be called by the server side dispatch function if there is any error in the transaction with the client.

Routines

See *rpc*(3N) for the definition of the **SVCXPRT** data structure.

void svcerr_auth()
Called by a service dispatch routine that refuses to perform a remote procedure call due to an authentication error.

void svcerr_decode()
Called by a service dispatch routine that cannot successfully decode the remote parameters (see **svc_getargs()** in *rpc_svc_reg*(3N)).

void svcerr_noproc()
Called by a service dispatch routine that does not implement the procedure number that the caller requests.

void svcerr_noprog()
Called when the desired program is not registered with the RPC package. Service implementors usually do not need this routine.

void svcerr_progvers()
Called when the desired version of a program is not registered with the RPC package. *low_vers* is the lowest version number, and *high_vers* is the highest version number. Service implementors usually do not need this routine.

void svcerr_systemerr()
Called by a service dispatch routine when it detects a system error not covered by any particular protocol. For example, if a service can no longer allocate storage, it may call this routine.

void svcerr_weakauth()
Called by a service dispatch routine that refuses to perform a remote procedure call due to insufficient (but correct) authentication parameters. The routine calls **svcerr_auth(xprt, AUTH_TOOWEAK)**.

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

SEE ALSO

rpc(3N), rpc_svc_calls(3N), rpc_svc_create(3N), rpc_svc_reg(3N).

NAME

rpc_svc_reg, rpc_reg, svc_reg, svc_unreg, svc_auth_reg, xprt_register, xprt_unregister - library routines for registering servers

SYNOPSIS

```
#include <rpc/rpc.h>

bool_t rpc_reg(const u_long prognum, const u_long versnum,
    const u_long procnum, const char *(*procname), const xdrproc_t inproc,
    const xdrproc_t outproc, const char *nettype);

int svc_reg(const SVCXPRT *xprt, const u_long prognum, const u_long versnum,
    const void (*dispatch), const struct netconfig *netconf);

void svc_unreg(const u_long prognum, const u_long versnum);

int svc_auth_reg(const int cred_flavor, const enum auth_stat (*handler));

void xprt_register(const SVCXPRT *xprt);

void xprt_unregister(const SVCXPRT *xprt);
```

DESCRIPTION

These routines are a part of the RPC library which allows the RPC servers to register themselves with **rpcbind()** (see *rpcbind*(1M)), and associate the given program and version number with the dispatch function. When the RPC server receives a RPC request, the library invokes the dispatch routine with the appropriate arguments.

Routines

See *rpc*(3N) for the definition of the **SVCXPRT** data structure.

bool_t rpc_reg()

Register program *prognum*, procedure *procname*, and version *versnum* with the RPC service package. If a request arrives for program *prognum*, version *versnum*, and procedure *procnum*, *procname* is called with a pointer to its parameter(s); *procname* should return a pointer to its **static** result(s); *inproc* is the XDR function used to decode the parameters while *outproc* is the XDR function used to encode the results. Procedures are registered on all available transports of the class *nettype*. See *rpc*(3N). This routine returns **0** if the registration succeeded, **-1** otherwise.

int svc_reg()

Associates *prognum* and *versnum* with the service dispatch procedure, *dispatch*. If *netconf* is **NULL**, the service is not registered with the *rpcbind* service. For example, if a service has already been registered using some other means, such as **inetd** (see *inetd*(1M)), it will not need to be registered again. If *netconf* is non-zero, then a mapping of the triple [*prognum*, *versnum*, *netconf→nc_netid*] to *xprt→xp_ltaddr* is established with the local **rpcbind** service.

The **svc_reg()** routine returns **1** if it succeeds, and **0** otherwise.

void svc_unreg()

Remove from the **rpcbind** service, all mappings of the triple [*prognum*, *versnum*, *all-transports*] to network address and all mappings within the RPC service package of the double [*prognum*, *versnum*] to dispatch routines.

int svc_auth_reg()

Registers the service authentication routine *handler* with the dispatch mechanism so that it can be invoked to authenticate RPC requests received with authentication type *cred_flavor*. This interface allows developers to add new authentication types to their RPC applications without needing to modify the libraries. Service implementors usually do not need this routine.

Typical service application would call **svc_auth_reg()** after registering the service and prior to calling **svc_run()**. When needed to process an RPC credential of type *cred_flavor*, the *handler* procedure will be called with two parameters **(struct svc_req *rqst, struct rpc_msg *msg)** and is expected to return a valid **enum auth_stat** value. There is no provision to change or delete an authentication handler once registered.

The **svc_auth_reg()** routine returns **0** if the registration is successful, **1** if *cred_flavor* already has an authentication handler registered for it, and **-1** otherwise.

void xprt_register()
After RPC service transport handle *xprt* is created, it is registered with the RPC service package. This routine modifies the global variable **svc_fdset** (see *rpc_svc_calls*(3N)). Service implementors usually do not need this routine.

void xprt_unregister()
Before an RPC service transport handle *xprt* is destroyed, it unregisters itself with the RPC service package. This routine modifies the global variable **svc_fdset** (see *rpc_svc_calls*(3N)). Service implementors usually do not need this routine.

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

SEE ALSO
inetd(1M), rpcbind(1M), rpc(3N), rpc_svc_calls(3N), rpc_svc_create(3N), rpc_svc_err(3N), rpcbind(3N), select(2).

NAME

rpc_xdr, xdr_accepted_reply, xdr_authsys_parms, xdr_callhdr, xdr_callmsg, xdr_opaque_auth, xdr_rejected_reply, xdr_replymsg - XDR library routines for remote procedure calls

SYNOPSIS

```
#include <rpc/rpc.h>
bool_t xdr_accepted_reply(XDR *xdrs, const struct accepted_reply *ar);
bool_t xdr_authsys_parms(XDR *xdrs, struct authsys_parms *aupp);
void xdr_callhdr(XDR *xdrs, struct rpc_msg *chdr);
bool_t xdr_callmsg(XDR *xdrs, struct rpc_msg *cmsg);
bool_t xdr_opaque_auth(XDR *xdrs, struct opaque_auth *ap);
bool_t xdr_rejected_reply(XDR *xdrs, const struct rejected_reply *rr);
bool_t xdr_replymsg(XDR *xdrs, const struct rpc_msg *rmsg);
```

DESCRIPTION

These routines are used for describing the RPC messages in XDR language. They should normally be used by those who do not want to use the RPC package directly. These routines return **TRUE** if they succeed, **FALSE** otherwise.

Routines

See *rpc*(3N) for the definition of the **XDR** data structure.

bool_t xdr_accepted_reply()
Used to translate between RPC reply messages and their external representation. It includes the status of the RPC call in the XDR language format. In the case of success, it also includes the call results.

bool_t xdr_authsys_parms()
Used for describing UNIX operating system credentials. It includes machine-name, uid, gid list, etc.

void xdr_callhdr()
Used for describing RPC call header messages. It encodes the static part of the call message header in the XDR language format. It includes information such as transaction ID, RPC version number, program and version number.

bool_t xdr_callmsg()
Used for describing RPC call messages. This includes all the RPC call information such as transaction ID, RPC version number, program number, version number, authentication information, etc. This is normally used by servers to determine information about the client RPC call.

bool_t xdr_opaque_auth()
Used for describing RPC opaque authentication information messages.

bool_t xdr_rejected_reply()
Used for describing RPC reply messages. It encodes the rejected RPC message in the XDR language format. The message could be rejected either because of version number mis-match or because of authentication errors.

bool_t xdr_replymsg()
Used for describing RPC reply messages. It translates between the RPC reply message and its external representation. This reply could be either an acceptance, rejection or **NULL**.

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

SEE ALSO

rpc(3N), xdr(3N).

NAME

rpcbind, rpcb_getmaps, rpcb_getaddr, rpcb_gettime, rpcb_rmtcall, rpcb_set, rpcb_unset - library routines for RPC bind service

SYNOPSIS

```
#include <rpc/rpc.h>

struct rpcblist *rpcb_getmaps(const struct netconfig *netconf,
    const char *host);

bool_t rpcb_getaddr(const u_long prognum, const u_long versnum,
    const struct netconfig *netconf, struct netbuf *svcaddr,
    const char *host);

bool_t rpcb_gettime(const char *host, time_t *timep);

enum clnt_stat rpcb_rmtcall(const struct netconfig *netconf,
    const char *host, const u_long prognum, const u_long versnum,
    const u_long procnum, const xdrproc_t inproc, const caddr_t in,
    const xdrproc_t outproc, caddr_t out, const struct timeval tout,
    struct netbuf  *svcaddr);

bool_t rpcb_set(const u_long prognum, const u_long versnum,
    const struct netconfig *netconf, const struct netbuf *svcaddr);

bool_t rpcb_unset(const u_long prognum, const u_long versnum,
    const struct netconfig *netconf);
```

DESCRIPTION

These routines allow client C programs to make procedure calls to the RPC binder service. **rpcbind** (see *rpcbind*(1M)) maintains a list of mappings between programs and their universal addresses.

Routines

struct rpcblist *rpcb_getmaps()

An interface to the **rpcbind** service, which returns a list of the current RPC program-to-address mappings on *host*. It uses the transport specified through *netconf* to contact the remote **rpcbind** service on *host*. This routine will return **NULL**, if the remote **rpcbind** could not be contacted.

bool_t rpcb_getaddr()

An interface to the **rpcbind** service, which finds the address of the service on *host* that is registered with program number *prognum*, version *versnum*, and speaks the transport protocol associated with *netconf*. The address found is returned in *svcaddr*. *svcaddr* should be preallocated. This routine returns **TRUE** if it succeeds. A return value of **FALSE** means that the mapping does not exist or that the RPC system failed to contact the remote **rpcbind** service. In the latter case, the global variable **rpc_createerr** (see *rpc_clnt_create*(3N)) contains the RPC status.

bool_t rpcb_gettime()

This routine returns the time on *host* in *timep*. If *host* is **NULL**, **rpcb_gettime()** returns the time on its own machine. This routine returns **TRUE** if it succeeds, **FALSE** if it fails. **rpcb_gettime()** can be used to synchronize the time between the client and the remote server. This routine is particularly useful for secure RPC.

enum clnt_stat rpcb_rmtcall()

An interface to the **rpcbind** service, which instructs **rpcbind** on *host* to make an RPC call on your behalf to a procedure on that host. The **netconfig** structure should correspond to a connectionless transport. The parameter ***svcaddr** will be modified to the server's address if the procedure succeeds (see **rpc_call()** and **clnt_call()** in *rpc_clnt_calls*(3N) for the definitions of other parameters).

This procedure should normally be used for a **ping** and nothing else. This routine allows programs to do lookup and call, all in one step.

Note: Even if the server is not running **rpcbind** does not return any error messages to the caller. In such a case, the caller times out.

Note: **rpcb_rmtcall()** is only available for connectionless transports.

bool_t rpcb_set()

An interface to the **rpcbind** service, which establishes a mapping between the triple [*prognum*, *versnum*, *netconf*→*nc_netid*] and *svcaddr* on the machine's **rpcbind** service. The value of *nc_netid*

must correspond to a network identifier that is defined by the netconfig database. This routine returns **TRUE** if it succeeds, **FALSE** otherwise. (See also **svc_reg()** in *rpc_svc_calls*(3N)). If there already exists such an entry with **rpcbind**, **rpcb_set()** will fail.

bool_t rpcb_unset()
An interface to the **rpcbind** service, which destroys the mapping between the triple [*prognum*, *versnum*, *netconf*→*nc_netid*] and the address on the machine's **rpcbind** service. If *netconf* is **NULL**, **rpcb_unset()** destroys all mapping between the triple [*prognum*, *versnum*, *all-transports*] and the addresses on the machine's **rpcbind** service. This routine returns **TRUE** if it succeeds, **FALSE** otherwise. Only the owner of the service or the super-user can destroy the mapping. (See also **svc_unreg()** in *rpc_svc_calls*(3N)).

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

SEE ALSO
rpcbind(1M), rpcinfo(1M), rpc_clnt_calls(3N), rpc_svc_calls(3N).

NAME

rstat(), havedisk() - get performance data from remote kernel

SYNOPSIS

```
#include <time.h>
#include <rpcsvc/rstat.h>

int havedisk(char *host);

int rstat(char *host, struct statstime *statp);
```

DESCRIPTION

havedisk() returns **1** if *host* has a disk, **0** if it does not, and –1 if this cannot be determined. The *host* string is either the official name of the host or an alias for it. See *hosts*(4) for more information regarding host names.

rstat() fills in the *statstime* structure for *host*, and returns **0** if it was successful. The relevant structures are:

```
struct stats {                  /* RSTATVERS_ORIG */
    int cp_time[CPUSTATES];     /* the time spent in each CPU state */
    int dk_xfer[DK_NDRIVE];     /* total number of disk transfers
                                   on each of the disk interfaces */
    unsigned v_pgpgin;          /* total VM pages paged in */
    unsigned v_pgpgout;         /* total VM pages paged out */
    unsigned v_pswpin;          /* total VM pages paged swapped in */
    unsigned v_pswpout;         /* total VM pages paged swapped out */
    unsigned v_intr;            /* total interrupts */
    int if_ipackets;            /* inbound packets on all interfaces */
    int if_ierrors;             /* inbound errors on all interfaces */
    int if_opackets;            /* outbound packets on all interfaces */
    int if_oerrors;             /* outbound errors on all interfaces */
    int if_collisions;          /* collisions seen on all interfaces */
};

struct statsswtch {             /* RSTATVERS_SWTCH */
    int cp_time[CPUSTATES];     /* the time spent in each CPU state */
    int dk_xfer[DK_NDRIVE];     /* total number of disk transfers
                                   on each of the disk interfaces */
    unsigned v_pgpgin;          /* total VM pages paged in */
    unsigned v_pgpgout;         /* total VM pages paged out */
    unsigned v_pswpin;          /* total VM pages paged swapped in */
    unsigned v_pswpout;         /* total VM pages paged swapped out */
    unsigned v_intr;            /* total interrupts */
    int if_ipackets;            /* inbound packets on all interfaces */
    int if_ierrors;             /* inbound errors on all interfaces */
    int if_opackets;            /* outbound packets on all interfaces */
    int if_oerrors;             /* outbound errors on all interfaces */
    int if_collisions;          /* collisions seen on all interfaces */
    unsigned v_swtch;           /* total context switches */
    long avenrun[3];            /* average number of running jobs */
    struct timeval boottime;    /* time of last boot */
};

struct statstime {              /* RSTATVERS_TIME */
    int cp_time[CPUSTATES];     /* the time spent in each CPU state */
    int dk_xfer[DK_NDRIVE];     /* total number of disk transfers
                                   on each of the disk interfaces */
    unsigned v_pgpgin;          /* total VM pages paged in */
    unsigned v_pgpgout;         /* total VM pages paged out */
    unsigned v_pswpin;          /* total VM pages paged swapped in */
    unsigned v_pswpout;         /* total VM pages paged swapped out */
    unsigned v_intr;            /* total interrupts */
    int if_ipackets;            /* inbound packets on all interfaces */
```

```
        int if_ierrors;          /* inbound errors on all interfaces */
        int if_opackets;         /* outbound packets on all interfaces */
        int if_oerrors;          /* outbound errors on all interfaces */
        int if_collisions;       /* collisions seen on all interfaces */
        unsigned v_swtch;        /* total context switches */
        long avenrun[3];         /* average number of running jobs */
        struct timeval boottime; /* time of last boot */
        struct timeval curtime;  /* current system time */
    };
```

RPC Info

Program number:

RSTATPROG

XDR routines:

```
int xdr_stats(xdrs, stat)
        XDR *xdrs;
        struct stats *stat;
int xdr_statsswtch(xdrs, stat)
        XDR *xdrs;
        struct statsswtch *stat;
int xdr_statstime(xdrs, stat)
        XDR *xdrs;
        struct statstime *stat;
int xdr_timeval(xdrs, tv)
        XDR *xdrs;
        struct timeval *tv;
```

Procs:

RSTATPROC_HAVEDISK
Takes no arguments, returns *long* which is true if remote host has a disk.

RSTATPROC_STATS
Takes no arguments, return *struct statsxxx*, depending on version.

Versions:

RSTATVERS_ORIG
RSTATVERS_SWTCH
RSTATVERS_TIME

WARNINGS

User applications that call this routine must be linked with **/usr/lib/librpcsvc.a**. For example,

```
cc my_source.c -lrpcsvc
```

AUTHOR

rstat() was developed by Sun Microsystems, Inc.

SEE ALSO

rup(1), rstatd(1M).

NAME
rwall() - write to specified remote machines

SYNOPSIS
```
#include <rpcsvc/rwall.h>
int rwall(char *host, char *msg);
```

DESCRIPTION
rwall() causes *host* to print the string *msg* to all its users. It returns 0 if successful.

RPC Info
Program number:

WALLPROG

Procs:

WALLPROC_WALL
Takes string as argument (wrapstring), returns no arguments. Executes *wall* on remote host with string.

Versions:

RSTATVERS_ORIG

WARNINGS
User applications that call this routine must be linked with **/usr/lib/librpcsvc.a**. For example,

```
cc my_source.c -lrpcsvc
```

AUTHOR
rwall() was developed by Sun Microsystems, Inc.

SEE ALSO
rwall(1M), rwalld(1M), shutdown(1M).

r

NAME

scalb() - scale exponent of a radix-independent floating-point number

SYNOPSIS

```
#include <math.h>

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

DESCRIPTION

The **scalb()** function returns $x * r^y$, where r is the radix of the machine's floating-point arithmetic. When r is 2 (as it is on all PA-RISC systems), **scalb()** is equivalent to **ldexp()**.

The **scalb()** function is recommended by the IEEE-754 standard for floating-point arithmetic. The ISO/ANSI C committee has approved the **scalb()** function for inclusion in the C9X draft standard.

To use this functions, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

Upon successful completion, the **scalb()** function returns $x * r^y$.

scalb(-x, y) and **-scalb(x, y)** are equivalent.

If y is nonintegral, **scalb()** returns a NaN.

If y is ±zero, **scalb()** returns x.

If x is +zero and y is +INFINITY, **scalb()** returns a NaN and raises the invalid exception.

If x is +zero and y is an integer less than +INFINITY, **scalb()** returns +zero.

If x is +INFINITY and y is an integer greater than -INFINITY, **scalb()** returns +INFINITY.

If x is finite and positive-signed and y is -INFINITY, **scalb()** returns +zero.

If x is positive and y is +INFINITY, **scalb()** returns +INFINITY.

If x is +INFINITY and y is -INFINITY, **scalb()** returns a NaN.

If x or y is NaN, **scalb()** returns a NaN.

If the correct value after rounding would be smaller in magnitude than **MINDOUBLE**, **scalb()** returns zero.

If the correct value would overflow, **scalb()** returns ±**HUGE_VAL** (according to the sign of x) and sets **errno** to [ERANGE].

ERRORS

If **scalb()** fails, **errno** is set to the following value.

[ERANGE] The correct value would overflow.

SEE ALSO

scalbn(3M), ilogb(3M), ldexp(3M), logb(3M), math(5), values(5).

STANDARDS CONFORMANCE

scalb(): SVID3, XPG4.2

NAME

scalbn() - scale exponent of a radix-independent floating-point number

SYNOPSIS

```
#include <math.h>

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

DESCRIPTION

The **scalbn()** function returns $x * r^n$, where r is the radix of the machine's floating-point arithmetic. When r is 2 (as it is on all PA-RISC systems), **scalbn()** is equivalent to **ldexp()**.

The **scalb()** function is recommended by the IEEE-754 standard for floating-point arithmetic. The ISO/ANSI C committee has approved the **scalbn()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

Upon successful completion, the **scalbn()** function returns $x * r^n$.

scalbn(-x, n) and **-scalbn(x, n)** are equivalent.

If x is ±INFINITY, zero, or a NaN, **scalbn()** returns x.

If the correct value after rounding would be smaller in magnitude than **MINDOUBLE**, **scalbn()** returns zero.

If the correct value would overflow, **scalbn()** returns ±**INFINITY** (according to the sign of x).

ERRORS

No errors are defined.

SEE ALSO

scalb(3M), ilogb(3M), ldexp(3M), logb(3M), math(5), values(5).

S

NAME

scandir(), alphasort() - scan a directory

SYNOPSIS

```
#include <dirent.h>

int scandir(
     const char *dirname,
     struct dirent ***namelist,
     int (*select)(const struct dirent *),
     int (*compar)(
          const struct dirent **,
          const struct dirent **
     )
);

int alphasort(
     const struct dirent **d1,
     const struct dirent **d2
);
```

DESCRIPTION

scandir() reads the directory *dirname* and builds an array of pointers to directory entries using **malloc()** (see *malloc*(3C)). It returns the number of entries in the array and a pointer to the array through *namelist*.

The *select* parameter is a pointer to a user-supplied subroutine which is called by **scandir()** to select which entries are to be included in the array. The select routine is passed a pointer to a directory entry and should return a non-zero value if the directory entry is to be included in the array. If *select* is null, then all the directory entries will be included.

The *compar* parameter is a pointer to a user-supplied subroutine which is passed to *qsort*(3C) to sort the completed array. If this pointer is null, the array is not sorted. **alphasort()** is a routine which can be used for the *compar* parameter to sort the array alphabetically.

The memory allocated for the array can be deallocated with **free()** (see *malloc*(3C)) by freeing each pointer in the array and the array itself.

APPLICATION USAGE

scandir() is thread-safe. It is not async-cancel-safe. A cancellation point may occur when a thread is executing **scandir()**.

EXTERNAL INFLUENCES

Locale

S The **LC_COLLATE** category determines the collation ordering used by **alphasort()**. See *hpnls*(5) for a description of supported collation features.

The **LC_CTYPE** category determines the interpretation of bytes in the file name portion of directory entries as single- and/or multi-byte characters by the **alphasort()** function.

Results are undefined if the locales specified by the **LC_COLLATE** and **LC_CTYPE** categories use different code sets.

International Code Set Support

Single- and multi-byte character code sets are supported for **alphasort()**.

RETURN VALUE

scandir() returns –1 if the directory cannot be opened for reading or if **malloc()** cannot allocate enough memory to hold all the data structures.

EXAMPLE

The example program below scans the **/tmp** directory. It does not exclude any entries since *select* is NULL. The contents of **namelist** are sorted by **alphasort()**. It prints out how many entries are in **/tmp** and the sorted entries of the **/tmp** directory. The memory used by **scandir()** is returned using **free()**.

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

extern int scandir();
extern int alphasort();

main()
{
    int num_entries, i;
    struct dirent **namelist, **list;

    if ((num_entries =
              scandir("/tmp", &namelist, NULL, alphasort)) < 0) {
        fprintf(stderr, "Unexpected error\n");
        exit(1);
    }
    printf("Number of entries is %d\n", num_entries);
    if (num_entries) {
        printf("Entries are:");
        for (i=0, list=namelist; i<num_entries; i++) {
            printf(" %s", (*list)->d_name);
            free(*list);
            list++;
        }
        free(namelist);
        printf("\n");
    }
    printf("\n");
    exit(0);
}
```

SEE ALSO

directory(3C), malloc(3C), qsort(3C), string(3C), dirent(5), hpnls(5).

S

NAME

scanf, fscanf, sscanf - formatted input conversion, read from stream file

SYNOPSIS

```
#include <stdio.h>
int scanf(const char *format, /* [pointer,] */ ...);
int fscanf(FILE *stream, const char *format, /* [pointer,] */ ...);
int sscanf(const char *s, const char *format, /* [pointer,] */ ...);
```

DESCRIPTION

scanf() reads from the standard input stream *stdin*.

fscanf() reads from the named input *stream*.

sscanf() reads from the character string *s*.

Each function reads characters, interprets them according to the control string *format* argument, and stores the results in its *pointer* arguments. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are ignored. The control string contains conversion specifications and other characters used to direct interpretation of input sequences. The control string contains:

- White-space characters (blanks, tabs, newlines, or formfeeds) that cause input to be read up to the next non-white-space character (except in two cases described below).
- An ordinary character (not **%**) that must match the next character of the input stream.
- Conversion specifications, consisting of the character **%**, an optional assignment suppressing character *****, an optional numerical maximum-field width, an optional **l** (ell), **ll**, (ell ell), **h** or **L** indicating the size of the receiving variable, and a conversion code.
- The conversion specification can alternatively be prefixed by the character sequence **%***n***$** instead of the character **%**, where *n* is a decimal integer in the range (1–**{NL_ARGMAX}**) (**NL_ARGMAX** is defined in <**limits.h**>). The **%***n***$** construction indicates that the value of the next input field should be placed in the *n*th argument, rather than to the next unused one. The two forms of introducing a conversion specification, **%** and **%***n***$**, must not be mixed within a single *format* string with the following exception: Skip fields (see below) can be designated as **%*** or **%***n***$***. In the latter case, *n* is ignored.

Unless the specification contains the **n** conversion character (described below), a conversion specification directs the conversion of the next input field. The result of a conversion specification is placed in the variable to which the corresponding argument points, unless ***** indicates assignment suppression. Assignment suppression provides a way to describe an input field 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 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 unsigned integer pointer.
x, **X**	A hexadecimal integer is expected; the corresponding argument should be an unsigned integer pointer. The **x** and **X** conversion characters are equivalent.
i	An integer is expected; the corresponding argument should be an integer pointer. The value of the next input item, interpreted according to C conventions, will be stored; a leading **0** implies octal, a leading **0x** implies hexadecimal; otherwise, decimal is assumed.

n Cause the total number of bytes (including white space) scanned since the function call to be stored; the corresponding argument should be an integer pointer. No input is consumed. The function return value does not include **%***n* assignments in the count of successfully matched and assigned input items.

e,**E**,**f**,**g**,**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 radix character, followed by an optional exponent field consisting of an **E** or an **e**, followed by an optional **+**, **-**, or space, followed by an integer. The conversion characters **E** and **G** behave the same as, respectively, **e** and **g**. The **e**, **E**, **f**, **g**, and **G** conversions will convert the string **inf** (case insensitive) or the string **infinity** (case insensitive) to the appropriate floating point infinity value (single, double, or quadruple precision, as specified by the conversion precision modifier). The **e**, **E**, **f**, **g**, and **G** conversions will convert the string **nan** (case insensitive) to the appropriate floating point **NaN** value (single, double, or quadruple precision, as specified by the conversion precision modifier).

C A character is expected; the corresponding argument should be a **wchar_t** pointer. The normal skip-over-white-space is suppressed in this case; to read the next non-space character, use **%1S**. The character is read and converted into a wide character according to the setting of **LC_CTYPE**. If a field width is given, the corresponding argument refers to a wide character array; the indicated number of characters is read and converted.

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 refers to a character array; the indicated number of characters is read.

S A character string is expected; the corresponding argument should be a **wchar_t** pointer pointing to an array of wide characters large enough to accept the string and a terminating **(wchar_t)0**, which is added automatically. Characters are read and converted into wide characters according to the setting of **LC_CTYPE**. The input field is terminated by a white-space character. **scanf()** cannot read a null string.

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 is added automatically. The input field is terminated by a white-space character. **scanf()** cannot read a null string.

[Indicates string data and the normal skip-over-leading-white-space is suppressed. The left bracket is followed by a set of characters, called 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. Construction of the *scanset* follows certain conventions. A range of characters may be represented by the construct *first-last*, enabling [0123456789] to be expressed [0-9]. Using this convention, *first* must be lexically less than or equal to *last*; otherwise, the dash stands for itself. The dash also stands for itself when 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, in which case it will not be interpreted syntactically 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 are added automatically. At least one character must match for this conversion to succeed.

p A sequence of unsigned hexadecimal numbers is expected. This sequence may be produced by the **p** conversion character of **printf()**. The corresponding argument shall be a pointer to a pointer to **void** into which the value represented by the hexadecimal sequence is stored. The behavior of this conversion is undefined for any input item other than a value converted earlier during the same program execution.

The conversion characters **d**, **i**, and **n** can be preceded by **l**, **ll** or **h** to indicate that a pointer to a **long int**, **long long int** or **short int** rather than to an **int** is in the argument list. Similarly, the conversion characters **u**, **o**, **x**, and **X** can be preceded by **l**, **ll** or **h** to indicate that a pointer to **unsigned long int**, **unsigned long long int**, or **unsigned short int** rather than to an **unsigned int**, is in the argument list. Finally, the conversion characters **e**, **E**, **f**, **g**, and **G** can be

preceded by **l** or **L** to indicate that a pointer to a **double** or **long double** rather than to a **float** is in the argument list. The **l**, **ll**, **L** or **h** modifier is ignored for other conversion characters.

The **scanf()** functions terminate their conversions 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.

EXTERNAL INFLUENCES

Locale

The **LC_CTYPE** category determines the interpretation of ordinary characters within format strings as single and/or multi-byte characters. Field width is given in terms of bytes. Characters received from the input stream are interpreted as single- or multi-byte characters as determined by the **LC_TYPE** category and the field width is decremented by the length of the character.

The **LC_NUMERIC** category determines the radix character expected within floating-point numbers.

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUES

If the input ends before the first conflict or conversion, EOF is returned. Otherwise, these functions return the number of successfully assigned input items. This number is a short count, or even zero if a conflict ensues between an input character and the control string.

ERRORS

scanf() and **fscanf()** fail if data needs to be read into the *stream*'s buffer, and:

[EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the process would be delayed in the read operation.

[EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for reading.

[EINTR] The read operation was terminated due to the receipt of a signal, and either no data was transferred or the implementation does not report partial transfer for this file.

[EIO] The process is a member of a background process and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the **SIGTTIN** signal or the process group of the process is orphaned.

Additional **errno** values can be set by the underlying **read()** function (see *read*(2)).

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
```

assigns to *n* the value **3**, to *i* the value **25**, to *x* the value **5.432**, and *name* contains **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
```

assigns **56** to *i*, **789.0** to *x*, skips **0123**, and places the string **56\0** in *name*. The next call to **getchar()** (see *getc*(3S)) returns **a**.

For another example, to create a language-independent date scanning routine, use:

```
char month[20]; int day, year;
(void) scanf(format, month, &day, &year);
```

For American usage, *format* would point to a string:

```
%1$s %2$d %3$d
```

The input:

```
July 3 1986
```

would assign **July** to *month*, **3** to *day* and **1986** to *year*.

For German usage, *format* would point to a string:

```
%2$d %1$s %3$d
```

The input:

3 Juli 1986

would assign **Juli** to *month*, **3** to *day* and **1986** to *year*.

The success of literal matches and suppressed assignments can be determined with the **%n** conversion specification. Here is an example that checks the success of literal matches:

```
int i, n1, n2, n3, n4;

n1 = n2 = n3 = n4 = -1;
scanf("%nBEGIN%n %d %nEND%n", &n1, &n2, &i, &n3, &n4);
if (n2 - n1 == 5) puts( "matched BEGIN");
if (n4 - n3 == 3) puts( "matched END");
```

Here is an example that checks the success of suppressed assignments:

```
int i, n1, n2;
n1 = n2 = -1;

scanf( "%d %n%*s%n", &i, &n1, &n2);
if (n2 > n1)
    printf("successful assignment suppression of %d chars\n", n2-n1);
```

APPLICATION USAGE

scanf(), **fscanf()** and **sscanf()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing **scanf()** or **fscanf()**.

WARNINGS

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

Truncation of multi-byte characters may occur if a field width is used with the conversion character.

AUTHOR

scanf() was developed by AT&T and HP.

SEE ALSO

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

STANDARDS CONFORMANCE

scanf(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

fscanf(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

sscanf(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

(ENHANCED CURSES)

NAME

scr_dump, scr_init, scr_restore, scr_set — screen file input/output functions

SYNOPSIS

```
#include <curses.h>
int scr_dump(const char *filename);
int scr_init(const char *filename);
int scr_restore(const char *filename);
int scr_set(const char *filename);
```

DESCRIPTION

The **scr_dump()** function writes the current contents of the virtual screen to the file named by *filename* in an unspecified format.

The **scr_restore()** function sets the virtual screen to the contents of the file named by *filename*, which must have been written using **scr_dump()**. The next refresh operation restores the screen to the way it looked in the dump file.

The **scr_init()** function reads the contents of the file named by *filename* and uses them to initialise the Curses data structures to what the terminal currently has on its screen. The next refresh operation bases any updates on this information, unless either of the following conditions is true:

- The terminal has been written to since the virtual screen was dumped to *filename*
- The terminfo capabilities **rmcup** and **nrrmc** are defined for the current terminal.

The **scr_set()** function is a combination of **scr_restore()** and **scr_init()**. It tells the program that the information in the file named by *filename* is what is currently on the screen, and also what the program wants on the screen. This can be thought of as a screen inheritance function.

RETURN VALUE

On successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

The **scr_init()** function is called after **initscr()** or a **system()** call to share the screen with another process that has done a **scr_dump()** after its **endwin()** call.

To read a window from a file, call **getwin()**; to write a window to a file, call **putwin()**.

SEE ALSO

S

delscreen(3X), doupdate(3X), endwin(3X), getwin(3X), open(2) (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification), read(2) (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification), write(2) (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

(ENHANCED CURSES)

NAME

scrl, wscrl — enhanced scroll a Curses window functions

SYNOPSIS

```
#include <curses.h>
int scrl(int n);
int wscrl(WINDOW *win, int n);
```

DESCRIPTION

The **scrl()** and **wscrl()** functions scroll the current or specified window. If *n* is positive, the window scrolls *n* lines toward the first line. Otherwise, the window scrolls –*n* lines toward the last line.

These functions do not change the cursor position. If scrolling is disabled for the current or specified window, these functions have no effect.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

SEE ALSO

<curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME
scroll — scroll a Curses window

SYNOPSIS
```
#include <curses.h>
int scroll(WINDOW *win);
```

DESCRIPTION
The **scroll()** function scrolls *win* one line in the direction of the first line.

This function does not change the cursor position. If scrolling is disabled for the current or specified window, this function has no effect.

RETURN VALUE
Upon successful completion, this function returns OK. Otherwise, it returns ERR.

ERRORS
No errors are defined.

SEE ALSO
scrl(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
This description has been rewritten for clarity, but otherwise its functionality is identical.

S

NAME

secure_rpc, authdes_getucred, authdes_seccreate, getnetname, host2netname, key_decryptsession, key_encryptsession, key_gendes, key_setsecret, key_secretkey_is_set, netname2host, netname2user, user2netname - library routines for secure remote procedure calls

SYNOPSIS

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

int authdes_getucred(const struct authdes_cred *adc,
    uid_t *uidp,
    gid_t *gidp,
    short *gidlenp,
    gid_t *gidlist);

AUTH *authdes_seccreate(const char *name,
    const unsigned int window,
    const char *timehost,
    const des_block *ckey);

int getnetname(char name[MAXNETNAMELEN+1]);

int host2netname(char name[MAXNETNAMELEN+1],
    const char *host,
    const char *domain);

int key_decryptsession(const char *remotename,
    des_block *deskey);

int key_encryptsession(const char *remotename,
    des_block *deskey);

int key_gendes(des_block *deskey);

int key_setsecret(const char *key);

int key_secretkey_is_set(void);

int netname2host(const char *name,
    char *host,
    const int hostlen);

int netname2user(const char *name,
    uid_t *uidp,
    gid_t *gidp,
    int *gidlenp,
    gid_t gidlist[NGROUPS]);

int user2netname(char name[MAXNETNAMELEN+1],
    const uid_t uid,
    const char *domain);
```

DESCRIPTION

RPC library routines allow C programs to make procedure calls on other machines across the network.

RPC supports various authentication flavors. Among them are:

AUTH_NONE (none) No authentication.

AUTH_SYS Traditional UNIX-style authentication.

AUTH_DES DES encryption-based authentication.

The **authdes_getucred()** and **authdes_seccreate()** routines implement the **AUTH_DES** authentication flavor. The keyserver daemon **keyserv** (see *keyserv*(1M)) must be running for the **AUTH_DES** authentication system to work, and *keylogin*(1) must have been run. Only the **AUTH_DES** style of authentication is discussed here. For information about the **AUTH_NONE** and **AUTH_SYS** styles of authentication, refer to *rpc_clnt_auth*(3N).

The routines documented on this page are MT-Safe. See the pages of the other authentication styles for their MT-level.

S

Routines

See *rpc*(3N) for the definition of the **AUTH** data structure.

int authdes_getucred()

authdes_getucred() is the first of the two routines which interface to the RPC secure authentication system known as **AUTH_DES**. The second is **authdes_seccreate()**, below. **authdes_getucred()** is used on the server side for converting an **AUTH_DES** credential, which is operating system independent, into an **AUTH_SYS** credential. This routine returns **1** if it succeeds, **0** if it fails.

***uidp** is set to the user's numerical ID associated with *adc*. ***gidp** is set to the numerical ID of the user's group. ***gidlist** contains the numerical IDs of the other groups to which the user belongs. ***gidlenp** is set to the number of valid group ID entries in ***gidlist** (see **netname2user()**, below).

Warning: **authdes_getucred()** will fail if the authdes_cred structure was created with the netname of a host. In such a case, **netname2host()** should be used on the host netname in the authdes_cred structure to get the host name.

AUTH *authdes_seccreate()

authdes_seccreate(), the second of two **AUTH_DES** authentication routines, is used on the client side to return an authentication handle that will enable the use of the secure authentication system. The first parameter *name* is the network name, or *netname*, of the owner of the server process. This field usually represents a hostname derived from the utility routine **host2netname()**, but could also represent a user name using **user2netname()**, described below.

The second field is *window* on the validity of the client credential, given in seconds. If the difference in time between the client's clock and the server's clock exceeds *window*, the server will reject the client's credentials, and the clock will have to be resynchronized. A small window is more secure than a large one, but choosing too small of a window will increase the frequency of resynchronizations because of clock drift.

The third parameter, *timehost*, the host's name, is optional. If it is **NULL**, then the authentication system will assume that the local clock is always in sync with the *timehost* clock, and will not attempt resynchronizations. If a timehost is supplied, however, then the system will consult with the remote time service whenever resynchronization is required. This parameter is usually the name of the host on which the server is running.

The final parameter *ckey* is also optional. If it is **NULL,** then the authentication system will generate a random DES key to be used for the encryption of credentials. If *ckey* is supplied, then it will be used instead.

If **authdes_seccreate()** fails, it returns **NULL**.

int getnetname()

getnetname() returns the unique, operating system independent netname of the caller in the fixed-length array *name*. Returns **1** if it succeeds, and **0** if it fails.

S

int host2netname()

Convert from a domain-specific hostname *host* to an operating system independent netname. Returns **1** if it succeeds, and **0** if it fails. Inverse of **netname2host()**. If *domain* is **NULL**, **host2netname()** uses the default domain name of the machine. If *host* is **NULL**, it defaults to that machine itself.

int key_decryptsession()

key_decryptsession() is an interface to the keyserver daemon, which is associated with RPC's secure authentication system (**AUTH_DES** authentication). User programs rarely need to call it, or its associated routines **key_encryptsession()**, **key_gendes()**, and **key_setsecret()**.

key_decryptsession() takes a server netname *remotename* and a DES key *deskey*, and decrypts the key by using the the public key of the the server and the secret key associated with the effective UID of the calling process. It is the inverse of **key_encryptsession()**.

int key_encryptsession()

key_encryptsession() is a keyserver interface routine. It takes a server netname *remotename* and a DES key *deskey*, and encrypts it using the public key of the the server and the secret key associated with the effective UID of the calling process. It is the inverse of **key_decryptsession()**. This routine returns **0** if it succeeds, **-1** if it fails.

int key_gendes()
key_gendes() is a keyserver interface routine. It is used to ask the keyserver for a secure conversation key. Choosing one at random is usually not good enough, because the common ways of choosing random numbers, such as using the current time, are very easy to guess. This routine returns **0** if it succeeds, **-1** if it fails.

int key_setsecret()
key_setsecret() is a keyserver interface routine. It is used to set the key for the effective UID of the calling process. This routine returns **0** if it succeeds, **-1** if it fails.

int key_secretkey_is_set(void)
key_secretkey_is_set() is a keyserver interface routine that may be used to determine whether a key has been set for the effective UID of the calling process. If the keyserver has a key stored for the effective UID of the calling process, this routine returns **1**. Otherwise it returns **0**.

int netname2host()
Convert from an operating system independent netname *name* to a domain-specific hostname *host*. *hostlen* is the maximum size of *host*. Returns **1** if it succeeds, and **0** if it fails. Inverse of **host2netname()**.

int netname2user()
Convert from an operating system independent netname to a domain-specific user ID. Returns **1** if it succeeds, and **0** if it fails. Inverse of **user2netname()**.

***uidp** is set to the user's numerical ID associated with *name*. ***gidp** is set to the numerical ID of the user's group. *gidlist* contains the numerical IDs of the other groups to which the user belongs. ***gidlenp** is set to the number of valid group ID entries in *gidlist*.

int user2netname()
Convert from a domain-specific username to an operating system independent netname. Returns **1** if it succeeds, and **0** if it fails. Inverse of **netname2user()**.

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

SEE ALSO
chkey(1), keyserv(1M), newkey(1M), rpc(3N), rpc_clnt_auth(3N).

S

NAME
set_term — switch between screens

SYNOPSIS
```
#include <curses.h>
SCREEN *set_term(SCREEN *new);
```

DESCRIPTION
The **set_term()** function switches between different screens. The *new* argument specifies the new current screen.

RETURN VALUE
Upon successful completion, **set_term()** returns a pointer to the previous screen. Otherwise, it returns a null pointer.

ERRORS
No errors are defined.

APPLICATION USAGE
This is the only function that manipulates **SCREEN** pointers; all other functions affect only the current screen.

SEE ALSO
Screens, Windows and Terminals in curs_intro, initscr(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The entry is rewritten for clarity.

NAME

setaclentry(), fsetaclentry() - add, modify, or delete one entry in file's access control list (ACL) (HFS File Systems only)

SYNOPSIS

```
#include <unistd.h>
#include <acllib.h>

int setaclentry(const char *path, uid_t uid, gid_t gid, int mode);

int fsetaclentry(int fd, uid_t uid, gid_t gid, int mode);
```

DESCRIPTION

Both forms of this call add, modify, or delete one entry in a file's access control list (ACL). **setaclentry()** and **fsetaclentry()** take a path name (*path*) or open file descriptor (*fd*) and an entry identifier (*uid*, *gid*). They change the indicated entry's access mode bits to the given value (*mode*), meanings of which are defined in <**unistd.h**>. *modes* are represented as **R_OK**, **W_OK**, and **X_OK**. Irrelevant bits in *mode* values must be zero.

If the file's ACL does not have an entry for the given *uid* and *gid*, the entry is created and added to the ACL. If *mode* is **MODE_DEL** (defined in <**acllib.h**>), the matching entry is deleted from the file's ACL if it is an optional entry, or its mode bits are set to zero (no access) if it is a base entry.

uid or *gid* can be **ACL_NSUSER** or **ACL_NSGROUP** (defined in <**sys/acl.h**>), respectively, to represent non-specific entries *u*.**%**, **%.***g*, or **%.%**. The file's *u*.**%** or **%.***g* base entries can be referred to using **ACL_FILEOWNER** or **ACL_FILEGROUP** (defined in <**acllib.h**>), for the file's owner or group ID, respectively.

setaclentry() and **fsetaclentry()** read the file's ACL with **getacl()** or **fgetacl()** and modify it with **setacl()** or **fsetacl()**, respectively.

APPLICATION USAGE

setaclentry() and **fsetaclentry()** are thread-safe and async-cancel-safe.

RETURN VALUE

If successful, **setaclentry()** and **fsetaclentry()** return zero.

ERRORS

If an error occurs, **setaclentry()** and **fsetaclentry()** return the following negative values and set **errno**:

–1 Unable to perform **getacl()** or **fgetacl()** on the file. **errno** indicates the cause.

–2 Unable to perform **stat()** or **fstat()** on the file. **errno** indicates the cause.

–3 Cannot add a new entry because the ACL already has **NACLENTRIES** (defined in <**sys/acl.h**>) entries.

–4 Cannot delete a nonexisting entry.

–5 Unable to perform **setacl()** or **fsetacl()** on the file. **errno** indicates the cause.

EXAMPLES

The following code fragment adds an entry to file "work/list" for user ID 115, group ID 32, or modifies the existing entry for that user and group, if any, with a new access mode of read only. It also changes the owner base entry to have all access rights, and deletes the entry, if any, for any user in group 109.

```
#include <unistd.h>
#include <acllib.h>

char *filename = "work/list";

setaclentry (filename, 115, 32, R_OK);
setaclentry (filename, ACL_FILEOWNER, ACL_NSGROUP, R_OK | W_OK | X_OK);
setaclentry (filename, ACL_NSUSER, 109, MODE_DEL);
```

DEPENDENCIES

HFS **setaclentry()** and **fsetaclentry()** are only supported on HFS file system on standard HP-UX operating system.

NFS **setaclentry()** and **fsetaclentry()** are not supported on remote files.

AUTHOR

setaclentry() and **fsetaclentry()** were developed by HP.

SEE ALSO

getacl(2), setacl(2), stat(2), acltostr(3C), cpacl(3C), chownacl(3C), strtoacl(3C), acl(5).

S

NAME

setbuf(), setvbuf() - assign buffering to a stream file

SYNOPSIS

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int type, size_t size);
```

Obsolescent Interface

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

DESCRIPTION

setbuf() can 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() can be used after a stream has been opened but before it is read or written. *type* determines how *stream* is to 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.

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 the output stream is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new-line character is written or terminal input is requested). **fflush()** can also be used to explicitly write the buffer.

If *buf* is not the NULL pointer, the array it points to is used for buffering instead of an automatically allocated buffer (from **malloc()**). *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.

Obsolescent Interface

setvbuf_unlocked() assigns buffering to a stream file.

APPLICATION USAGE

setbuf() and **setvbuf()** are thread-safe interfaces. These interfaces are not async-cancel-safe.

DIAGNOSTICS

If an illegal value for *type* or *size* is provided, **setvbuf()** and **setvbuf_unlocked()** return 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, then failing to close the stream in the same block.

SEE ALSO

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

STANDARDS CONFORMANCE

setbuf(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

setvbuf(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, ANSI C

NAME

setcat() - set the default message catalog

SYNOPSIS

```
#include <pfmt.h>
char *setcat(const char *msgcat);
```

DESCRIPTION

The **setcat()** routine sets the default message catalog for use by other formatting routines (see *gettxt*(3C) and *pfmt*(3C)).

msgcat specifies the file to use as the default catalog. The file name is limited to 14 characters. No checking is done to make sure that the file exists. The value of *msgcat* determines what happens to the default message catalog:

string	Sets the default message catalog to *string*.
NULL	Returns a pointer to the current default message catalog. The default catalog is not changed.
empty string	Resets to no catalog.

APPLICATION USAGE

setcat() is thread-safe and async-cancel-safe.

RETURN VALUE

Upon successful completion, **setcat()** returns a pointer to the default catalog. Otherwise, **setcat()** returns a NULL pointer.

EXAMPLES

The following example gets its message from the **my_appl_cat** message catalog file:

```
setcat("my_appl_cat");
pfmt(stderr, MM_INFO,":1:The file is writable");
```

SEE ALSO

gettxt(3C), pfmt(3C), setlocale(3C), pfmt(3C).

STANDARDS COMPLIANCE

setcat(): SVID3

S

(ENHANCED CURSES)

NAME

setcchar — set **cchar_t** from a wide character string and rendition

SYNOPSIS

```
#include <curses.h>

int setcchar(cchar_t *wcval, const wchar_t *wch, const attr_t attrs,
             short color_pair, const void *opts);
```

DESCRIPTION

The **setcchar()** function initialises the object pointed to by *wcval* according to the character attributes in *attrs*, the colour pair in *color_pair* and the wide character string pointed to by *wch*.

The *opts* argument is reserved for definition in a future edition of this document. Currently, the application must provide a null pointer as *opts*.

RETURN VALUE

Upon successful completion, **setcchar()** returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

SEE ALSO

Characters in curs_intro, attroff(3X), can_change_color(3X), getcchar(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

S

NAME

setclock - set value of system-wide clock

SYNOPSIS

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

int setclock(int clock_type, struct timespec *tp);
```

DESCRIPTION

The **setclock()** function sets the current value **tp** of the specified system-wide clock, **clock_type**.

setclock() supports a **clock_type** of **TIMEOFDAY**, defined in <**sys/timers.h**>, which represents the time-of-day clock for the system. For this clock, the values returned by **setclock()** represent the amount of time since the Epoch.

The calling process must have appropriate privileges to set the **TIMEOFDAY** clock.

APPLICATION USAGE

setclock() is thread-safe and async-cancel-safe.

RETURN VALUE

Upon successful completion, **setclock()** returns a value of zero; otherwise it returns –1 and sets **errno** to indicate the error.

ERRORS

setclock() fails if any of the following conditions are encountered:

[EINVAL] **clock_type** does not specify a known system-wide clock, or **tp** either is outside the range for a given clock type, or it specifies a nanosecond value less than zero or greater than or equal to 1000 million.

[EIO] An error occurred while accessing the clock device.

[EPERM] The requesting process does not have the required appropriate privileges to set the specified clock.

FILES

/usr/include/sys/timers.h

SEE ALSO

clocks(2), getclock(3C), gettimer(3C).

STANDARDS CONFORMANCE

setclock(): AES

S

NAME

setjmp(), longjmp(), sigsetjmp(), siglongjmp() - non-local goto

SYNOPSIS

```
#include <setjmp.h>
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
int _setjmp(jmp_buf env);
void _longjmp(jmp_buf env, int val);
int sigsetjmp(sigjmp_buf env, int savemask);
void siglongjmp(sigjmp_buf env, int val);
```

DESCRIPTION

setjmp() and **longjmp()** are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program. They exist in three variant forms: **setjmp()** and **longjmp()**; **_setjmp()** and **_longjmp()**; **sigsetjmp()** and **siglongjmp()**. Unless indicated otherwise, references to **setjmp()** and **longjmp()** apply to all three versions.

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()** returns **1**. All accessible data values are valid as of the time **longjmp()** is called.

Upon the return from a **setjmp()** call caused by a **longjmp()**, the values of any non-static local variables belonging to the routine from which **setjmp()** was called are undefined. Code which depends on such values is not guaranteed to be portable.

Variant Forms

The following functions behave the same as **setjmp()** and **longjmp()** except in the handling of the process' signal mask (see *sigaction*(2) and *sigvector*(2)). This distinction is only significant for programs which use **sigaction()**, **sigprocmask()**, **sigvector()**, **sigblock()**, and/or **sigsetmask()**.

setjmp()
longjmp() These always save and restore the signal mask.

_setjmp()
_longjmp() These never manipulate the signal mask.

sigsetjmp() Saves the signal mask of the calling thread if and only if *savemask* is non-zero.

siglongjmp() Restores the signal mask if and only if it is saved by **sigsetjmp()**.

Programming Considerations

If a **longjmp()** is executed and the environment in which the **setjmp()** is executed no longer exists, errors can occur. The conditions under which the environment of the **setjmp()** no longer exists include exiting the procedure that contains the **setjmp()** call, and exiting an inner block with temporary storage (such as a block with declarations in C or a **with** statement in Pascal). This condition might not be detectable, in which case the **longjmp()** occurs and, if the environment no longer exists, the contents of the temporary storage of an inner block are unpredictable. This condition might also cause unexpected process termination. If the procedure has been exited the results are unpredictable.

Passing **longjmp()** a pointer to a buffer not created by **setjmp()**, passing **_longjmp()** a pointer to a buffer not created by either **setjmp()** or **_setjmp()**, passing **siglongjmp()** a pointer to a buffer not created by **sigsetjmp()** or passing any of these three functions a buffer that has been modified by the user, can cause all the problems listed above, and more.

Some implementations of Pascal support a "try/recover" mechanism, which also creates stack marker information. If a **longjmp()** operation occurs in a scope which is nested inside a try/recover, and the corresponding **setjmp()** is not inside the scope of the try/recover, the recover block will not be executed and the currently active recover block will become the one enclosing the **setjmp()**, if one exists.

WARNINGS

A call to **longjmp()** to leave the guaranteed stack space reserved by **sigspace()** might remove the guarantee that the ordinary execution of the program will not extend into the guaranteed space. It might also cause the program to forever lose its ability to automatically increase the stack size, and the program might then be limited to the guaranteed space.

The result of using **setjmp()** within an expression can be unpredictable.

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

The interfaces **setjmp()**, **longjmp()**, **sigsetjmp()** and **siglongjmp()** are thread-safe. The effect of a call to **longjmp()** where the initialization of the **jmp_buf** argument was not performed in the calling thread is undefined. The effect of a call to **siglongjmp()** where the initialization of the **sigjmp_buf** argument was not performed in the calling thread is undefined.

The contents of the **jmp_buf** buffer are architecture and compilation environment specific. Thus, objects built using these functions may not be supported across architectures.

AUTHOR

setjmp() was developed by AT&T and HP.

SEE ALSO

sigaction(2), sigblock(2), signal(5), sigprocmask(2), sigsetmask(2), sigspace(2), sigsuspend(2), sigvector(2).

STANDARDS CONFORMANCE

setjmp(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

longjmp(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

siglongjmp(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

sigsetjmp(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

S

NAME

setlabel() - define label for formatting routines

SYNOPSIS

```
#include <pfmt.h>
int setlabel(const char *label);
```

DESCRIPTION

The **setlabel()** system call defines a label to be printed by formatting routines (see *pfmt*(3C)) in the standard message format. *label* is a character string limited to 25 characters in length.

If *label* is NULL or an empty string, the label is reset to no label. No label is defined before **setlabel()**.

setlabel() assumes that *label* has already been translated into a locale-specific string using the current locale.

APPLICATION USAGE

setlabel() is thread-safe. It is not async-cancel-safe.

RETURN VALUE

setlabel() returns zero upon successful completion or nonzero if the routine failed.

EXAMPLES

This example, with no label defined:

```
pfmt(stderr, MM_INFO,"my_appl_cat:1:file is writable");
```

generates:

```
INFO: file is writable
```

Using **setlabel()**:

```
setlabel("my_appl");
pfmt(stderr, MM_INFO,"my_appl_cat:1:file is writable");
```

generates:

```
my_appl: INFO: file is writable
```

SEE ALSO

pfmt(3C).

STANDARDS COMPLIANCE

setlabel(): SVID3

S

NAME

setlocale( ), getlocale( ) - set and get the locale of a program

SYNOPSIS

```
#include <locale.h>
char *setlocale(int category, const char *locale);
struct locale_data *getlocale(int type);
```

Obsolescent Interfaces

```
int  setlocale_r(int category, const char *locale, char *buffer,
     int buflen);
int  getlocale_r(int type, struct locale_data *ld);
```

DESCRIPTION

The **setlocale()** function sets, queries, or restores the aspect of a program's locale that is specified by the *category* argument. A program's locale refers to those areas of the program's Native Language Support (NLS) environment for which the following values of *category* have been defined:

LC_ALL Affects behavior of all categories below, as well as all *nl_langinfo*(3C) items.

LC_COLLATE Affects behavior of regular expressions and the NLS string collation functions (see *string*(3C), and *regexp*(5)).

LC_CTYPE Affects behavior of regular expressions, character classification, and conversion functions (see *ctype*(3C), *wctype*(3C), *wconv*(3C), *conv*(3C), and *regexp*(5)). The **LC_CTYPE** category also affects the behavior of all routines that process multibyte characters (see *multibyte*(3C)).

LC_MESSAGES Affects the language in which messages are displayed and the processing of affirmative and negative responses (see *catopen*(3C) and *nl_langinfo*(3C)).

LC_MONETARY Affects behavior of functions that handle monetary values (see *localeconv*(3C) and *strfmon*(3C)).

LC_NUMERIC Affects handling of the radix character in the formatted input/output functions (see *printf*(3S), *scanf*(3S) and *vprintf*(3S)) and the string conversion functions (see *ecvt*(3C) and *strtod*(3C)). **LC_NUMERIC** also affects the numeric values found in the *localeconv* structure.

LC_TIME Affects the behavior of time conversion functions (see *getdate*(3C), *strftime*(3C), and *strptime*(3C)).

All *nl_langinfo*(3C) items are affected by the setting of one of the categories listed above. See *langinfo*(5) to determine which categories affect each item.

S The value of the *locale* argument determines the action taken by **setlocale()**. *locale* is a pointer to a character string.

Note that while **setlocale()** can be called concurrently from multiple threads, the locale data structures that get updated are not protected against reads by other threads. Hence it is up to the application developer to ensure proper synchronization when changing locales.

Setting the Locale of a Program

To set the program's locale for *category*, **setlocale()** accepts one of the following values as the *locale* argument: *locale name*, **C**, **POSIX**, or **""** (the empty string). The actions prescribed by these values are as follows:

locale name If *locale* is a valid locale name (see *lang*(5)), **setlocale()** sets that part of the NLS environment associated with *category* as defined for that locale.

C If the value of *locale* is set to **C**, **setlocale()** sets that part of the NLS environment associated with *category* as defined for the **C** locale (see *lang*(5)). The **C** locale is the default prior to successfully calling **setlocale()**.

POSIX Same as **C**.

"" If the value of *locale* is the empty string, the setting of that part of the NLS environment associated with *category* depends upon the setting of the following environment variables in the user's environment (see *environ*(5)):

LANG	**LC_MESSAGES**
LC_ALL	**LC_MONETARY**
LC_COLLATE	**LC_NUMERIC**
LC_CTYPE	**LC_TIME**

If *category* is any defined value other than **LC_ALL**, **setlocale()** sets that category as specified by the value of the **LC_ALL** environment. This is also the case if **LC_ALL** is not set to the corresponding environment variable. If the environment variable is not set or is set to the empty string, **setlocale()** sets the category as specified by the value of the **LANG** environment variable. If **LANG** is not set or is set to the empty string, then **setlocale()** sets the category to the **C** locale. For example,

```
setlocale(LC_TIME,"")
```

sets the program's NLS environment associated with the **LC_TIME** *category* to the value specified by the user's **LC_TIME** environment variable. All other aspects of the NLS environment are unaffected.

If *category* is **LC_ALL**, then all categories are set corresponding to the value of **LC_ALL** if **LC_ALL** is set, or **LANG** if **LC_ALL** is not set, except for those categories in which the corresponding environment variable is set to a valid language name (see *lang*(5)). In this case the value of the environment variable overrides the values of **LC_ALL** and **LANG** for that category. If the values of both **LC_ALL** and **LANG** are not set or are set to the empty string, then the **C** locale is used.

The following usage of **setlocale()** results in the program's locale being set according to the the user's language requirements:

```
setlocale(LC_ALL,"")
```

Querying the Locale of a Program

setlocale() queries the current NLS environment pertaining to *category*, if the value of *locale* is NULL. The query operation does not change the environment. The purpose of performing a query is to save that aspect of the user's current NLS environment associated with *category*, in the value returned by **setlocale()**, such that it can be restored with a subsequent call to **setlocale()**.

Restoring the Locale of a Program

To restore a category within the program locale, a **setlocale()** call is made with the same *category* argument and the return string of the previous **setlocale()** call given as the *locale* argument.

getlocale() returns a pointer to a **locale_data** structure (see **/usr/include/locale.h**). The members of the **locale_data** structure contain information about the setting of each **setlocale()** category. *type* determines what information is contained in the **locale_data** structure. The only supported value of *type* is:

LOCALE_STATUS The structure member corresponding to each category contains a string with the name of the locale currently set for that category. The string does not include modifier information.

Obsolescent Interfaces

setlocale_r() and **getlocale_r()** set and get the locale of a program..

RETURN VALUE

If the pointer to a string is given for *locale* and the selection can be honored, the **setlocale()** function returns a pointer to the string associated with the specified *category* for the new locale. The maximum length of this string is **LC_BUFSIZ** bytes (see **<locale.h>**). If the selection cannot be honored, the **setlocale()** function returns a null pointer and the program's locale is not changed.

A null pointer for *locale* causes **setlocale()** to return a string associated with the *category* for the program's current locale.

The string returned by **setlocale()** is such that a subsequent call with that string as the *locale* argument and its associated *category* restores that part of the program's locale.

ERRORS

If a language name given through the *locale* argument does not identify a valid language name or the language is not available on the system (see *lang*(5)) a null pointer is returned and the program's locale is not changed. The same behavior occurs when the call:

```
setlocale(LC_ALL, "");
```

is made and any category related environment variable in the user's environment identifies an invalid language name or a language that is not available on the system.

If the *category* argument is not a defined category value, a null pointer is returned and the program's locale is not changed.

setlocale() returns a string that reflects the current setting of that aspect of the NLS environment corresponding to the *category* argument. If this return string is used in a subsequent **setlocale()** call and the *category* arguments of the two calls do not match, the locale remains unchanged and a null pointer is returned.

EXAMPLES

To set a program's entire locale based on the language requirements specified via the user's environment variables:

```
setlocale(LC_ALL, "");
```

If in the example the user's environment variables were set as follows:

```
LANG ="de_DE.iso88591"
LC_COLLATE ="es_ES.iso88591"
LC_MONETARY =""
LC_TIME ="en_US.iso88591"
```

the **LC_CTYPE**, **LC_MONETARY**, and **LC_NUMERIC** category items would be set to correspond to the **de_DE.iso88591** language definition, the **LC_COLLATE** category items would be set to correspond to the **es_ES.iso88591** language definition for collation and the **LC_TIME** category items would be set corresponding to the **en_US.iso88591** language definition.

Using the same example, if the following call was made:

```
struct locale_data *locale_info=getlocale(LOCALE_STATUS);
```

the contents of ***locale_info** would be:

```
locale_info->LC_ALL_D="de_DE.iso88591"
locale_info->LC_COLLATE_D="es_ES.iso88591"
locale_info->LC_CTYPE_D="de_DE.iso88591"
locale_info->LC_MESSAGES_D="de_DE.iso88591"
locale_info->LC_MONETARY_D="de_DE.iso88591"
locale_info->LC_NUMERIC_D="de_DE.iso88591"
locale_info->LC_TIME_D="en_US.iso88591"
```

The next example shows the precedence of the **LC_ALL** environment variable:

```
setlocale(LC_ALL, "");
```

with the following settings in the user's environment:

```
LANG=de_DE.iso88591
LC_ALL=fr_FR.iso88591
```

All categories will be loaded with **fr_FR.iso88591**.

Another example showing the precedence of the **LC_ALL** environment variable:

```
setlocale(LC_CTYPE, "");
```

with the following settings in the user's environment:

```
LANG=tr_TR.iso88599
LC_ALL=da_DK.iso88591
LC_CTYPE=ru_RU.iso88595
```

All categories will be loaded with **da_DK.iso88591**.

Another example with the **LC_ALL** environment variable:

```
setlocale(LC_TIME, "pl_PL.iso88592");
```

with the following settings in the user's environment:

```
LANG=it_IT.iso88591
LC_ALL=nl_NL.iso88591
```

The **LC_TIME** category will be set to **pl_PL.iso88592**, but all other categories will be set to **nl_NL.iso88591**.

To set the date/time formats to **fr_FR.iso88591**:

```
setlocale(LC_TIME, "fr_FR.iso88591");
```

To set the collating sequence to the **C** locale:

```
setlocale(LC_COLLATE, "C");
```

To set monetary handling to the value of the user's **LC_MONETARY** environment variable:

```
setlocale(LC_MONETARY, "");
```

Note that if the **LC_MONETARY** environment variable is not set or is empty, the value of the user's **LANG** environment variable is used.

To query a user's locale:

```
char *ch = setlocale(LC_ALL, NULL);
```

To restore the locale saved in the above example:

```
setlocale(LC_ALL, ch);
```

To query only the part of the user's locale pertaining to the **LC_NUMERIC** category:

```
char *ch = setlocale(LC_NUMERIC, NULL);
```

To restore the **LC_NUMERIC** category of the user's locale saved in the above example:

```
setlocale(LC_NUMERIC, ch);
```

WARNINGS

The format of the return string from **setlocale()** is implementation specific, is not standardized across vendor's platforms, and is subject to change in future releases. The return string is valid only for the life of the process invoking the **setlocale()** and should only be used for restoring a previously stored locale setting within the same process.

Using **getenv()** as the *locale* argument is not recommended. An example of such incorrect usage is:

```
setlocale(LC_ALL, getenv("LANG"));
```

getenv() returns a character string which can be a language name, an empty string, or a null pointer; depending on the setting of the user's **LANG** environment variable. Each of these values as the *locale* argument define a specific action to be taken by **setlocale()**. Therefore the action taken by **setlocale()** depends upon the value returned from the **getenv()** call. To ensure that **setlocale()** sets the program's locale based upon the setting of the user's environment variables the following usage is recommended:

```
setlocale(LC_ALL, "");
```

The value returned by **setlocale()** points to an area that is overwritten during the next call to **setlocale()**. Be sure to copy these values to another area if they are to be used after a subsequent **setlocale()** call.

getlocale() is an HP proprietary interface, which will be *obsoleted* in a future release, and is not portable to other vendor's platforms.

The structure returned through a call to **getlocale()** is overwritten during the next call to **getlocale()**. Be sure to save these values if they are to be used after a subsequent **getlocale()** call.

getlocale() and **setlocale()** are thread-safe. It should be noted that the locale state is common to all threads within a process.

S

getlocale_r() and **setlocale_r()** are obsolescent interfaces supported for compatibility with existing DCE applications. New multithreaded applications should use **getlocale()** and **setlocale()**.

Any program that calls **setlocale()** before **catopen()** with the *oflag* parameter set to **NL_CAT_LOCALE** may behave differently in this release than in previous releases because of the addition of **LC_MESSAGES** to **XPG4**. In the past, **catopen()** was directed to the desired language by **LANG**. Now, **catopen()** with the *oflag* parameter set to **NL_CAT_LOCALE**, is controlled by **LC_MESSAGES**. **setlocale()** can modify the **LC_MESSAGES** category. For example, if the environment variables are set as follows:

```
LC_MESSAGES="fr_FR.iso88591"
```

and the following call to **setlocale()** is made:

```
setlocale(LC_ALL, "de_DE.iso88591");
```

followed by a call to **catopen()**, then **catopen()** will open the message catalogs for **de_DE.iso88591** rather than **fr_FR.iso88591**.

If you use **setlocale()** and compile/link your application archive, please note that **setlocale()** has a dependency on **libdld.sl** that will require a change to the compile/link command.

Compile:

cc -Wl,-a,archive -Wl,-E -Wl,+n -l:libdld.sl -o *outfile source*

Or compile with **CCOPTS** and **LDOPTS**:

export CCOPTS="-Wl,-a,archive *options* **-Wl,-E -l:libdld.sl"**

export LDOPTS="*options* **-E +n -l:libdld.sl"**

cc -o *outfile source*

The option **-Wl,-a,archive** is positionally dependent and should occur at the beginning of the compile line. For optimum compatibility in future releases, avoid using archive **libc** with other shared libraries except for **libdld.sl** as needed above.

AUTHOR

setlocale(), **setlocale_r()**, **getlocale()**, and **getlocale_r()** were developed by OSF and HP.

FILES

/usr/include/langinfo.h
/usr/include/locale.h

SEE ALSO

locale(1), localedef(1M), conv(3C), ctype(3C), ecvt(3C), getdate(3C), multibyte(3C), nl_langinfo(3C), regcomp(3C), string(3C), perror(3C), strfmon(3C), strftime(3C), string(3C), strptime(3C), strtod(3C), wcsftime(3C), wcstring(3C), printf(3S), scanf(3S), vprintf(3S), wconv(3C), wctype(3C), wcstod(3C), wcstol(3C), environ(5), hpnls(5), lang(5), langinfo(5).

STANDARDS COMPLIANCE

setlocale(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

shl_load(), shl_definesym(), shl_findsym(), shl_gethandle(), shl_getsymbols(), shl_unload(), shl_get(), shl_gethandle_r(), shl_get_r() - explicit load of shared libraries

SYNOPSIS

```
#include <dl.h>

shl_t shl_load(const char *path, int flags, long address);

int shl_findsym(
     shl_t *handle,
     const char *sym,
     short type,
     void *value
);

int shl_definesym(
     const char *sym,
     short type,
     long value,
     int flags
);

int shl_getsymbols(
     shl_t handle,
     short type,
     int flags,
     void *(*memory) (),
     struct shl_symbol **symbols,
);

int shl_unload(shl_t handle);

int shl_get(int index, struct shl_descriptor **desc);

int shl_gethandle(shl_t handle, struct shl_descriptor **desc);

int shl_get_r(int index, struct shl_descriptor *desc);

int shl_gethandle_r(shl_t handle, struct shl_descriptor *desc);
```

MULTITHREAD USAGE

These routines are safe to be called from multithreaded applications.

DESCRIPTION

These routines can be used to programmatically load and unload shared libraries, and to obtain information about the libraries (such as the addresses of symbols defined within them). On PA32, the routines themselves are accessed by specifying the **-ldld** option on the command line with the **cc** or **ld** command (see *cc*(1) and *ld*(1)). On PA64, the routines are accessed by specifying either **-ldld** or **-ldl** on the command line. See *WARNINGS*. In addition, the **-E** option to the **ld** command can be used to ensure that all symbols defined in the program are available to the loaded libraries. This is the default behavior on PA64.

Shared libraries are created by compiling source files and linking the resultant object files with the **-b** (create shared library) option.

shl_load() Attaches the shared library named by *path* **or** the shared library name that is constructed by using the path part of *path* plus the shared library basename followed by the suffix **.0** (e.g. **/usr/lib/pa20_64/libname.0**) to the process, along with all its dependent libraries. A **.0** version is looked for first for those shared libraries that do not have internal names. See *ld*(1)). The library is mapped at the specified *address*. If *address* is **0L**, the system chooses an appropriate address for the library. This is the recommended practice because the system has the most complete knowledge of the address space; currently, the *address* field is ignored, and assumed to be **0L**. If the shared library contains thread local storage, you cannot load it with this routine. The flags argument is made up of several fields. One of the following must be specified:

S

BIND_IMMEDIATE Resolve symbol references when the library is loaded.

BIND_DEFERRED Delay code symbol resolution until actual reference.

Zero or more of the following can be specified by doing a bitwise OR operation:

BIND_FIRST Place the library at the head of the symbol search order. In default mode, the library and its dependent libraries are bound independently of each other (see **BIND_TOGETHER**).

BIND_NONFATAL Default **BIND_IMMEDIATE** behavior is to treat all unsatisfied symbols as fatal. This flag allows binding of unsatisfied code symbols to be deferred until use.

BIND_NOSTART Do not call the initializers for the shared library when the library is loaded, nor on a future call to **shl_unload()** or **dlclose()**; by default, all the initializers registered with the specified library are invoked upon loading.

BIND_VERBOSE Print verbose messages concerning possible unsatisfied symbols.

BIND_RESTRICTED Restrict symbols visible to the library to those present at the time the library is loaded.

DYNAMIC_PATH Allow the loader to dynamically search for the library specified by the *path* argument. The directories to be searched are determined by the **+s** and **+b** options of the **ld** command used when the program was linked. On PA64, this is enabled by default if **ld +compat** was *not* specified.

BIND_TOGETHER When used with **BIND_FIRST**, the library being mapped and its dependent libraries will be bound together. This is the default behavior for all **shl_load()** requests not using **BIND_FIRST**.

If successful, **shl_load()** returns a handle which can be used in subsequent calls to **shl_findsym()**, **shl_unload()**, **shl_gethandle()**, or **shl_gethandle_r()**; otherwise NULL is returned. On PA64, the handle can also be used in subsequent calls to **dlclose()** or **dlsym()**.

shl_findsym() Obtains the address of an exported symbol *sym* from a shared library. The *handle* argument should be a pointer to the handle of a loaded shared library that was returned from a previous call to **shl_load()**, or **shl_get()**. On PA64, you can also get *handle* from a call to **dlopen()**. If a pointer to NULL is passed for this argument, **shl_findsym()** searches all user defined symbols (those defined by **shl_definesym()**), all currently loaded shared libraries and the program to find the symbol; otherwise **shl_findsym()** searches only the specified shared library. The return value of handle will be NULL if the symbol found was generated via **shl_definesym()**. Otherwise the handle of the library where the symbol was found is returned. The special handle **PROG_HANDLE** can be used to refer to the program itself, so that symbols exported from the program can also be accessed dynamically. The *type* argument specifies the expected type for the symbol, and should be one of the defined constants **TYPE_PROCEDURE**, **TYPE_DATA**, or **TYPE_UNDEFINED**. The latter value suppresses type checking. These are the only accepted type arguments on PA64. On PA32, you can also specify **TYPE_STORAGE**, or **TYPE_TSTORAGE**. The address of the symbol is returned in the variable pointed to by *value*. If the symbol is a thread local storage symbol, the address of the symbol is the value of the thread pointer + the starting address of the shared library + the offset of the symbol in the library. This routine returns 0 if successful; otherwise –1 is returned. See **DIAGNOSTICS** for errno settings.

shl_definesym()
Adds a symbol to the user hash table for the current process. If *value* falls in the range of a currently loaded library, an association will be made and the symbol is

undefined once the associated library is unloaded. The defined symbol can be overridden by a subsequent call to this routine or by loading a more visible library that provides a definition. Symbols overridden in this manner may become visible again if the overriding definition is removed.

Possible symbol types include:

TYPE_PROCEDURE Symbol is a function.

TYPE_DATA Symbol is data.

Possible flag values include: None defined at the present time. Zero should be passed in to prevent conflicts with future uses of this flag. You cannot use this routine to define a thread local storage symbol. **dlsym()** will not search for any user defined symbol.

shl_getsymbols()

Provides an array of symbol records, allocated using the supplied memory allocator, that are associated with the library specified by *handle*. If the *handle* argument is a pointer to NULL, symbols defined using **shl_definesym()** are returned. If multiple versions of the same symbol have been defined with **shl_definesym()**, only the version from the specified symbol information source that would be considered for symbol binding is returned. The *type* argument is used to restrict the return information to a specific type. Values of **TYPE_PROCEDURE** and **TYPE_DATA** can be used to limit the returned symbols to be either code or data respectively. On PA32, you can also specify a type of **TYPE_STORAGE** or **TYPE_TSTORAGE**. The **TYPE_DATA** value cause both data, storage, and tstorage symbols to be returned. The constant **TYPE_UNDEFINED** can be used to return all symbols, regardless of type. The *flags* argument must have one of the following values:

IMPORT_SYMBOLS
Return symbols found on the import list.

EXPORT_SYMBOLS
Return symbols found on the export list. All symbols defined by **shl_definesym()** are export symbols.

INITIALIZERS
Return symbols that are specified as the initializers of the library.

Zero or more of the following can be specified by doing a bitwise OR operation:

NO_VALUES Only makes sense when combined with **EXPORT_SYMBOLS** or **INITIALIZERS**. Do not calculate the value field in the return structure to avoid symbol binding by the loader to resolve symbol dependencies. If only a few symbol values are needed, **shl_findsym()** can be used to find the values of interesting symbols. This is not to be used with **GLOBAL_VALUES**.

GLOBAL_VALUES
Only makes sense when combined with **EXPORT_SYMBOLS** or **INITIALIZERS**. Use the name and type information of each return symbol and find the most visible occurrence using all symbol information sources. The value and handle fields in the symbol return structure reflect where the most visible occurrence was found. Not to be used with **NO_VALUES**.

The *memory* argument should point to a function with the same interface as **malloc()** (see *malloc*(3C)).

The return information consists of an array of the following records (defined in **<dl.h>**):

```
struct shl_symbol {
    char *name,
    short type,
    void  *value,
    shl_t handle,
```

```
};
```

The *type* field in the return structure can have the values **TYPE_PROCEDURE**, or **TYPE_DATA**. On PA32, you can also have the values **TYPE_STORAGE** or **TYPE_TSTORAGE**. These are a subset of **TYPE_DATA**. The *value* and *handle* fields are only valid if export symbols are requested and the **NO_VALUES** flag is not specified. The *value* field contains the address of the symbol, while the *handle* field is the handle of the library that defined the symbol, or NULL for symbols defined via the **shl_definesym()** routine and is useful in conjunction with the **GLOBAL_VALUES** flag.

If successful, **shl_getsymbols()** returns the number of symbols found; otherwise it returns –1.

shl_unload() Can be used to detach a shared library from the process. The *handle* argument should be the handle returned from a previous call to **shl_load()**. On PA64, you can also get the handle from a call to **dlopen()**. **shl_unload()** returns 0 if successful; otherwise –1 is returned. Any initializers registered with the library are called before detachment. All explicitly loaded libraries are detached automatically on process termination.

shl_get() Returns information about currently loaded libraries, including those loaded implicitly at startup time. The index argument is the ordinal position of the shared library in the shared library search list for the process. A subsequent call to **shl_unload()** or **dlclose()** decrements the index values of all libraries having an index greater than the unloaded library. The index value –1 refers to the dynamic loader. The *desc* argument is used to return a pointer to a statically allocated buffer containing a descriptor for the shared library. The format of the descriptor is implementation dependent; to examine its format, look at the contents of file **/usr/include/dl.h**. Information common to all implementations includes the library handle, pathname, and the range of addresses the library occupies. The buffer for the descriptor used by **shl_get()** is static; the contents should be copied elsewhere before a subsequent call to the routine. The routine returns 0 normally, or –1 if an invalid *index* is given.

shl_gethandle()
Returns information about the library specified by the handle argument. The special handle **PROG_HANDLE** can be used to refer to the program itself. The descriptor returned is the same as the one returned by the **shl_get()** routine. The buffer for the descriptor used by **shl_gethandle()** is static; the contents should be copied elsewhere before a subsequent call to the routine. The routine returns 0 normally, or –1 on error.

shl_get_r() This is a reentrant version of **shl_get()**. The *desc* argument must point to a buffer of enough user-defined storage to be filled with the library descriptor described in **/usr/include/dl.h**. Its semantics are otherwise identical to **shl_get()**.

S

shl_gethandle_r()
This is a reentrant version of **shl_gethandle()**. The *desc* argument must point to a buffer of enough user-defined storage to be filled with the library descriptor described in **/usr/include/dl.h**. Its semantics are otherwise identical to **shl_gethandle()**.

DIAGNOSTICS

If a library cannot be loaded, **shl_load()** returns NULL and sets **errno** to indicate the error. This includes trying to **shl_load()** a library containing thread local storage. All other functions return –1 on error and set **errno**.

If **shl_findsym()** cannot find the indicated symbol, **errno** is set to zero. On PA32 only, if **shl_findsym()** finds the indicated symbol but cannot resolve all the symbols it depends on, **errno** is set to ENOSYM.

ERRORS

Possible values for **errno** include:

[ENOEXEC] The specified file is not a shared library, or a format error was detected.

[ENOSYM]	Some symbol required by the shared library could not be found.
[EINVAL]	The specified handle or index is not valid or an attempt was made to load a library at an invalid address.
[ENOMEM]	There is insufficient room in the address space to load the library.
[ENOENT]	The specified library does not exist.
[EACCES]	Read or execute permission is denied for the specified library.

WARNINGS

shl_unload() detaches the library from the process and frees the memory allocated for it, but does not break existing symbolic linkages into the library. In this respect, an unloaded shared library is much like a block of memory deallocated via **free()** (see *malloc*(3C)).

Some implementations may not, by default, export all symbols defined by a program (instead exporting only those symbols that are imported by a shared library seen at link time). Therefore the **-E** option to *ld*(1) should be used when using these routines if the loaded libraries are to refer to program symbols.

All symbol information returned by **shl_getsymbols()**, including the name field, become invalid once the associated library is unloaded by **shl_unload()**. On PA64, this is also true if a library is unloaded by a call to **dlclose()**.

When a routine or flag is used which may not be supported in the future, the dynamic loader can display a warning message. See the *WARNINGS* section in *dld.sl*(5) for further details.

Future HP-UX 64-bit environments may not support these routines and flags or may only support a subset of them. Instead, they will use the SVR4 dynamic loading API. Users are encouraged to migrate to the **dl*** family of dynamic linking routines. See the *dlclose*(3C), *dlerror*(3C), *dlget*(3C), *dlgetname*(3C), *dlmodinfo*(3C), *dlopen*(3C), and *dlsym*(3C) man pages for more information.

AUTHOR

shl_load(3X) and related functions were developed by HP.

SEE ALSO

System Tools:

ld(1)	invoke the link editor

Miscellaneous:

dld.sl(5)	dynamic loader
dlclose(3C)	unload a shared library previously loaded by dlopen()
dlerror(3C)	print the last error message recorded by dld
dlget(3C)	return information about a loaded module
dlgetname(3C)	return the name of the storage containing a load module
dlmodinfo(3C)	return information about a loaded module
dlopen(3C)	load a shared library
dlsym(3C)	get the address of a symbol in a shared library

Texts and Tutorials

HP-UX Linker and Libraries Online User Guide
(See the **+help** option to *ld*(1))

HP-UX Linker and Libraries User's Guide
(See *manuals*(5) for ordering information)

S

NAME

signbit( ) - floating-point sign-determination macro

SYNOPSIS

```
#include <math.h>
int signbit(floating-type x);
```

DESCRIPTION

The **signbit()** macro determines whether the sign of its argument value is negative. The macro can be used with either **double** or **float** arguments, including infinities, zeros, and NaNs.

The ISO/ANSI C committee has approved the **signbit()** macro for inclusion in the C9X draft standard.

To use the **signbit()** macro, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

The **signbit()** macro returns a nonzero value if and only if the sign of its argument value is negative.

ERRORS

No errors are defined.

EXAMPLE

Take certain actions if **x** is negative zero:

```
#include <math.h>
/*...*/
   double x;
   /*...*/
   if ( (x == 0.0) && signbit(x) ) {
      /*...*/
   }
```

SEE ALSO

finite(3M), fpclassify(3M), isfinite(3M), isinf(3M), isnan(3M), isnormal(3M), math(5).

S

NAME

sigpause - atomically release blocked signals and wait for interrupt

SYNOPSIS

```
#include <signal.h>

long sigpause(long mask);
```

DESCRIPTION

sigpause() blocks signals according to the value of *mask* in the same manner as *sigsetmask*(2), then atomically waits for an unmasked signal to arrive. On return, **sigpause()** restores the current signal mask to the value that existed before the **sigpause()** call. When no signals are to be blocked, a value of **0L** is used for *mask*.

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

RETURN VALUE

sigpause() terminates when it is interrupted by a signal. When **sigpause()** terminates, it returns –l and sets **errno** to EINTR.

EXAMPLES

The following call to **sigpause()** waits until the calling process receives a signal:

```
sigpause (0L);
```

The following example blocks the SIGIO signal until **sigpause()** is called. When a signal is received at the **sigpause()** statement, the signal mask is restored to its value before **sigpause()** was called:

```
long savemask;
savemask = sigblock (sigmask (SIGIO));
/* critical section */
sigpause (savemask);
```

WARNINGS

Check all references to *signal*(5) for appropriateness on systems that support *sigvector*(2). **sigvector()** can affect the behavior described on this page.

Do not use **sigpause()** in conjunction with the facilities described under *sigset*(3C).

APPLICATION USAGE

Threads Considerations

Since blocked signal masks are maintained at the thread level, **sigpause()** modifies only the calling thread's blocked signal mask. **sigpause()** suspends only the calling thread until it receives a signal.

If other threads in the process do not block the signal, the signal may be delivered to another thread in the process and the thread in **sigpause()** may continue waiting. For this reason, the use of *sigwait*(2) is recommended instead of **sigpause()** for multi-threaded applications.

For more information regarding signals and threads, refer to *signal*(5).

LP64 Programs

sigpause() accepts a long (64 bit) value. However, as for ILP32 programs, **sigpause()** supports signals numbered 1 through 32. The upper 32 bits of the mask argument are ignored.

AUTHOR

sigpause() was developed by the University of California, Berkeley.

SEE ALSO

sigblock(2), sigsetmask(2), sigsuspend(2), sigvector(2), sigwait(2).

NAME

sigset, sighold, sigrelse, sigignore, sigpause - signal management

SYNOPSIS

```
#include <signal.h>
void (*sigset(int sig, void (*func)(int)))(int);
int sighold(int sig);
int sigrelse(int sig);
int sigignore(int sig);
int sigpause(int sig);
```

DESCRIPTION

The system defines a set of signals that can be delivered to a process. The set of signals is defined in *signal*(5), along with the meaning and side effects of each signal. An alternate mechanism for handling these signals is defined here. The facilities described here should not be used in conjunction with the other facilities described under *signal*(2), *sigvector*(2), *sigblock*(2), *sigsetmask*(2), *sigpause*(3C) and *sigspace*(2).

sigset() allows the calling process to choose one of four ways to handle the receipt of a specific signal. *sig* specifies the signal and *func* specifies the choice.

sig can be any one of the signals described under *signal*(5) except **SIGKILL** or **SIGSTOP**.

func is assigned one of four values: **SIG_DFL**, **SIG_IGN**, **SIG_HOLD**, or a *function address*. The actions prescribed by **SIG_DFL** and **SIG_IGN** are described under *signal*(5). The action prescribed by **SIG_HOLD** and *function address* are described below:

SIG_HOLD Hold signal.

The signal *sig* is held upon receipt. Any pending signal of this signal type remains held. Only one signal of each type is held.

Note: the signals **SIGKILL**, **SIGCONT**, and **SIGSTOP** cannot be held.

function address

Catch signal.

func must be a pointer to a function, the signal-catching handler, that is called when signal *sig* occurs. **sigset()** specifies that the process calls this function upon receipt of signal *sig*. Any pending signal of this type is released. This handler address is retained across calls to the other signal management functions listed here. Upon receipt of signal *sig*, the receiving process executes the signal-catching function pointed to by *func* as described under *signal*(5) with the following differences:

Before calling the signal-catching handler, the system signal action of *sig* is set to **SIG_HOLD**. During a normal return from the signal-catching handler, the system signal action is restored to *func* and any held signal of this type is released. If a non-local goto (*longjmp*(3C)) is taken, **sigrelse()** must be called to restore the system signal action to *func* and release any held signal of this type.

sighold() holds the signal *sig*. **sigrelse()** restores the system signal action of *sig* to that specified previously by **sigset()**. **sighold()** and **sigrelse()** are used to establish critical regions of code. **sighold()** is analogous to raising the priority level and deferring or holding a signal until the priority is lowered by **sigrelse()**.

sigignore() sets the action for signal *sig* to **SIG_IGN** (see *signal*(5)).

sigpause() suspends the calling process until it receives an unblocked signal. If the signal *sig* is held, it is released before the process pauses. **sigpause()** is useful for testing variables that are changed when a signal occurs. For example, **sighold()** should be used to block the signal first, then test the variables. If they have not changed, call **sigpause()** to wait for the signal.

RETURN VALUE

Upon successful completion, **sigset()** returns the previous value of the system signal action for the specified signal *sig*. Otherwise, a value of **SIG_ERR** is returned and **errno** is set to indicate the error. **SIG_ERR** is defined in <**signal.h**>.

For the other functions, a 0 value indicates that the call succeeded. A -1 return value indicates an error occurred and **errno** is set to indicate the reason.

ERRORS

sigset() fails and the system signal action for *sig* is not changed if any of the following occur:

[EFAULT] The *func* argument points to memory that is not a valid part of the process address space. Reliable detection of this error is implementation dependent.

sigset(), **sighold()**, **sigrelse()**, **sigignore()**, and **sigpause()** fail and the system signal action for *sig* is not changed if any of the following occur:

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

[EINVAL] An attempt is made to ignore, hold, or supply a handler for a signal that cannot be ignored, held, or caught; see *signal*(5).

sigpause returns when the following occurs:

[EINTR] A signal was caught.

WARNINGS

These signal facilities should not be used in conjunction with *bsdproc*(3C), *signal*(2), *sigvector*(2), *sigblock*(2), *sigsetmask*(2), *sigpause*(3C) and *sigspace*(2).

SEE ALSO

kill(1), kill(2), signal(2), pause(2), wait(2), abort(3C), setjmp(3C), signal(5).

STANDARDS CONFORMANCE

sigset(): SVID2, SVID3

sighold(): SVID2, SVID3

sigignore(): SVID2, SVID3

sigpause(): SVID2, SVID3

sigrelse(): SVID2, SVID3

S

NAME

sigemptyset(), sigfillset(), sigaddset(), sigdelset(), sigismember() - initialize, manipulate, and test signal sets

SYNOPSIS

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(const sigset_t *set, int signo);
```

DESCRIPTION

sigemptyset() initializes the signal set pointed to by *set*, to exclude all signals supported by HP-UX.

sigfillset() initializes the signal set pointed to by *set*, to include all signals supported by HP-UX.

Applications must call either **sigemptyset()** or **sigfillset()** at least once for each object of type **sigset_t** before using that object for anything else, including cases where the object is returned from a function (for example, the *oset* argument to **sigprocmask()** — see *sigprocmask*(2)).

sigaddset() adds the signal specified by *signo* to the signal set pointed to by *set*.

sigdelset() deletes the signal specified by *signo* from the signal set pointed to by *set*.

sigismember() tests whether the signal specified by *signo* is a member of the signal set pointed to by *set*.

RETURN VALUE

Upon successful completion, **sigismember()** returns a value of 1 if the specified signal is a member of the specified set, or a value of 0 if it is not. The other functions return a value of 0 upon successful completion. For all of the above functions, if an error is detected, a value of –1 is returned and **errno** is set to indicate the error.

ERRORS

sigaddset(), **sigdelset()**, and **sigismember()** fail if the following is true:

[EINVAL] The value of the *signo* argument is out of range.

WARNINGS

The above functions do not detect a bad address passed in for the *set* argument. A segmentation fault is the most likely result.

S

AUTHOR

sigfillset(), **sigemptyset()**, **sigaddset()**, **sigdelset()**, and **sigismember()** were derived from the *IEEE Standard POSIX 1003.1-1988*.

SEE ALSO

sigaction(2), sigsuspend(2), sigpending(2), sigprocmask(2), signal(5).

STANDARDS CONFORMANCE

sigaddset(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

sigdelset(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

sigemptyset(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

sigfillset(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

sigismember(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

sin(), sinf() - sine functions

SYNOPSIS

```
#include <math.h>
double sin(double x);
float sinf(float x);
```

DESCRIPTION

sin() returns the sine of *x* (*x* specified in radians).

sin() may lose accuracy when *x* is far from zero.

sinf() is a **float** version of **sin()**; it takes a **float** argument and returns a **float** result. To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options.

sinf() is not specified by any standard, but it is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, make sure your program includes <**math.h**>, and link in the math library by specifying **-lm** on the compiler or linker command line.

Millicode versions of the **sin()** and **sinf()** functions are available. Millicode versions of math library functions are usually faster than their counterparts in the standard library. To use these versions, compile your program with the **+Olibcalls** or the **+Oaggressive** optimization option.

For special cases, the millicode versions return the same values as their standard library counterparts (see the RETURN VALUE section).

For more information, see the *HP-UX Floating-Point Guide*.

RETURN VALUE

If *x* is NaN or ±INFINITY, **sin()** returns NaN.

If the correct value after rounding would be smaller in magnitude than **MINDOUBLE**, **sin()** returns zero.

ERRORS

No errors are defined.

SEE ALSO

acos(3M), asin(3M), atan(3M), atan2(3M), cos(3M), sind(3M), tan(3M), math(5), values(5).

STANDARDS CONFORMANCE

sin(): SVID3, XPG4.2, ANSI C

S

NAME

sind(), sindf() - degree-valued sine functions

SYNOPSIS

```
#include <math.h>
double sind(double x);
float sindf(float x);
```

DESCRIPTION

sind() is a degree-valued version of the **sin()** function. It returns the sine of *x* (*x* specified in degrees).

sind() may lose accuracy when *x* is far from zero.

sindf() is a **float** version of **sind()**; it takes a **float** argument and returns a **float** result.

sind() and **sindf()** are not specified by any standard, but **sindf()** is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Llink in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

If *x* is NaN or ±INFINITY, **sind()** returns NaN.

If the correct value after rounding would be smaller in magnitude than **MINDOUBLE**, **sind()** returns zero.

ERRORS

No errors are defined.

SEE ALSO

acosd(3M), asind(3M), atand(3M), atan2d(3M), cosd(3M), sin(3M), tand(3M), math(5), values(5).

S

NAME
sinh(), sinhf() - hyperbolic sine functions

SYNOPSIS
```
#include <math.h>
double sinh(double x);
float sinhf(float x);
```

DESCRIPTION
sinh() returns the hyperbolic sine of its argument.

sinhf() is a **float** version of **sinh()**; it takes a **float** argument and returns a **float** result. To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options.

sinhf() is not specified by any standard, but it is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, make sure your program includes <**math.h**>, and link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE
If *x* is ±INFINITY, **sinh()** returns ±INFINITY respectively.

If *x* is NaN, **sinh()** returns NaN.

If the correct value after rounding would be smaller in magnitude than **MINDOUBLE**, **sinh()** returns zero.

If the correct value would overflow, **sinh()** returns **HUGE_VAL** (it returns –**HUGE_VAL** for negative *x*) and sets **errno** to [ERANGE].

ERRORS
If **sinh()** fails, **errno** is set to the following value.

[ERANGE] The correct value would overflow.

SEE ALSO
asinh(3M), cosh(3M), tanh(3M), math(5), values(5).

STANDARDS CONFORMANCE
sinh(): SVID3, XPG4.2, ANSI C

S

NAME

sleep() - suspend execution for interval

SYNOPSIS

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```

DESCRIPTION

sleep() suspends the current process from execution for the number of *seconds* specified by the argument.

Actual suspension time can be less than that requested for two reasons:

- Scheduled wakeups occur at fixed 1-second intervals (on the second, according to an internal clock), and
- Any caught signal terminates the *sleep* following execution of that signal's catching routine.

Suspension time can be an arbitrary amount longer than requested due to the scheduling of other activity in the system. The value returned by **sleep()** is 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.

seconds must be less than 2^{32}.

APPLICATION USAGE

sleep() is thread-safe. It is not async-cancel-safe. A cancellation point may occur when a thread is executing **sleep()**.

If a SIGALRM is generated for a multi-threaded process, it may not be delivered to a thread currently in **sleep()**. See *sigwait(2)* man page for details. In a multi-threaded process delivery of a SIGALRM to a thread in **sleep()** simply causes **sleep()** to return without invoking the SIGALRM handler.

SEE ALSO

sigwait(2), signal(5).

STANDARDS CONFORMANCE

sleep(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

S

NAME

slk_attroff, slk_attr_off, slk_attron, slk_attr_on, slk_attrset, slk_attr_set, slk_clear, slk_color, slk_init, slk_label, slk_noutrefresh, slk_refresh, slk_restore, slk_set, slk_touch, slk_wset — soft label functions

SYNOPSIS

```
#include <curses.h>
int slk_attroff(const chtype attrs);
int slk_attr_off(const attr_t attrs, void *opts);
int slk_attron(const chtype attrs);
int slk_attr_on(const attr_t attrs, void *opts);
int slk_attrset(const chtype attrs);
int slk_attr_set(const attr_t attrs, short color_pair, void *opts);
int slk_clear(void);
int slk_color(short color_pair);
int slk_init(int fmt);
char *slk_label(int labnum);
int slk_noutrefresh(void);
int slk_refresh(void);
int slk_restore(void);
int slk_set(int labnum, const char *label, int justify);
int slk_touch(void);
int slk_wset(int labnum, const wchar_t *label, int justify);
```

DESCRIPTION

The Curses interface manipulates the set of soft function-key labels that exist on many terminals. For those terminals that do not have soft labels, Curses takes over the bottom line of *stdscr*, reducing the size of *stdscr* and the value of the **LINES** external variable. There can be up to eight labels of up to eight display columns each.

To use soft labels, **slk_init()** must be called before **initscr()**, **newterm()** or **ripoffline()** is called. If **initscr()** eventually uses a line from *stdscr* to emulate the soft labels, then *fmt* determines how the labels are arranged on the screen. Setting *fmt* to 0 indicates a 3-2-3 arrangement of the labels; 1 indicates a 4-4 arrangement. Other values for *fmt* are unspecified.

The **slk_init()** function has the effect of calling **ripoffline()** to reserve one screen line to accommodate the requested format.

The **slk_set()** and **slk_wset()** functions specify the text of soft label number *labnum*, within the range from 1 to and including 8. The *label* argument is the string to be put on the label. With **slk_set()**, and **slk_wset()**, the width of the label is limited to eight column positions. A null string or a null pointer specifies a blank label. The *justify* argument can have the following values to indicate how to justify *label* within the space reserved for it:

- **0** Align the start of *label* with the start of the space
- **1** Center *label* within the space
- **2** Align the end of *label* with the end of the space

The **slk_refresh()** and **slk_noutrefresh()** functions correspond to the **wrefresh()** and **wnoutrefresh()** functions.

The **slk_label()** function obtains soft label number *labnum*.

The **slk_clear()** function immediately clears the soft labels from the screen.

The **slk_restore()** function immediately restores the soft labels to the screen after a call to **slk_clear()**.

The **slk_touch()** function forces all the soft labels to be output the next time **slk_noutrefresh()** or **slk_refresh()** is called.

The **slk_attron()**, **slk_attrset()** and **slk_attroff()** functions correspond to **attron()**, **attrset()**, and **attroff()**. They have an effect only if soft labels are simulated on the bottom line of the screen.

The **slk_attr_off()**, **slk_attr_on()**, **slk_attr_set()** and **slk_color()** functions correspond to **attr_off()**, **attr_on()**, **attr_set()** and **color_set()**, respectively and thus support the attribute constants with the WA_ prefix and color.

RETURN VALUE

Upon successful completion, **slk_label()** returns the requested label with leading and trailing blanks stripped. Otherwise, it returns a null pointer.

Upon successful completion, the other functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

When using multi-byte character sets, applications should check the width of the string by calling **mbstowcs()** and then **wcswidth()** before calling **slk_set()**. When using wide characters, applications should check the width of the string by calling **wcswidth()** before calling **slk_set()**.

Since the number of columns that a wide character string will occupy is codeset-specific, call **wcwidth()** and **wcswidth()** to check the number of column positions in the string before calling **slk_wset()**.

Most applications would use **slk_noutrefresh()** because a **wrefresh()** is likely to follow soon.

SEE ALSO

attr_get(), attroff(), delscreen(), mbstowcs() (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification), ripoffline(), wcswidth() (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

S

NAME

spray - scatter data in order to check the network

SYNOPSIS

```
#include <time.h>
#include <rpcsvc/spray.h>
```

DESCRIPTION

This reference page describes the data structures and XDR routines used by the *spray*(1M) program. A **spray()** function call does not exist. Refer to *spray*(1M) for more information.

RPC Information

Program number:

SPRAYPROG

XDR routines:

```
xdr_sprayarr(xdrs, arr);
        XDR *xdrs;
        struct sprayarr *arr;
xdr_spraycumul(xdrs, cumul);
        XDR *xdrs;
        struct spraycumul *cumul;
```

Procedures:

SPRAYPROC_SPRAY

Takes no arguments, returns no value. Increments a counter in server daemon. The server does not return this call, so the caller should have a timeout of 0. The sprayarr is only used by the caller, to vary the size of the UDP packets sent.

SPRAYPROC_GET

Takes no arguments, returns *struct spraycumul* with the values of counter and clock set to reflect the number of SPRAYPROC_SPRAY calls, and the total time (seconds and microseconds) elapsed since the last SPRAYPROC_CLEAR request.

SPRAYPROC_CLEAR

Takes no arguments and returns no value. Zeros out counter and clock in preparation for calls to SPRAYPROC_SPRAY.

Versions:

SPRAYVERS_ORIG

Structures:

```
struct spraycumul {
        unsigned counter;
        struct timeval clock;
};
struct sprayarr {
        int *data;
        int lnth;
};
```

WARNING

User applications that call this routine must be linked with **/usr/include/librpcsvc.a**. For example,

```
cc my_source.c -lrpcsvc
```

AUTHOR

spray was developed by Sun Microsystems, Inc.

SEE ALSO

spray(1M), sprayd(1M).

NAME

sqrt(), sqrtf() - square root functions

SYNOPSIS

```
#include <math.h>
double sqrt(double x);
float sqrtf(float x);
```

DESCRIPTION

sqrt() returns the non-negative square root of *x*. The value of *x* must not be less than zero.

sqrtf() is a **float** version of **sqrt()**; it takes a **float** argument and returns a **float** result. To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options.

sqrtf() is not specified by any standard, but it is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, make sure your program includes <**math.h**>, and link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

If *x* is +INFINITY, **sqrt()** returns +INFINITY.

If *x* is NaN, **sqrt()** returns NaN.

If *x* is less than zero, **sqrt()** returns NaN and sets **errno** to [EDOM].

ERRORS

If **sqrt()** fails, **errno** is set to the following value.

[EDOM] *x* is less than zero.

SEE ALSO

cbrt(3M), exp(3M), log(3M), pow(3M), math(5).

STANDARDS CONFORMANCE

sqrt(): SVID3, XPG4.2, ANSI C

S

NAME

ssignal(), gsignal() - software signals

SYNOPSIS

```
#include <signal.h>
int (*ssignal(int sig, int (*action)(int)))(int);
int gsignal(int sig);
```

DESCRIPTION

ssignal() and **gsignal()** implement a software facility similar to *signal*(5). 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*, 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.

APPLICATION USAGE

The interfaces **ssignal()** and **gsignal()** are thread-safe. These interfaces are not async-cancel-safe.

SEE ALSO

signal(5).

NOTES

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

STANDARDS CONFORMANCE

ssignal(): SVID2, SVID3, XPG2

gsignal(): SVID2, SVID3, XPG2

NAME

standend, standout, wstandend, wstandout — set and clear window attributes

SYNOPSIS

```
#include <curses.h>
int standend(void);
int standout(void);
int wstandend(WINDOW *win);
int wstandout(WINDOW *win);
```

DESCRIPTION

The **standend()** and **wstandend()** functions turn off all attributes of the current or specified window.

The **standout()** and **wstandout()** functions turn on the standout attribute of the current or specified window.

RETURN VALUE

These functions always return 1.

ERRORS

No errors are defined.

SEE ALSO

attroff(3X), attr_get(3X), <curses.h>.

CHANGE HISTORY

Derived from the **attroff()** entry in X/Open Curses, Issue 3. The entry is reworded for clarity, but otherwise the functionality is identical to previous issues.

S

NAME

statfsdev, fstatfsdev - get file system statistics

SYNOPSIS

```
#include <sys/vfs.h>
int statfsdev(const char *path, struct statfs *buf);
int fstatfsdev(int fildes, struct statfs *buf);
```

DESCRIPTION

statfsdev() returns information about the file system on the file specified by *path*.

buf is a pointer to a **statfs** structure into which information is placed concerning the file system. The contents of the structure pointed to by *buf* include the following members:

```
long  f_bavail    /* free blocks available to non-superuser */
long  f_bfree     /* free blocks */
long  f_blocks    /* total blocks in file system */
long  f_bsize     /* fundamental file system block size in bytes */
long  f_ffree     /* free file nodes in file system */
long  f_files     /* total file nodes in file system */
long  f_type      /* type of info, zero for now */
fsid_t  f_fsid    /* file system ID.  f_fsid[1] is MOUNT_UFS,
                                      MOUNT_NFS, or MOUNT_CDFS */
```

Fields that are undefined for a particular file system are set to –1 .

fstatfsdev() returns the same information as above, but about the open file referred to by file descriptor *fildes*.

APPLICATION USAGE

statfsdev() and **fstatfsdev()** are thread-safe.

RETURN VALUE

Upon successful completion, **statfsdev()** and **fstatfsdev()** return zero. Otherwise, they return –1 and set the global variable **errno** to indicate the error.

ERRORS

statfsdev() fails if one or more of the following conditions are encountered:

[EACCES] Search permission is denied for a component of the path prefix.

[EAGAIN] The file exists, enforcement mode file/record locking is set, and there are outstanding record locks on the file.

[EFAULT] *path* points to an invalid address.

[ELOOP] Too many symbolic links are encountered in translating the path name.

[EMFILE] The maximum number of file descriptors allowed are currently open.

[ENAMETOOLONG] The length of the specified path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.

[ENFILE] The system file table is full.

[ENOENT] The named file does not exist.

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

[ENXIO] The device specified by the named special file does not exist.

fstatfsdev() fails if one or more of the following is true:

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

[ESPIPE] *filedes* points to an invalid address.

S

Both **fstatfsdev()** and **statfsdev()** fail if one or more of the following is true:

[EAGAIN]	Enforcement-mode record locking was set, and there was a blocking write lock.
[EDEADLK]	A resource deadlock would occur as a result of this operation.
[EINTR]	A system call was interrupted by a signal.
[EINVAL]	The file specified by *path* or *filedes* does not contain a file system of any known type.
[ENOLCK]	The system lock table was full, so the read could not go to sleep until the blocking write lock was removed.

AUTHOR

statfsdev() and **fstatfsdev()** were developed by HP.

FILES

/usr/include/sys/mount.h

SEE ALSO

bdf(1M), df(1M), stat(2), statfs(2).

NAME

statvfsdev, fstatvfsdev - get file system information

SYNOPSIS

```
#include <sys/statvfs.h>
int statvfsdev(const char *path, struct statvfs *buf);
int fstatvfsdev(int fildes, struct statvfs *buf);
```

DESCRIPTION

statvfsdev() returns information about the file system on the device file specified by *path*. The file system need not be mounted.

fstatvfsdev() returns similar information for an open file.

The parameters for the **stat()**, **fstat()**, and **lstat()** functions are as follows:

path is a pointer to the name of the device file. (All directories listed in the path name must be searchable.)

buf is a pointer to a **statvfs()** structure, which is where the file status information is stored.

fildes is a file descriptor for an open file, which is created with the successful completion of an **open()**, **creat()**, **dup()**, **fcntl()**, or **pipe()** system call (see *open*(2), *creat*(2), *dup*(2), *fcntl*(2), or *pipe*(2)).

buf is a pointer to a **statvfs** structure into which information is placed concerning the file system. The contents of the structure pointed to by *buf* are described in *statvfs*(2).

fstatvfsdev() returns the same information as above, but about the open device file referred to by file descriptor *fildes*.

APPLICATION USAGE

statvfsdev() and **fstatvfsdev()** are thread-safe.

RETURN VALUE

Upon successful completion, **statvfsdev()** and **fstatvfsdev()** return zero. Otherwise, they return –1 and set the global variable **errno** to indicate the error.

ERRORS

If **statvfsdev()** fails, **errno** is set to one of the following values:

[EACCES] Search permission is denied for a component of the path prefix.

[EFAULT] *path* points to an invalid address.

[ELOOP] Too many symbolic links are encountered during path-name translation.

[EMFILE] The maximum number of file descriptors allowed are currently open.

[ENAMETOOLONG] The length of the specified path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.

[ENFILE] The system file table is full.

[ENOENT] The named file does not exist.

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

[ENXIO] The device specified by the named special file does not exist.

If **fstatvfsdev()** fails, **errno** is set to one of the following values:

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

[ESPIPE] *filedes* is invalid.

When both **fstatvfsdev()** and **statvfsdev()** fail, **errno** is set to one of the following values:

[EINTR] A system call was interrupted by a signal.

[EINVAL] The file specified by *path* or *filedes* does not contain a file system of any known type.

AUTHOR

statvfsdev() and **fstatvfsdev()** were developed by HP.

SEE ALSO

bdf(1M), df(1M), stat(2), statvfs(2), fgetpos64(3S).

NAME

stdio( ) - standard buffered input/output stream file package

SYNOPSIS

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

DESCRIPTION

The Standard I/O functions described in the subsection (3S) entries of this manual constitute an efficient, user-level I/O buffering scheme. The **getc()** and **putc()** functions handle characters quickly. The following funtions all use or act as if they use **getc()** and **putc()**, and can be freely intermixed:

fgetc()	**fputs()**	**getchar()**	**putchar()**
fgets()	**fread()**	**gets()**	**puts()**
fprintf()	**fscanf()**	**getw()**	**putw()**
fputc()	**fwrite()**	**printf()**	**scanf()**

A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type **FILE**. **fopen()** creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Section (3S) library routines operate on this stream.

At program startup, three streams, **standard input**, **standard output**, and **standard error**, are predefined and do not need to be explicitly opened. When opened, the standard input and standard output streams are fully buffered if the output refers to a file and line-buffered if the output refers to a terminal. The standard error output stream is by default unbuffered. These three streams have the following constant pointers declared in the <**stdio.h**> header file :

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 individual descriptions for details).

An integer constant **BUFSIZ** specifies the size of the buffers used by the particular implementation (see *setbuf*(3S)).

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 subsection (3S) entries of this manual are declared in that header file and need no further declaration.

A constant **_NFILE** defines the default maximum number of open files allowed per process. To increase the open file limit beyond this default value, see *getrlimit*(2).

S

WARNINGS

Use of **stdio** interfaces with a shared read/write file descriptor on **non-positional devices** will provide undefined behavior. Applications which are doing **stdio** operations on **non-positional devices** need to use seperate file pointers for input and output, even if using the same file descriptor for both types of operations.

ERRORS

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

SEE ALSO

close(2), lseek(2), open(2), pipe(2), read(2), getrlimit(2), write(2), ctermid(3S), cuserid(3S), fclose(3S), ferror(3S), fgetpos(3S), fileno(3S), fopen(3S), fread(3S), fseek(3S), fgetpos(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).

STANDARDS CONFORMANCE

stderr: AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

stdin: AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

stdout: AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME
stdscr — default window

SYNOPSIS
```
#include <curses.h>
extern WINDOW *stdscr;
```

DESCRIPTION
The external variable *stdscr* specifies the default window used by functions that do not specify a window using an argument of type **WINDOW *** . Other windows may be created using **newwin()** .

SEE ALSO
derwin(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 4.

NAME

strfmon - convert monetary value to string

SYNOPSIS

```
#include <monetary.h>
ssize_t strfmon(char *s, size_t maxsize, const char *format, ...);
```

Remarks

The ANSI C "`, ...`" construct denotes a variable length argument list whose optional [or required] members are given in the associated comment (`/* */`).

DESCRIPTION

The **`strfmon()`** function places characters into the array pointed to by *s* as controlled by the string pointed to by *format*. No more than *maxsize* bytes are placed into the array.

The format is a character string that contains two types of objects: plain characters, which are simply copied to the output, and conversion specifications, each of which results in the fetching of zero or more arguments that are converted and formatted. The arguments are of type **`double`**, see the *Conversion Characters* section for details. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are ignored.

A conversion specification is the string

%[*flag ...*][*field_width*][**#***left_precision*][**.***right_precision*] *conversion_character*

Each element of the sequence is specified as follows:

Flags

One or more of the following optional flags can be specified to control the conversion:

=*f* An **=** (equal sign) followed by a single character *f* which is used as the numeric fill character. The fill character must be representable in a single byte in order to work with precision and width counts. The default numeric fill character is the space character This flag does not affect field width filling which always uses the space character. This flag is ignored unless a left precision (see below) is specified.

^ Do not format the currency amount with grouping characters The default is to insert the grouping characters if defined for the current locale.

+ or **(** Specify the style for representing positive and negative currency amounts. Only one of **+** or **(** (plus sign or left parenthesis) may be specified If **+** is specified, the locale's equivalent of + and – are used (for example, in the **`en_US.roman8`** locale: an empty string if positive and – if negative) If **(** is specified, negative amounts are enclosed within parentheses If neither flag is specified, the **+** style is used.

! Suppress the currency symbol from the output conversion.

- A minus sign specifying the alignment If this flag is present all fields are left-justified (padded to the right) rather than right-justified.

Field Width

w A decimal digit string *w* specifying a minimum field width in bytes in which the result of the conversion is right-justified (or left-justified if the flag – is specified) The default is zero.

Left Precision

#*n* A **#** followed by a decimal digit string *n* specifying a maximum number of digits expected to be formatted to the left of the radix character This option can be used to keep the formatted output from multiple calls to the **`strfmon()`** aligned in the same columns It can also be used to fill unused positions with a special character as in **`$***123.45`**. This option causes an amount to be formatted as if it has the number of digits specified by *n*. If more than *n* digit positions are required, this conversion specification is ignored. Digit positions in excess of those actually required are filled with numeric fill character (see the **=***f* flag above).

If grouping has not been suppressed with the **^** flag, and it is defined for the current locale, grouping separators are inserted before the fill characters (if any) are added Grouping separators are not applied to fill characters even if the fill character is a digit.

To ensure alignment, any characters appearing before or after the number in the formatted output such as currency or sign symbols are padded as necessary with space characters to make their positive and negative formats an equal length.

Right Precision

.*p* A period followed by a decimal digit string *p* specifying the number of digits after the radix character If the value of the right precision *p* is zero, no radix character appears If a right precision is not included, a default specified by the current locale is used The amount being formatted is rounded to the specified number of digits prior to formatting.

Conversion Characters

The conversion characters and their meanings are:

i The **double** argument is formatted according to the locale's international currency format (for example, in the **en_US.roman8** locale: **USD 1,234.56**).

n The **double** argument is formatted according to the locale's national currency format (for example, in the **en_US.roman8** locale: **$1,234.56**).

% Convert to a **%**; no argument is converted The entire conversion specification must be %%.

EXTERNAL INFLUENCES

Locale

The LC_MONETARY category of the program's locale affects the behavior of this function including the monetary radix character (which may be different from the numeric radix character affected by the LC_NUMERIC category), the grouping separator, the currency symbols and formats.

APPLICATION USAGE

strfmon() is thread-safe. It is not async-cancel-safe.

RETURN VALUE

If the total number of resulting bytes including the terminating null byte is not more than *maxsize*, the **strfmon()** function returns the number of bytes placed into the array pointed to by *s*, not including the terminating null byte Otherwise, –1 is returned, the contents of the array are indeterminate, and *errno* is set to indicate the error.

ERRORS

The **strfmon()** function will fail if:

[E2BIG] Conversion stopped due to lack of space in the buffer.

EXAMPLES

The following program segment formats the monetary value **-4321.123** using the **en_US.roman8** locale with a left precision of **5** and ***** as the fill character.

```
char string[31];
double amt = -4321.123;

setlocale(LC_MONETARY, "en_US.roman8");
strfmon(string, 31, "The amount is %=*#5n.", amt);
```

The *string* array will contain:

```
The amount is -$*4,321.12.
```

S

As an other example, given the locale of **en_US.roman8** and the values **123.45**, **-123.45** and **3456.781**:

Conversion Specification	Output	Comments
%n	$123.45 -$123.45 $3,456.78	default formatting
%11n	$123.45 -$123.45 $3,456.78	right align within an 11 character field
%#5n	$ 123.45 -$ 123.45 $ 3,456.78	align columns for values up to 99,999
%=*#5n	$***123.45 -$***123.45 $*3,456.78	specify a fill character
%=0#5n	$000123.45 -$000123.45 $03,456.78	fill characters do not use grouping even if the fill character is a digit
%^#5n	$ 123.45 -$ 123.45 $ 3456.78	disable the grouping separator
%^#5.0n	$ 123 -$ 123 $ 3457	round off to whole units
%^#5.4n	$ 123.4500 -$ 123.4500 $ 3456.7810	increase the precision
%(#5n	$ 123.45 ($ 123.45) $ 3,456.78	use an alternative positive/negative style
%!(#5n	123.45 (123.45) 3,456.78	disable the currency symbol

AUTHOR

strfmon was developed by HP.

SEE ALSO

localeconv(3C).

STANDARDS COMPLIANCE

strfmon(): XPG4

NAME

strftime() - convert date and time to string

SYNOPSIS

```
#include <time.h>

size_t strftime(
    char *s,
    size_t maxsize,
    const char *format,
    const struct tm *timeptr
);
```

DESCRIPTION

The **strftime()** function converts the contents of a **tm** structure (see *ctime*(3C)) to a formatted date and time string.

strftime() places characters into the array pointed to by *s* as controlled by the string pointed to by *format*. The *format* string consists of zero or more directives and ordinary characters. A directive consists of a **%** character, an optional field width and precision specification, and a terminating character that determines the directive's behavior. All ordinary characters (including the terminating null character are copied unchanged into the array. No more than *maxsize* characters are placed into the array. Each directive is replaced by the appropriate characters as described in the following list. The appropriate characters are determined by the program's locale, by the values contained in the structure pointed to by *timeptr*, and by the **TZ** environment variable (see External Influences below).

Directives

The following directives, shown without the optional field width and precision specification, are replaced by the indicated characters:

%a	Locale's abbreviated weekday name.
%A	Locale's full weekday name.
%b	Locale's abbreviated month name.
%B	Locale's full month name.
%c	Locale's appropriate date and time representation.
%C	The century number (the year divided by 100 and truncated to an integer) as a decimal number [00-99].
%d	Day of the month as a decimal number [01,31].
%D	Equivalent to the directive string **%m/%d/%y**.
%e	Day of the month as a decimal number [1,31]; a single digit is preceded by a space.
%h	Equivalent to **%b**.
%H	Hour (24-hour clock) as a decimal number [00,23].
%I	Hour (12-hour clock) as a decimal number [01,12].
%j	Day of the year as a decimal number [001,366].
%m	Month as a decimal number [01,12].
%M	Minute as a decimal number [00,59].
%n	The New-line character.
%p	Locale's equivalent of either AM or PM.
%r	The time in AM and PM notation; in the POSIX locale this is equivalent to **%I:%M:%S %p**.
%R	The time in 24 hour notation (**%H:%M**).
%S	Second as a decimal number [00,61].
%t	The Tab character.
%T	The time in hours, minutes, and seconds (**%H:%M:%S**).
%u	The weekday as a decimal number [1(Monday),7].
%U	Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.
%V	The week number of the year (Monday as the first day of the week) as a decimal number [01,53]. If the week containing January 1st has four or more days in the new year, then it is considered week 1; otherwise, it is week 53 of the previous year, and the next week is week 1.
%w	Weekday as a decimal number [0(Sunday),6].
%W	Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.

%x Locale's appropriate date representation.
%X Locale's appropriate time representation.
%y Year without century as a decimal number [00,99].
%Y Year with century as a decimal number.
%Z Time zone name (or by no characters if no time zone exists).
%% The percent (%) character.

The following directives are provided for backward compatibility with the directives supported by *date*(1) and the *ctime*(3C) functions. These directives may be removed in a future release. It is recommended that the directives above be used in preference to those below.

%E Locale's combined Emperor/Era name and year (use **%EC%Ey** instead).

%F Locale's full month name (use **%B** instead).

%N Locale's Emperor/Era name (use **%EC** instead).

%o Locale's Emperor/Era year (use **%Ey** instead).

%z Time zone name (or by no characters if no time zone exists) (use **%Z** instead).

If a directive is not one of the above, the behavior is undefined.

Modified Conversion Specifiers

Some conversion specifiers can be modified by the E or O modifer characters to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified conversion specifier. If the alternative format or specification does not exist for the current locale, the behavior will be as if the unmodified conversion specification were used. Alternative numeric symbols refers to those symbols defined by the **ALT_DIGIT** (see *langinfo*(5)) in the locale.

%Ec The locales alternative appropriate date and time representation.

%EC The name of the base year (period/Emperor/Era) in the locale's alternative representation.

%Ex The locale's alternative date representation

%EX The locale's alternative time representation.

%Ey The offset from **%EC** (year only) in the locale's alternative representation.

%EY The full alternative year representation.

%Od The day of the month, using the locale's alternative numeric symbols, filled as needed with leading zeros if there is any alternative symbol for zero, otherwise with leading spaces.

%Oe the day of the month, using the locale's alternative numeric symbols, filled as needed with leading spaces.

%OH The hour (24-hour clock) using the locale's alternative numeric symbols.

%OI The hour (12-hour clock) using the locale's alternative numeric symbols.

%Om The month using the locale's alternative numeric symbols.

%OM The minutes using the locale's alternative numeric symbols.

%OS The seconds using the locale's alternative numeric symbols.

%Ou The weekday as a number in the locale's alternative representation (Monday=1).

%OU The week number of the year (Sunday as the first day of the week, rules corresponding to **%U**) using the locale's alternative numeric symbols.

%OV The week number of the year (Monday as the first day of the week, rules corresponding to **%V**) using tht locale's alternative numeric symbols.

%Ow The number of the weekday (Sunday=0) using the locale's alternative numeric symbols.

%OW The week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.

%Oy The year (offset from **%C**) in the locale's alternative representation and using the locale's alternative symbols.

S

Field Width and Precision

An optional field width and precision specification can immediately follow the initial **%** of a directive in the following order:

[**-**|**0**]*w* The decimal digit string *w* specifies a minimum field width in which the result of the conversion is right- or left-justified. It is right-justified (with space padding) by default. If the optional flag **-** is specified, it is left-justified with space padding on the right. If the optional flag **0** is specified, it is right-justified and padded with zeros on the left.

.*p* The decimal digit string *p* specifies the minimum number of digits to appear for the **d**, **H**, **I**, **j**, **m**, **M**, **o**, **S**, **U**, **w**, **W**, **y** and **Y** directives, and the maximum number of bytes to be used from the **a**, **A**, **b**, **B**, **c**, **D**, **E**, **F**, **h**, **n**, **N**, **p**, **r**, **t**, **T**, **x**, **X**, **z**, **Z**, and **%** directives. In the first case, if a directive supplies fewer digits than specified by the precision, it will be expanded with leading zeros. In the second case, if a directive supplies more bytes than specified by the precision, excess bytes will truncated on the right.

If no field width or precision is specified for a **d**, **H**, **I**, **m**, **M**, **S**, **U**, **W**, **y**, or **j** directive, a default of **.2** is used for all but **j** for which **.3** is used.

EXTERNAL INFLUENCES

Locale

The **LC_TIME** category determines the characters to be substituted for those directives described above as being from the locale.

The **LC_CTYPE** category determines the interpretation of the bytes within *format* as single and/or multi-byte characters.

The **LC_NUMERIC** category determines the characters used to form numbers for those directives that produce numbers in the output. If **ALT_DIGITS** (see *langinfo*(5)) is defined for the locale, the characters so specified are used in place of the default ASCII characters. If both **ALT_DIGITS** and **ALT_DIGIT** is defined for the locale, **ALT_DIGITS** will take precedence over **ALT_DIGIT.**

Environment Variables

TZ determines the time zone name substituted for the **%Z** and **%z** directives. The time zone name is determined by calling the function **tzset()** which sets the external variable **tzname** (see *ctime*(3C)).

International Code Set Support

Single- and multi-byte character code sets are supported.

APPLICATION USAGE

strftime() is thread-safe. It is not async-cancel-safe.

RETURN VALUE

If the total number of resulting bytes including the terminating null byte is not more than *maxsize*, **strftime()** returns the number of bytes placed into the array pointed to by *s*, not including the terminating null byte. Otherwise, zero is returned and the contents of the array are indeterminate.

S

EXAMPLES

If the *timeptr* argument contains the following values:

```
timeptr->tm_sec =    4;
timeptr->tm_min =    9;
timeptr->tm_hour =  15;
timeptr->tm_mday =   4;
timeptr->tm_mon =    6;
timeptr->tm_year =  88;
timeptr->tm_wday =   1;
timeptr->tm_yday = 185;
timeptr->tm_isdst =  1;
```

the following combinations of the **LC_TIME** category and format strings produce the indicated output:

LC_TIME	Format String	Output
en_US.roman8	%x	Mon, Jul 4, 1988
de_De.roman8	%x	Mo., 4. Juli 1988
en_US.roman8	%X	03:09:04 PM

fr_FR.roman8	**%X**	**15h09 04**
*any**	**%H:%M:%S**	**15:09:04**
*any**	**%.1H:%.1M:%.1S**	**15:9:4**
*any**	**%2.1H:%-3M:%03.1S**	**15:9 :004**

* The directives used in these examples are not affected by the **LC_TIME** category of the locale.

WARNINGS

If the arguments *s* and *format* are defined such that they overlap, the behavior is undefined.

The function **tzset()** is called upon every invocation of **strftime()** (whether or not the time zone name is copied to the output array).

The range of values for **%S** ([0,61]) extends to 61 to allow for the occasional one or two leap seconds. However, the system does not accumulate leap seconds and the **tm** structure generated by the functions **localtime()** and **gmtime()** (see *ctime*(3C)) never reflects any leap seconds.

Results are undefined if values contained in the structure pointed to by *timeptr* exceed the ranges defined for the **tm** structure (see *ctime*(3C)) or are not consistent (such as if the **tm_yday** element is set to 0, indicating the first day of January, while the **tm_mon** element is set to 11, indicating a day in December).

AUTHOR

strftime() was developed by HP.

SEE ALSO

date(1), ctime(3C), getdate(3C), setlocale(3C), environ(5), hpnls(5), langinfo(5).

STANDARDS CONFORMANCE

strftime(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

strcat(), strncat(), strcmp(), strncmp(), strcasecmp(), strncasecmp(), strcpy(), strncpy(), strdup(), strlen(), strchr(), strrchr(), strpbrk(), strspn(), strcspn(), strstr(), strrstr(), strtok(), strtok_r(), strcoll(), strxfrm(), index(), rindex() - character string operations

SYNOPSIS

```
#include <string.h>
#include <strings.h>
char *strcat(char *s1, const char *s2);
char *strncat(char *s1, const char *s2, size_t n);
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
int strcasecmp(const char *s1, const char *s2);
int strncasecmp(const char *s1, const char *s2, size_t n);
char *strcpy(char *s1, const char *s2);
char *strncpy(char *s1, const char *s2, size_t n);
char *strdup(const char *s);
size_t strlen(const char *s);
char *strchr(const char *s, int c);
char *strrchr(const char *s, int c);
char *strpbrk(const char *s1, const char *s2);
size_t strspn(const char *s1, const char *s2);
size_t strcspn(const char *s1, const char *s2);
char *strstr(const char *s1, const char *s2);
char *strrstr(const char *s1, const char *s2);
char *strtok(char *s1, const char *s2);
char *strtok_r(char *s1, const char *s2, char **last);
int strcoll(const char *s1, const char *s2);
size_t strxfrm(char *s1, const char *s2, size_t n);
char *index(const char *s, int c);
char *rindex(const char *s, int c);
```

S

Remarks:

All functions except **index()** and **rindex()** are declared in both headers, so only one of the two headers needs to be included.

The functions **index()** and **rindex()** are declared only in <**strings.h**>, and <**strings.h**>. They are provided solely for portability of BSD applications, and are not recommended for new applications where portability is important. For portable applications, use <**string.h**>, **strchr()**, and **strrchr()** instead.

DESCRIPTION

Arguments *s1*, *s2*, and *s* point to strings (arrays of characters terminated by a null byte).

Definitions for all these functions, the type **size_t**, and the constant **NULL** are provided in the <**string.h**> header.

strcat() Appends a copy of string *s2* to the end of string *s1*. **strncat()** appends a maximum of *n* characters. It copies fewer if *s2* is shorter than *n* characters. Each returns a pointer to the null-terminated result (the value of *s1*).

strcmp() Compares its arguments and returns an integer less than, equal to, or greater than zero, depending on whether *s1* is lexicographically less than, equal to, or greater than *s2*. The

comparison of corresponding characters is done as if the type of the characters were **unsigned char**. Null pointer values for *s1* and *s2* are treated the same as pointers to empty strings. **strncmp()** makes the same comparison but examines a maximum of *n* characters (*n* less than or equal to zero yields equality). **strcasecmp()** and **strncasecmp()** are identical in function to **strcmp()** and **strncmp()** respectively, but characters are folded by **_tolower()** (see *conv*(3C)) prior to comparison. The returned lexicographic difference reflects the folding to lowercase.

strcpy() Copies string *s2* to *s1*, stopping after the null byte has been copied. **strncpy()** copies exactly *n* characters, truncating *s2* or adding null bytes to *s1* if necessary, until a total of *n* have been written. The result is not null-terminated if the length of *s2* is *n* or more. Each function returns *s1*. Note that **strncpy()** should not be used to copy *n* bytes of an arbitrary structure. If that structure contains a null byte anywhere, **strncpy()** copies fewer than *n* bytes from the source to the destination and fills the remainder with null bytes. Use the **memcpy()** function (see *memory*(3C)) to copy arbitrary binary data.

strdup() Returns a pointer to a new string which is a duplicate of the string to which *s1* points. The space for the new string is obtained using the **malloc()** function (see *malloc*(3C)).

strlen() Returns the number of characters in *s*, not including the terminating null byte.

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 byte terminating a string is considered to be part of the string. **index()** (**rindex()**) is identical to **strchr()** (**strrchr()**), and is provided solely for portability of BSD applications.

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 maximum initial segment of string *s1*, which consists entirely of characters from (not from) string *s2*.

strstr() (**strrstr()**) Returns a pointer to the first (last) occurrence of string *s2* in string *s1*, or a NULL pointer if *s2* does not occur in the string. If *s2* points to a string of zero length, **strstr()** (**strrstr()**) returns *s1*.

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 a nonnull pointer *s1* specified) returns a pointer to the first character of the first token, and writes a null byte into *s1* immediately following the returned token. The function keeps track of its position in the string *s1* between separate calls, so that subsequent calls made with the first argument a null pointer work through the string immediately following that token. In this way subsequent calls work through the string *s1* until no tokens remain. The separator string *s2* can be different from call to call. When no token remains in *s1*, a null pointer is returned.

S **strtok_r()** is identical to **strtok()**, except that it expects to be passed the address of a character string pointer as the third argument. It will use this argument to keep track of the current position in the string being searched. It returns a pointer to the current token in the string or a NULL value if there are no more tokens.

strcoll() Returns an integer greater than, equal to, or less than zero, according to whether the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2*. The comparison is based on strings interpreted as appropriate to the program's locale (see Locale below). In the "C" locale **strcoll()** works like **strcmp()**.

strxfrm() Transforms the string pointed to by *s2* and places the resulting string into the array pointed to by *s1*. The transformation is such that if the **strcmp()** function is applied to two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the **strcoll()** function applied to the same two original strings. No more than *n* bytes are placed into the resulting string, including the terminating null character. If the transformed string fits in no more than *n* bytes, the length of the resulting string is returned (not including the terminating null character). Otherwise the return value is the number of bytes that the *s1* string would occupy (not including the terminating null character), and the contents of the array are indeterminate.

strcoll() has better performance with respect to **strxfrm()** in cases where a given string is compared to other strings only a few times, or where the strings to be compared are long but a difference in the

strings that determines their relative ordering usually comes among the first few characters. **strxfrm()** offers better performance in, for example, a sorting routine where a number of strings are each transformed just once and the transformed versions are compared against each other many times.

APPLICATION USAGE

The interface **strtok()** is not thread-safe. All other interfaces documented on this man page are thread-safe and async-cancel-safe.

EXTERNAL INFLUENCES

Locale

The **LC_CTYPE** category determines the interpretation of the bytes within the string arguments to the **strcoll()** and **strxfrm()** functions as single and/or multibyte characters. It also determines the case conversions to be done for the **strcasecmp()** and **strncasecmp()** functions.

The **LC_COLLATE** category determines the collation ordering used by the **strcoll()** and **strxfrm()** functions. See *hpnls*(5) for a description of supported collation features.

International Code Set Support

Single- and multibyte character code sets are supported for the **strcoll()** and **strxfrm()** functions. All other functions support only single-byte character code sets.

EXAMPLE

The following sample piece of code finds the tokens, separated by blanks, that are in the string *s* (assuming that there are at most **MAXTOK** tokens):

```
int i = 0;
char *s, *last, *tok[MAXTOK];

tok[0] = strtok_r(s, " ", &last);
while (tok[++i] = strtok_r(NULL, " ", &last));
```

WARNINGS

The functions **strcat()**, **strncat()**, **strcpy()**, **strncpy()**, **strtok()**, and **strtok_r()** alter the contents of the array to which *s1* points. They do not check for overflow of the array.

Null pointers for destination strings cause undefined behavior.

Character movement is performed differently in different implementations, so moves involving overlapping source and destination strings may yield surprises.

The transformed string produced by **strxfrm()** for a language using an 8-bit code set is usually at least twice as large as the original string and may be as much four times as large (ordinary characters occupy two bytes each in the transformed string, 1-to-2 characters four bytes, 2-to-1 characters two bytes per original pair, and don't-care characters no bytes). Each character of a multibyte code set (Asian languages) occupies three bytes in the transformed string.

For functions **strcoll()** and **strxfrm()** results are undefined if the languages specified by the **LC_COLLATE** and **LC_CTYPE** categories use different code sets.

strtok() is unsafe for multi-thread applications. **strtok_r()** is MT-Safe and should be used instead.

Users of **strtok_r()** should also note that the prototype of this function will change in the next release for conformance with the new POSIX Threads standard.

AUTHOR

string() was developed by the University of California, Berkeley, AT&T, OSF, and HP.

SEE ALSO

conv(3C), malloc(3C), memory(3C), setlocale(3C), hpnls(5).

STANDARDS CONFORMANCE

nl_strcmp(): XPG2

nl_strncmp(): XPG2

strcat(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

strchr(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
strcmp(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
strcoll(): AES, SVID3, XPG3, XPG4, ANSI C
strcpy(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
strcspn(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
strdup(): SVID2, SVID3
strlen(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
strncat(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
strncmp(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
strncpy(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
strpbrk(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
strrchr(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
strspn(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
strstr(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
strtok(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
strxfrm(): AES, SVID3, XPG3, XPG4, ANSI C

NAME

strord - convert string data order

SYNOPSIS

```
#include <nl_types.h>
char *strord(char *s1, const char *s2, nl_mode m);
```

DESCRIPTION

The text orientation (mode) of a file can be right-to-left (non-Latin) or left-to-right (Latin). This text orientation can affect the way data is arranged in the file. The data arrangements that result are called screen order and keyboard order (see *hpnls*(5) for more details).

strord() converts the order of characters in *s2* from screen to keyboard order or vice versa and places the result in *s1*. The arguments *s1* and *s2* point to strings (arrays of characters terminated by a null character). **strord()** returns *s1*.

strord() performs the conversion based on mode information indicated by the argument *m*. The argument *m* is of type *nl_mode* found in the header file <**nl_types.h**>. The mode argument can have two possible values: **NL_LATIN** and **NL_NONLATIN**.

If the mode argument is **NL_LATIN**, the text orientation is left-to-right, and all non-Latin sub-strings are reversed. Non-Latin sub-strings are any number of contiguous right-to-left language characters. Non-Latin sub-strings are delimited by ASCII characters.

Similarly, if the mode argument is **NL_NONLATIN**, the text orientation is right-to-left and all Latin sub-strings are reversed. Latin sub-strings are any number of contiguous printable ASCII characters. Latin sub-strings are delimited by right-to-left language characters and ASCII control codes.

Some right-to-left languages have a duplicate set of digits called alternative numbers. Alternative numbers always have a left-to-right orientation.

EXTERNAL INFLUENCES

Locale

The **LC_NUMERIC** category determines whether a right-to-left language has alternative numbers.

International Code Set Support

Single-byte character code sets are supported.

APPLICATION USAGE

strord() is thread-safe and async-cancel-safe.

WARNINGS

strord() does not check for overflow of the array pointed to by *s1*.

AUTHOR

strord() was developed by HP.

SEE ALSO

setlocale(3C), hpnls(5), environ(5), forder(1), nljust(1).

NAME

strptime - date and time conversion

SYNOPSIS

```
#include <time.h>

char *strptime(const char *buf, const char *format, struct tm *tm);
```

DESCRIPTION

The **strptime()** function converts the character string pointed to by *buf* to values which are stored in the **tm** structure pointed to by *tm*, using the format specified by *format*.

The *format* is composed of zero or more directives. Each directive is composed of one of the following: one or more white-space characters (as specified by the **isspace()** function); an ordinary character (neither **%** nor a white-space character); or a conversion specification. Each conversion specification is composed of a **%** character followed by a conversion character which specifies the replacement required. There must be white-space or other non-alphanumeric characters between any two conversion specifications. The following conversion specifications are supported:

%a is the day of week, using the locale's weekday names; either the abbreviated or full name may be specified.

%A is the same as **%a**.

%b is the month, using the locale's month names; either the abbreviated or full name may be specified.

%B is the same as **%b**.

%c is the date and time, using locale's date and time format (for example, as **%x %X**).

%C is the century number [0,99]; leading zeros are permitted but not required.

%d is the day of month [1,31]; leading zeros are permitted but not required.

%D is the date as **%m/%d/%y**.

%e is the same as **%d**.

%h is the same as **%b**.

%H is the hour (24-hour clock) [0,23]; leading zeros are permitted but not required.

%I is the hour (12-hour clock) [1,12]; leading zeros are permitted but not required.

%j is the day number of the year [1,366]; leading zeros are permitted but not required.

%m is the month number [1,12]; leading zeros are permitted but not required.

%M is the minute [0,59]; leading zeros are permitted but not required.

S

%n is any white-space.

%p is the locale's equivalent of a.m or p.m..

%r is the time as **%I:%M:%S %p**.

%R is the time as **%H:%M**.

%S is the seconds [0,61]; leading zeros are permitted but not required.

%t is any white-space.

%T is the time as **%H:%M:%S**.

%U is the week number of the year (Sunday as the first day of the week) as a decimal number [0,53]; leading zeros are permitted but not required. All days in a year preceding the first Sunday are considered to be in week 0.

%w is the weekday as a decimal number [0,6], with 0 representing Sunday; leading zeros are permitted but not required.

%W is the week number of the year (Monday as the first day of the week) as a decimal number [0,53]; leading zeros are permitted but not required. All days in a year preceding the first Monday are considered to be in week 0.

%x is the date, using the locale's date format.

%X is the time, using the locale's time format.

%y is the year within the century [0,99]; leading zeros are permitted but not required. If no century has been specified (for example, via the **%C** directive), the 20th century (1900s) is assumed for inputs in the range 69-99, and the 21st century (2000s) is assumed for inputs in the range 00-68.

%Y is the year, including the century (for example, 1992).

%% is replaced by **%**.

Modified Directives

Some directives can be modified by the E and O modifier characters to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified directive. If the alternative format or specification does not exist in the current locale, the behavior will be as if the unmodified directive were used.

%Ec is the locale's alternative appropriate date and time representation.

%EC is the name of the base year (period) in the locale's alternative representation.

%Ex is the locale's alternative date representation.

%EX is the locale's alternative time representation.

%Ey is the offset from **%EC** (year only) in the locale's alternative representation.

%EY is the full alternative year representation.

%Od is the day of the month using the locale's alternative numeric symbols; leading zeros are permitted by not required.

%Oe is the same as **%Od**.

%OH is the hour (24-hour clock) using the locale's alternative numeric symbols.

%OI is the hour (12-hour clock) using the locale's alternative numeric symbols.

%Om is the month using the locale's alternative numeric symbols.

%OM is the minutes using the locale's alternative numeric symbols.

%OS is the seconds using the locale's alternative numeric symbols.

%OU is the week number of the year (Sunday as the first day of the week) using the locale's' alternative numeric symbols.

%Ow is the number of the weekday (Sunday=0) using the locale's alternative numeric symbols.

%OW is the week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.

%Oy is the year (offset from **%C**) in the locale's alternative representation and using the locale's alternative numeric symbols.

A directive composed of white-space characters is executed by scanning input up to the first character that is not white-space (which remains unscanned), or until no more characters can be scanned.

A directive that is an ordinary character is executed by scanning the next character from the buffer. If the character scanned from the buffer differs from the one comprising the directive, the directive fails, and the differing and subsequent characters remain unscanned.

A series of directives composed of **%n**, **%t**, white-space characters or any combination thereof is executed by scanning up to the first character that is not white space (which remains unscanned), or until no more characters can be scanned.

Any other conversion specification is executed by scanning characters until a character matching the next directive is scanned, or until no more characters can be scanned. These characters, except the one matching the next directive, are then compared to the locale values associated with the conversion specifier. If a match is found, values for the appropriate **tm** structure members are set to values corresponding to the locale information. Case is ignored when matching items in *buf* such as month or weekday names. If no match is found, **strptime()** fails and no more characters are scanned. If the date specified exceeds the maximum time representable by the **time_t** data type in 32-bit HP-UX (which represents Tuesday

S

January 19 03:14:07 UTC, 2038) or if the date exceeds the maximum date supported in 64-bit HP-UX (which is Friday December 31 23:59:59 UTC, 9999), **strptime()** fails and a null pointer is returned.

EXTERNAL INFLUENCES

Locale

The **LC_NUMERIC** category may define the alternative symbols (*alt_digit*—see *localedef*(4)) used by the **%O** modifier. The *alt_digit* definition has precedence over *alt_digits* (LC_TIME). Support for *alt_digit* may be removed in a future release of HP-UX.

The **LC_TIME** category determines the characters to be interpreted for those directives described above as being from the locale.

The **LC_CTYPE** category determines the interpretation of the bytes within *format* as single and/or multi-byte characters.

International Code Set Support

Single- and multi-byte character code sets are supported.

APPLICATION USAGE

strptime() is thread-safe. It is not async-cancel-safe.

RETURN VALUE

Upon successful completion, **strptime()** returns a pointer to the character following the last character parsed. Otherwise, a null pointer is returned.

EXAMPLES

The following program segment uses **strptime()** to convert the string (first argument) to values according to the format specified in the second argument.

```
struct tm t;

setlocale(LC_TIME, "en_US.iso88591");
strptime("1:04:23 PM on 10/6/92", "%I:%M:%S %p on %D", &t);
```

The converted value is stored in the structure t as follows:

```
t.tm_sec   = 23
t.tm_min   = 4
t.tm_hour  = 13
t.tm_mday  = 6
t.tm_mon   = 9
t.tm_year  = 92
t.tm_wday  = 2
t.tm_yday  = 279
t.tm_isdst = 1
```

S

AUTHOR

strptime() was developed by OSF and HP.

SEE ALSO

scanf(3S), strftime(3C), getdate(3C), ctime(3C), setlocale(3C).

STANDARDS COMPLIANCE

strptime: XPG4

NAME

strtoacl(), strtoaclpatt(), aclentrystart[] - convert string form to access control list (ACL) structure (HFS File System only)

SYNOPSIS

```
#include <acllib.h>

int strtoacl(
     const char *string,
     int nentries,
     int maxentries,
     struct acl_entry *acl,
     uid_t fuid,
     gid_t fgid);

int strtoaclpatt(
     const char *string,
     int maxentries,
     struct acl_entry_patt *acl);

int strtoacl_r(
     const char *string,
     int nentries,
     int maxentries,
     struct acl_entry *acl,
     uid_t fuid,
     gid_t fgid,
     char *entrystart[]);

int strtoaclpatt_r(
     const char *string,
     int maxentries,
     struct acl_entry_patt *acl
     char *entrystart[]);
```

DESCRIPTION

strtoacl() converts an access control list from exact symbolic (string) representation to structure form. It parses the input string and verifies its validity. Optionally it applies the entries in the string as a series of changes to an existing ACL.

strtoaclpatt() converts an access control list pattern from symbolic (string) representation to structure form. It parses the input string and verifies its validity.

The external array **aclentrystart[]**, only valid until the next call of either routine, is useful for error reporting. See ERRORS below.

The "operator" and "short" symbolic forms of ACLs and ACL patterns (described in *acl*(5)) are acceptable as input strings. If the first non-whitespace character in *string* is **(**, the ACL or ACL pattern in *string* must be in short form. Otherwise operator form is assumed.

strtoacl() takes a pointer to the string to be converted, and a pointer to the first element of an array of ACL entries (**acl[]**) initially containing the indicated number (*nentries*) of valid entries (zero or more). This array can grow to the indicated number of entries (*maxentries*). **strtoacl()** also takes file user ID (*fuid*) and group ID (*fgid*) values to substitute for **@** characters in *string* and returns the resulting number of entries in **acl[]**.

Redundant entries (identical user ID and group ID values after processing **@** characters) are combined, so that **acl[]** contains unique entries in the order encountered. If a new entry is mentioned, it is added to the end of the **acl** array.

strtoaclpatt()

strtoaclpatt() differs from **strtoacl()** because it processes an ACL pattern instead of an ACL. Since modification of an existing initial ACL is not useful, it is not supported.

Entries with matching user and group ID values are not combined. Each entry input yields one entry in the returned array.

S

The **@** character for user and group IDs (see *acl*(5)) is converted to special values (**ACL_FILEOWNER** or **ACL_FILEGROUP**, respectively, defined in <**acllib.h**>), not to specific user or group names provided by the caller. Thus, **strtoaclpatt()** need not be called to reparse the ACL pattern for each file, but the caller must handle the special values when comparing an ACL pattern to an ACL.

Wildcard user names, group names, and mode values are supported, as are absent mode parts; see *acl*(5).

strtoaclpatt() returns a different structure than **strtoacl()**. The *acl_entry_patt* structure contains *onmode* and *offmode* masks rather than a single *mode* value.

In operator form input, operators have a different effect on **strtoaclpatt()**:

- **=** Sets bits in both the *onmode* and *offmode* fields appropriately, replacing existing bits in the entry, including any set by earlier operators.
- **+** Sets bits in *onmode* and clears the same bits in *offmode*.
- **-** Sets bits in *offmode* and clears the same bits in *onmode*.

In short form input, the mode is treated like the **=** operator in operator form.

For both routines, a non-specific user or group ID of **%** is converted to **ACL_NSUSER** or **ACL_NSGROUP**, respectively. For **strtoaclpatt()** only, a wildcard user or group ID of ***** is converted to **ACL_ANYUSER** or **ACL_ANYGROUP**, respectively. The values are defined in <**acllib.h**>.

Entries can appear in *string* in any order. *string* can contain redundant entries, and in operator form only, redundant **+** and **-** operators for ACL entry mode modifications (in exact form) or mode bit inclusions/exclusions (in patterns). Entries or modifications are applied left to right.

Suggested Use

To build a new ACL (ACL pattern) array using **strtoacl()** (**strtoaclpatt()**), define **acl[]** with as many entries as desired. Pass it to **strtoacl()** (**strtoaclpatt()**) with *nentries* set to zero (**strtoacl()** only) and *maxentries* set to the number of elements in **acl[]**.

To have **strtoacl()** modify a file's existing ACL, define **acl[]** with the maximum possible number of entries (**NACLENTRIES**; see <**sys/acl.h**>). Call **getacl()** (see *getacl*(2)) to read the file's ACL and **stat()** (see *stat*(2)) to get the file's owner and group IDs. Then pass the current number of entries, the current ACL, and the ID values to **strtoacl()** with *maxentries* set to **NACLENTRIES**.

If **strtoacl()** succeeds, the resulting ACL can be passed safely to **setacl()** (see *setacl*(2)) because all redundancies (if any) have been resolved. However, note that since neither **strtoacl()** nor **strtoaclpatt()** validate user and group ID values, if the values are not acceptable to the system, **setacl()** fails.

Performance Trick

Normally **strtoacl()** replaces user and group names of **@** with specific user and group ID values, and also combines redundant entries. Therefore, calling **stat()** and **strtoacl()** for each of a series of files to which an ACL is being applied is simplest, although time consuming.

If *string* contains no **@** character, or if the caller merely wants to compare one ACL against another (and will handle the special case itself), it is sufficient to call **strtoacl()** once, and pointless to call **stat()** for each file. To determine this, call **strtoacl()** the first time with *fuid* set to **ACL_FILEOWNER** and *fgid* set to **ACL_FILEGROUP**. Repeated calls with file-specific *fuid* and *fgid* values are needed only if the special values of *fuid* and *fgid* appear in **acl[]** and the caller needs an exact ACL to set on each file; see EXAMPLES below.

If **@** appears in *string* and **acl[]** will be used later for a call to **setacl()**, it is necessary to call **strtoacl()** again to reparse the ACL string for each file. It is possible that not all redundant entries were combined the first time because the **@** names were not resolved to specific IDs. This also complicates comparisons between two ACLs. Furthermore, the caller cannot do the combining later because operator information from operator form input might be lost.

strtoacl_r() and **strtoaclpatt_r()** convert string form to access control list (ACL) structure.

APPLICATION USAGE

strtoacl() and **strtoaclpatt()** are thread-safe. These interfaces are not async-cancel-safe.

RETURN VALUE

If **strtoacl()** (**strtoaclpatt()**) succeeds, it returns the number of entries in the resulting ACL (ACL pattern), always equal to or greater than *nentries* (zero).

strtoaclpatt() also sets values in global array **aclentrystart[]** to point to the start of each pattern entry it parsed in *string*, in some cases including leading or trailing whitespace. It only sets a number of pointers equal to its return value plus one (never more than **NACLENTRIES** + 1). The last valid element points to the null character at the end of *string*. After calling **strtoaclpatt()**, an entry pattern's corresponding input string can be used by the caller for error reporting by (temporarily) putting a null at the start of the next entry pattern in *string*.

ERRORS

If an error occurs, **strtoacl()** and **strtoaclpatt()** return a negative value and the content of **acl** is undefined (was probably altered). To help with error reporting in this case, **aclentrystart[0]** and **aclentrystart[1]** are set to point to the start of the current and next entries, respectively, being parsed when the error occurred. If the current entry does not start with **(**, **aclentrystart[1]** points to the next null character or comma at or after **aclentrystart[0]**. Otherwise, it points to the next null, or to the character following the next **)**.

The following values are returned in case of error:

–1 Syntax error: entry doesn't start with **(** as expected in short form.

–2 Syntax error: entry doesn't end with **)** as expected in short form.

–3 Syntax error: user name is not terminated by a dot.

–4 (**strtoacl()** only) Syntax error: group name is not terminated by an operator in operator-form input or a comma in short-form input.

–5 Syntax error: user name is null.

–6 Syntax error: group name is null.

–7 Invalid user name (not found in **/etc/passwd** file and not a valid number).

–8 Invalid group name (not found in **/etc/group** file and not a valid number).

–9 Syntax error: invalid mode character, other than **0**..**7**, **r**, **w**, **x**, **-** (allowed in short form only), ***** (allowed in patterns only), **,** (to end an entry in operator form), or **)** (to end an entry in short form). Or, **0**..**7** or ***** is followed by other mode characters.

–10 The resulting ACL would have more than *maxentries* entries.

EXAMPLES

The following code fragment converts an ACL from a string to an array of entries using an *fuid* of 103 for the file's owner and *fgid* of 45 for the file's group.

```
#include <acllib.h>

int nentries;
struct acl_entry acl [NACLENTRIES];

if ((nentries = strtoacl (string, 0, NACLENTRIES, acl, 103, 45)) < 0)
     error (...);
```

The following code gets the ACL, *fuid*, and *fgid* for file **../myfile**, modifies the ACL using a description string, and changes the ACL on file **../myfile2** to be the new version.

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

struct stat statbuf;
int nentries;
struct acl_entry acl [NACLENTRIES];

if (stat ("../myfile", & statbuf) < 0)
     error (...);

if ((nentries = getacl ("../myfile", NACLENTRIES, acl)) < 0)
     error (...);
```

```
if ((nentries = strtoacl (string, nentries, NACLENTRIES, acl,
    statbuf.st_uid, statbuf.st_gid)) < 0)
{
    error (...);
}
if (setacl ("../myfile2", nentries, acl) < 0)
     error (...);
```

The following code fragment calls **strtoacl()** with special values of *fuid* and *fgid*, then checks to see if they show up in **acl[]**.

```
#include <acllib.h>

int  perfile = 0;  /* need to stat() and reparse per file? */
int  entry;

if ((nentries = strtoacl (string, 0, NACLENTRIES, acl,
     ACL_FILEOWNER, ACL_FILEGROUP)) < 0)
{
     error (...);
}
for (entry = 0; entry < nentries; entry++)
{
     if ((acl[entry].uid == ACL_FILEOWNER)
      || (acl[entry].gid == ACL_FILEGROUP))
     {
             perfile = 1;
             break;
     }
}
```

The following code fragment converts an ACL pattern from a string to an array of pattern entries.

```
#include <acllib.h>

int nentries;
struct acl_entry_patt acl [NACLENTRIES];

if ((nentries = strtoaclpatt (string, NACLENTRIES, acl)) < 0)
     error (...);
```

The following code fragment inside a **for** loop checks an entry pattern (**p***, **onmask**, and **offmask** variable names) against an entry in a file's ACL (**a*** variable names) using the file's user and group IDs (**f*** variable names).

S

```
include <unistd.h>
if (((puid == ACL_FILEOWNER) && (fuid != auid))
 || ((puid != ACL_ANYUSER)   && (puid != auid)))
{
    continue;
}
if (((pgid == ACL_FILEGROUP) && (fgid != agid))
 || ((pgid != ACL_ANYGROUP)  && (pgid != agid)))
{
    continue;
}
if ((( amode) & MODEMASK & onmask ) != onmask)
 || ((~ amode) & MODEMASK & offmask) != offmask))
{
    continue;
}
```

DEPENDENCIES

strtoacl() and **strtoaclpatt()** are only supported on HFS file system on standalone HP-UX operating system.

AUTHOR

strtoacl() and **strtoaclpatt()** were developed by HP.

FILES

/etc/passwd
/etc/group

SEE ALSO

getacl(2), setacl(2), acltostr(3C), cpacl(3C), chownacl(3C), setaclentry(3C), acl(5).

S

NAME

strtod, atof - convert string to double-precision number

SYNOPSIS

```
#include <stdlib.h>
double strtod(const char *str, char **ptr);
double atof(const 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 (leading white-space characters as defined by **isspace()** in *ctype*(3C) are ignored) up to the first unrecognized character. If no conversion can take place, zero is returned.

strtod() recognizes characters in the following sequence:

1. An optional string of "white-space" characters which are ignored,
2. An optional sign,
3. A string of digits optionally containing a radix character,
4. An optional **e** or **E** followed by an optional sign or space, followed by an integer.

The radix character is determined by the loaded NLS environment (see *setlocale*(3C)). If **setlocale()** has not been called successfully, the default NLS environment, "C", is used (see *lang*(5)). The default environment specifies a period (**.**) as the radix character.

If the value of *ptr* is not **(char **)NULL**, the variable to which it points is set to point at the character after the last number, if any, that was recognized. If no number can be formed, ***ptr** is set to *str*, and zero is returned.

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

EXTERNAL INFLUENCES

Locale

The **LC_NUMERIC** category determines the value of the radix character within the currently loaded NLS environment.

APPLICATION USAGE

strtod() and **atof()** are thread-safe and async-cancel-safe.

RETURN VALUE

If the correct value would cause overflow, **+HUGE_VAL** or **-HUGE_VAL** 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.

When the input parameter character string for **strtod()** and **atof()** is either **inf** (case insensitive) or **infinity** (case insensitive) **strtod()** and **atof()** will return **HUGE_VAL**.

When the input parameter character string for **strtod()** and **atof()** is **nan** (case insensitive), **strtod()** and **atof()** will return **_DNANQ**.

AUTHOR

strtod() was developed by AT&T and HP.

SEE ALSO

ctype(3C), setlocale(3C), scanf(3S), strtol(3C), hpnls(5), lang(5).

STANDARDS CONFORMANCE

strtod(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, ANSI C

atof(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

strtol(), atol(), atoi(), strtoul() - convert string to integer

SYNOPSIS

```
#include <stdlib.h>
long strtol(const char *str, char **ptr, int base);
long atol(const char *str);
int atoi(const char *str);
unsigned long strtoul(const char *str, char **ptr, int base);
```

DESCRIPTION

strtol() (**strtoul()**) converts the character string pointed to by *str* to **long int** (**unsigned long int**) representation. 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 no conversion can take place, zero is returned.

If *base* is greater than or equal to 2 and less than or equal to 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 as follows: After an optional leading sign, a leading zero indicates octal conversion; a leading **0x** or **0X** hexadecimal conversion. Otherwise, decimal conversion is used.

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, the location pointed to by *ptr* is set to *str*, and zero is returned.

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

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

APPLICATION USAGE

strtol(), **strtoul()**, **atol()** and **atoi()** are thread-safe and async-cancel-safe.

RETURN VALUE

Upon successful completion, all functions return the converted value, if any. If the correct value would cause overflow, **strtol()** returns **LONG_MAX** or **LONG_MIN** (according to the sign of the value), and sets **errno** to ERANGE; **strtoul()** returns **ULONG_MAX** and sets **errno** to ERANGE. Overflow conditions are ignored by **atol()** and **atoi()**.

For all other errors, zero is returned and **errno** is set to indicate the error.

ERRORS

strtol() and **strtoul()** fail and **errno** is set if any of the following conditions are encountered:

[EINVAL] The value of *base* is not supported.

[ERANGE] The value to be returned would have caused overflow.

AUTHOR

strtol(), **atoi()**, and **atol()**, were developed by OSF and HP.

SEE ALSO

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

STANDARDS CONFORMANCE

strtol(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, ANSI C

atoi(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

atol(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

strtoul(): AES, SVID3, XPG4, ANSI C

NAME

strtold() - convert string to long double-precision number

SYNOPSIS

```
#include <stdlib.h>
long double strtold(const char *str, char **ptr);
```

DESCRIPTION

strtold() returns as a long double-precision number the value represented by the character string pointed to by *str*. The string is scanned up to the first unrecognized character.

strtold() 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 radix character, then an optional **e** or **E** followed by an optional sign or space, followed by an integer. The radix character is determined by the loaded NLS environment (see *setlocale*(3C)). If **setlocale()** has not been called successfully, the default NLS environment, "C" (see *lang*(5)), is used. The default environment specifies a period (**.**) as the radix character.

If the value of *ptr* is not **(char **) NULL**, the variable to which it points is set to point at the character after the last number, if any, that was recognized. If no number can be formed, ***ptr** is set to *str*, and zero is returned.

APPLICATION USAGE

strtold() is thread-safe and async-cancel-safe.

EXTERNAL INFLUENCES

International Code Set Support

Single-byte character code sets are supported.

RETURN VALUE

If the correct value would cause overflow, positive **_LDBL_MAX** or negative **_LDBL_NMAX** is returned (according to the sign of the value), and **errno** is set to ERANGE. The **__LDBL_MAX** and **__LDBL_NMAX** macros are provided to assist in checking the return values against **_LDBL_MAX** and **_LDBL_NMAX**.

If the correct value would cause underflow, **_ZERO** is returned and **errno** is set to ERANGE.

AUTHOR

strtold() was developed by HP.

SEE ALSO

ctype(3C), setlocale(3C), scanf(3S), hpnls(5), lang(5).

S

(ENHANCED CURSES)

NAME

subpad — enhanced pad management function

SYNOPSIS

```
#include <curses.h>

WINDOW *subpad(WINDOW *orig, int nlines, int ncols, int begin_y,
               int begin_x);
```

DESCRIPTION

The **subpad()** function creates a subwindow within a pad with *nlines* lines and *ncols* columns. Unlike **subwin()**, which uses screen coordinates, the window is at position (*begin_y*, *begin_x*) on the pad. The window is made in the middle of the window *orig*, so that changes made to one window affect both windows.

RETURN VALUE

subpad() function returns a pointer to the pad data structure. Otherwise, it returns a null pointer.

ERRORS

No errors are defined.

APPLICATION USAGE

To refresh a pad, call **prefresh()** or **pnoutrefresh()**, not **wrefresh()**. When porting code to use pads from WINDOWS, remember that these functions require additional arguments to specify the part of the pad to be displayed and the location on the screen to be used for the display.

Although a subwindow and its parent pad may share memory representing characters in the pad, they need not share status information about what has changed in the pad. Therefore, after modifying a subwindow within a pad, it may be necessary to call **touchwin()** or **touchline()** on the pad before calling **prefresh()**.

SEE ALSO

derwin(3X), doupdate(3X), is_linetouched(3X), newpad(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

S

NAME

swab() - swap bytes

SYNOPSIS

```
#include <unistd.h>

void swab(const void *from, void *to, ssize_t 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 byte-swapped and non-byte-swapped 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.

APPLICATION USAGE

swab() is thread-safe and async-cancel-safe.

STANDARDS CONFORMANCE

swab(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

S

(ENHANCED CURSES)

NAME

syncok, wcursyncup, wsyncdown, wsyncup — synchronise a window with its parents or children

SYNOPSIS

```
#include <curses.h>
int syncok(WINDOW *win, bool bf);
void wcursyncup(WINDOW *win);
void wsyncdown(WINDOW *win);
void wsyncup(WINDOW *win);
```

DESCRIPTION

The **syncok()** function determines whether all ancestors of the specified window are implicitly touched whenever there is a change in the window. If *bf* is TRUE, such implicit touching occurs. If *bf* is FALSE, such implicit touching does not occur. The initial state is FALSE.

The **wcursyncup()** function updates the current cursor position of the ancestors of *win* to reflect the current cursor position of *win*.

The **wsyncdown()** function touches *win* if any ancestor window has been touched.

The **wsyncup()** function unconditionally touches all ancestors of *win*.

RETURN VALUE

Upon successful completion, **syncok()** returns OK. Otherwise, it returns ERR.

The other functions do not return a value.

ERRORS

No errors are defined.

APPLICATION USAGE

Applications seldom call **wsyncdown()** because it is called by all refresh operations.

SEE ALSO

doupdate(3X), is_linetouched(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

S

NAME

syslog(), openlog(), closelog(), setlogmask() - control system log

SYNOPSIS

```
#include <syslog.h>
void syslog(int priority, const char *message, ...);
void openlog(const char *ident, int logopt, int facility);
void closelog(void);
int setlogmask(int maskpri);
```

Remarks

The ANSI C "`, ...`" construct denotes a variable length argument list whose optional [or required] members are given in the associated comment (`/* */`).

DESCRIPTION

`syslog()` writes a message onto the system log maintained by **`syslogd`** (see *syslogd*(1M)). The message is tagged with *priority*. The *message* is similar to a *printf*(3S) format string except that **`%m`** is replaced by the error message associated with the current value of **`errno`**. A trailing newline is added if needed.

This message is read by **`syslogd`** and written to the system console, log files, selected users' terminals, or forwarded to **`syslogd`** on another host as appropriate.

priority is encoded as the logical OR of a *level* and a *facility*. The *level* signifies the urgency of the message, and *facility* signifies the subsystem generating the message. *facility* can be encoded explicitly in *priority*, or a default *facility* can be set with **`openlog()`** (see below).

level is selected from an ordered list:

`LOG_EMERG`	A panic condition. This is normally broadcast to all users.
`LOG_ALERT`	A condition that should be corrected immediately, such as a corrupted system database.
`LOG_CRIT`	Critical conditions, such as hard device errors.
`LOG_ERR`	Errors.
`LOG_WARNING`	Warning messages.
`LOG_NOTICE`	Conditions that are not error conditions, but should possibly be handled specially.
`LOG_INFO`	Informational messages.
`LOG_DEBUG`	Messages that contain information normally of use only when debugging a program.

`syslog()` does not log a message that does not have a *level* set.

If **`syslog()`** cannot pass the message to **`syslogd`**, it attempts to write the message on **`/dev/console`** if the **`LOG_CONS`** option is set (see below).

`openlog()` can be called to initialize the log file, if special processing is needed. *ident* is a string that precedes every message. *logopt* is a mask of bits, logically OR'ed together, indicating logging options. The values for *logopt* are:

`LOG_PID`	Log the process ID with each message; useful for identifying instantiations of daemons.
`LOG_CONS`	Force writing messages to the console if unable to send it to **`syslogd`**. This option is safe to use in daemon processes that have no controlling terminal because **`syslog()`** forks before opening the console.
`LOG_NDELAY`	Open the connection to **`syslogd`** immediately. Normally, the open is delayed until the first message is logged. This is useful for programs that need to manage the order in which file descriptors are allocated.

LOG_NOWAIT Do not wait for children forked to log messages on the console. This option should be used by processes that enable notification of child termination via **SIGCLD**, because **syslog()** might otherwise block, waiting for a child whose exit status has already been collected.

facility encodes a default facility to be assigned to all messages written subsequently by **syslog()** with no explicit facility encoded.

LOG_KERN Messages generated by the kernel. These cannot be generated by any user processes.

LOG_USER Messages generated by random user processes. This is the default facility identifier if none is specified.

LOG_MAIL The mail system.

LOG_DAEMON System daemons, such as *inetd*(1M), *ftpd*(1M), etc.

LOG_AUTH The authorization system: *login*(1), *su*(1), *getty*(1M), etc.

LOG_LPR The line printer spooling system: *lp*(1), *lpsched*(1M), etc.

LOG_LOCAL0 Reserved for local use. Similarly for **LOG_LOCAL1** through **LOG_LOCAL7**.

closelog() closes the log file.

setlogmask()
sets the log priority mask to *maskpri* and returns the previous mask. Calls to **syslog()** with a priority not set in *maskpri* are rejected. The mask for an individual priority *pri* is calculated by the macro **LOG_MASK(***pri***)**; the mask for all priorities up to and including *toppri* is given by the macro **LOG_UPTO**(*toppri*). By default, all priorities are logged.

APPLICATION USAGE

The interfaces **syslog()**, **openlog()**, **closelog()** and **setlogmask()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing **syslog()**, **openlog()** or **closelog()**.

ERRORS

syslog fails if any of the following conditions are encountered:

[EAGAIN] The named pipe **/dev/log** is blocked for writing.

[ENOENT] The named pipe **/dev/log** bold) could not be opened successfully.

EXAMPLES

who logs a message regarding some sort of unexpected and serious error:

```
syslog(LOG_ALERT, "who: internal error 23");
```

ftpd uses **openlog()** to arrange to log its process ID, to log to the console if necessary, and to log in the name of the *daemon* facility:

```
openlog("ftpd", LOG_PID|LOG_CONS, LOG_DAEMON);
```

Arrange to log messages only at levels **LOG_ERR** and lower:

```
setlogmask(LOG_UPTO(LOG_ERR));
```

Typical usage of **syslog()** to log a connection:

```
syslog(LOG_INFO, "Connection from host %d", CallingHost);
```

If the facility has not been set with **openlog()**, it defaults to **LOG_USER**.

Explicitly set the facility for this message:

```
syslog(LOG_INFO|LOG_LOCAL2, "foobar error: %m");
```

WARNINGS

A call to **syslog()** has no effect unless the syslog daemon (*syslogd*(1M)) is running. **openlog()** does not copy and store the *ident* string internally; it stores only a character pointer. Therefore it is the

responsibility of the programmer to make sure that the *ident* argument points to the correct string until the log file is closed.

AUTHOR

syslog() was developed by the University of California, Berkeley.

SEE ALSO

logger(1), syslogd(1M).

NAME

system() - issue a shell command

SYNOPSIS

```
#include <stdlib.h>
int system(const char *command);
```

DESCRIPTION

system() executes the command specified by the string pointed to by *command*. The environment of the executed command is as if a child process were created using **fork()** (see *fork*(2)), and the child process invoked the *sh-posix*(1) utility via a call to **execl()** (see *exec*(2)) as follows:

```
execl("/usr/bin/sh", "sh", "-c", command, 0);
```

system() ignores the **SIGINT** and **SIGQUIT** signals, and blocks the **SIGCHLD** signal, while waiting for the command to terminate. If this might cause the application to miss a signal that would have killed it, the application should examine the return value from **system()** and take whatever action is appropriate to the application if the command terminated due to receipt of a signal.

system() does not affect the termination status of any child of the calling processes other than the process or processes it itself creates.

system() does not return until the child process has terminated.

APPLICATION USAGE

The interface **system()** is thread-safe. It is not async-cancel-safe. A cancellation point may occur when a thread is executing **system()**.

If the return value of **system()** is not –1, its value can be decoded through the use of the macros described in <**sys/wait.h**>. For convenience, these macros are also provided in <**stdlib.h**>.

Note that, while **system()** must ignore **SIGINT** and **SIGQUIT** and block **SIGCHLD** while waiting for the child to terminate, the handling of signals in the executed command is as specified by *fork*(2) and *exec*(2). For example, if **SIGINT** is being caught or is set to **SIG_DFL** when **system()** is called, the child is started with **SIGINT** handling set to **SIG_DFL**.

Ignoring **SIGINT** and **SIGQUIT** in the parent process prevents coordination problems (such as two processes reading from the same terminal) when the executed command ignores or catches one of the signals.

RETURN VALUE

If *command* is null, **system()** returns non-zero.

If *command* is not null, **system()** returns the termination status of the command language interpreter in the format specified by *wait*(2). The termination status of the command language interpreter is as specified for *sh-posix*(1), except that if some error prevents the command language interpreter from executing after the child process is created, the return value from **system()** is as if the command language interpreter had terminated using **_exit(127)**. If a child process cannot be created, or if the termination status for the command language interpreter cannot be obtained, **system()** returns –1 and sets **errno** to indicate the error.

DIAGNOSTICS

system() forks to create a child process which, in turn, **exec()**s **/usr/bin/sh** in order to execute *string*. If the fork fails, **system()** returns –1 and sets **errno**. If the exec fails, **system()** returns the status value returned by **waitpid()** (see *wait*(2)) for a process that terminates with a call of **exit(127)**.

ERRORS

If errors are encountered, **system()** sets **errno** values as described by *fork*(2).

FILES

/usr/bin/sh

SEE ALSO

sh(1), fork(2), exec(2), wait(2).

STANDARDS CONFORMANCE

system(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, POSIX.2, ANSI C

NAME

t_accept() - accept a connect request

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
/* or */
#include <tiuser.h>         /* for Transport Layer Interface - TLI  */

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

DESCRIPTION

The **t_accept()** function is issued by a transport user to accept a connect request. *fd* identifies the local transport endpoint where the connect indication arrived. *resfd* specifies the local transport endpoint where the connection is to be established. *call* contains information required by the transport provider to complete the connection. The parameter *call* points to a **t_call** structure which contains the following members:

```
struct netbuf addr;     /* address          */
struct netbuf opt;      /* options          */
struct netbuf udata;    /* user data        */
int sequence;           /* sequence number  */
```

The **netbuf** structure is defined in the **<xti.h>** or **<tiuser.h>** header file. This structure, which is used to define buffer parameters, has the following members:

unsigned int maxlen	maximum byte length of the data buffer
unsigned int len	actual byte length of data written to buffer
char *buf	points to buffer location

In *call*, the *addr*, *opt*, *udata*, and *sequence* parameters are explained here. *addr* is the protocol address of the calling transport user. *opt* indicates any options associated with the connection. For XTI over the OSI transport provider, this **netbuf** should point to a struct of type **isoco_options**. *udata* points to any user data to be returned to the caller. *sequence* is the value returned by **t_listen()** that uniquely associates the response with a previously received connect indication. The address of the caller, *addr*, may be null (length zero). When *addr* is not null, then it may optionally be checked by XTI.

A transport user may accept a connection on either the same, or on a different, local transport endpoint than the one on which the connect indication arrived. Before the connection can be accepted on the same endpoint (*resfd==fd*), the user must have responded to any previous connect indications received on that transport endpoint (via **t_accept()** or **t_snddis()**). Otherwise, **t_accept()** will fail and set **t_errno** to [TINDOUT].

If a different transport endpoint is specified (*resfd!=fd*), then the user may or may not choose to bind the endpoint before **t_accept()** is issued. If the endpoint is not bound prior to the **t_accept()**, then the transport provider will automatically bind it to the same protocol address *fd* is bound to. If the transport user chooses to bind the endpoint, it must be bound to a protocol address with a *qlen* of zero and must be in the T_IDLE state before the **t_accept()** is issued.

The call to **t_accept()** will fail with **t_errno** set to [TLOOK] if there are indications for example, connect or disconnect waiting to be received on the endpoint *fd*.

The *udata* argument enables the called transport user to send user data to the caller. The amount of user data must not exceed the limits supported by the transport provider as returned in the *connect* field of the *info* argument of **t_open()** or **t_getinfo()**. If the *len* field of *udata* is zero, no data will be sent to the caller. All the *maxlen* fields are meaningless.

When the user does not indicate any option (*call->opt.len* = 0), it is assumed that the connection is to be accepted unconditionally. The transport provider may choose options other than the defaults to ensure that the connection is accepted successfully.

Thread-Safeness

The **t_accept()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

Valid States
fd: **T_INCON**

resfd (fd != resfd): **T_IDLE**

Caveats
There may be transport provider-specific restrictions on address binding.

Some transport providers do not differentiate between a connect indication and the connection itself. If the connection has already been established after a successful return of **t_listen()**, **t_accept()** will assign the existing connection to the transport endpoint specified by *resfd*.

RETURN VALUE

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

ERRORS

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

[TBADF] The file descriptor, *fd* or *resfd*, does not refer to a transport endpoint, or the user is illegally accepting a connection of the same transport endpoint on which the connect indication arrived.

[TOUTSTATE] The function was issued in the wrong sequence on the transport endpoint referenced by *fd*, or the transport endpoint referred to by *resfd* is not in the appropriate state.

[TACCES] The user does not have permission to accept a connection on the responding transport endpoint or use the specified options.

[TBADOPT] The specified options were in an incorrect format or contained illegal information.

[TBADDATA] The amount of user data specified was not within the bounds allowed by the transport provider.

[TBADADDR] The specified protocol address was in an incorrect format or contained illegal information.

[TBADSEQ] An invalid sequence number was specified.

[TLOOK] An asynchronous event has occurred on the transport endpoint referenced by *fd* and requires immediate attention.

[TNOTSUPPORT] This function is not supported by the underlying transport provider.

[TSYSERR] A system error has occurred during execution of this function.

[TINDOUT] The function was called with *fd==resfd* but there are outstanding connection indications on the endpoint. Those other connection indications must be handled either by rejecting them via **t_snddis()** or accepting them on a different endpoint via **t_accept()**.

[TPROVMISTMATCH] (XTI only) The file descriptors *fd* and *resfd* do not refer to the same transport provider.

[TRESQLEN] The endpoint referenced by *resfd* (where *resfd != fd*) was bound to a protocol address with a *qlen* that is greater than zero.

[TPROTO] (XTI only) This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (**t_errno**).

[TRESADDR] This transport provider requires both *fd* and *resfd* to be bound to the same address. This error results if they are not.

FILES

/usr/include/xti.h XTI data structures

/usr/include/xti_iso.h XTI data structures

/usr/include/tiuser.h TLI data structures

SEE ALSO
t_connect(3), t_getstate(3), t_listen(3), t_open(3), t_rcvconnect(3).

STANDARDS CONFORMANCE
t_accept(): SVID2, XPG3, XPG4

NAME

t_alloc() - allocate a library structure

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
/* or */
#include <tiuser.h>         /* for Transport Layer Interface - TLI  */

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

DESCRIPTION

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

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

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

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

Each of the above structures, except T_INFO, contains at least one field of type **struct netbuf**. For each field of this type, the user may specify that the buffer for that field should be allocated as well. The length of the buffer allocated will be equal to or greater than the appropriate size returned in the *info* argument of **t_open()** or **t_getinfo().** The relevant fields of the *info* argument are described in the following list. The *fields* argument specifies which buffer to allocate, where the argument is the bitwise-OR of any of the following:

T_ADDR	The *addr* field of the **t_bind**, **t_call**, **t_unitdata**, or **t_uderr** structures.
T_OPT	The *opt* field of the **t_optmgmt**, **t_call**, **t_unitdata**, or **t_uderr** structures.
T_UDATA	The *udata* field of the **t_call**, **t_discon**, or **t_unitdata** structures.
T_ALL	All relevant fields of the given structure. Fields which are not supported by e transport provider specified by *fd* will not be allocated (*info* values are <= 0).

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

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

Thread-Safeness

The **t_alloc()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

Valid States

All - apart from **T_UNINIT**

RETURN VALUE

Upon successful completion, **t_alloc()** returns a pointer to the newly allocated structure. On failure, a null pointer is returned.

ERRORS

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

[TBADF] *fd* The specified endpoint identifier does not refer to a transport endpoint

[TSYSERR] A system error has occurred during execution of this function.

[TPROTO] (XTI only) This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no suitable XTI (**t_errno**).

[TNOSTRUCTYPE] Unsupported *struct_type* requested. This can include a request for a structure type which is inconsistent with the transport provider type specified, that is, connection-oriented or connectionless.

SEE ALSO

t_free(3), t_getinfo(3), t_open(3).

STANDARDS CONFORMANCE

R t_alloc() ": SVID2, XPG3, XPG4"

NAME

t_bind() - bind an address to a transport endpoint

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
/* or */
#include <tiuser.h>         /* for Transport Layer Interface - TLI  */

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

DESCRIPTION

The **t_bind()** function associates a protocol address with the transport endpoint specified by *fd* and activates that transport endpoint. In connection mode, the transport provider may begin enqueuing incoming connect indications or servicing a connection request on the transport endpoint. In connectionless mode, the transport user may send or receive data units through the transport endpoint.

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

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

The type **netbuf** structure is defined in the **<xti.h>** or **<tiuser.h>** header file. This structure, which is used to define buffer parameters, has the following members:

unsigned int maxlen	maximum byte length of the data buffer
unsigned int len	actual byte length of data written to buffer
char *buf	points to buffer location

The *addr* field of the **t_bind** structure specifies a protocol address. The *qlen* field is used to indicate the maximum number of outstanding connect indications.

The parameter *req* is used to request that an address, represented by the **netbuf** structure, be bound to the given transport endpoint. The parameter *len* specifies the number of bytes in the address, and *buf* points to the address buffer. The parameter *maxlen* has no meaning for the *req* argument. On return, *ret* contains the address of that the transport provider actually bound to the transport endpoint. This is the same as the address specified in *req*. In *ret*, the user specifies *maxlen*, which is the maximum size of the address buffer, and *buf* which points to the buffer where the address is to be placed. On return, *len* specifies the number of bytes in the bound address, and *buf* points to the bound address. If *maxlen* is not large enough to hold the returned address, an error will result.

If the request address is not available, **t_bind()** will return –1 with **t_errno** set as appropriate. If no address is specified in *req* (the *len* field of *addr* in *req* is zero or *req* is NULL), the transport provider will assign an appropriate address to be bound, and will return that address in the *addr* field of *ret*. If the transport provider could not allocate an address, **t_bind()** will fail with **t_errno** set to [TNOADDR]. HP OSI does not support the automatic generation of an address.

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

The *qlen* field has meaning only when initializing a connection-mode service. It specifies the number of outstanding connect indications that the transport provider should support for the given transport endpoint. An outstanding connect indication is one that has been passed to the transport user by the transport provider but which has not been accepted or rejected. A value of *qlen* greater than zero is only meaningful when issued by a passive transport user that expects others to call it. The value of *qlen* will be negotiated by the transport provider and may be changed if the transport provider cannot support the specified number of outstanding connect indications. However, this value of *qlen* will never be negotiated from a requested value greater than zero to zero. This is a requirement on transport providers; see *Caveats* below. On return, the *qlen* field in *ret* will contain the negotiated value.

If *fd* refers to a connection-mode service, this function allows more than one transport endpoint to be bound to the same protocol address (however, the transport provider must also support this capability), but it is not possible to bind more than one protocol address to the same transport endpoint. If a user binds more than one transport endpoint to the same protocol address, only one endpoint can be used to listen for the connect indications associated with that protocol address. In other words, only one **t_bind()** for a given protocol address may specify a value of *qlen* greater than zero. In this way, the transport provider can identify which transport endpoint should be notified of an incoming connect indication. If a user attempts to bind a protocol address to a second transport endpoint with a value of *qlen* greater than zero, **t_bind()** will return –1 and set **t_errno** to [TADDRBUSY] (XTI) or [TBADADDR] (TLI). When a user accepts a connection on the transport endpoint that is being used as the listening endpoint, the bound protocol address will be found to be busy for the duration of the connection, until a **t_unbind()** or **t_close()** call has been issued. No other transport endpoints may be bound for listening on the same protocol address while that initial listening endpoint is active (in the data transfer phase or in the T_IDLE state). This will prevent more than one transport endpoint bound to the same protocol address from accepting connect indications.

If *fd* refers to a connectionless-mode service, only one endpoint may be associated with a protocol address. If a user attempts to bind a second transport endpoint to an already bound address, **t_bind()** will return –1 and set **t_errno** to [TADDRBUSY].

Thread-Safeness

The **t_bind()** function is safe to be called by multi-threaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

Valid States

T_UNBND

Note

HP XTI does not support automatic generation of addresses. Therefore a valid local transport address must be specified in *req*.

Caveats

The requirement that the value of *qlen* never be negotiated from a requested value greater than zero to zero implies that transport providers, rather than the XTI implementation itself, accept this restriction.

A transport provider may not allow an explicit binding of more than one transport endpoint to the same protocol address although it allows more than one connection to be accepted for the same protocol address. To ensure portability, it is, therefore, recommended not to bind transport endpoints that are used as responding endpoints (*resfd*) in a call to **t_accept()** if the responding address is to be the same as the called address.

RETURN VALUE

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

ERRORS

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

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

[TPROTO] (XTI only) This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no suitable XTI (**t_errno**).

FILES

/usr/include/xti.h	XTI data structures
/usr/include/xti_iso.h	XTI data structures
/usr/include/tiuser.h	TLI data structures

SEE ALSO

t_alloc(3), t_close(3), t_open(3), t_unbind(3).

STANDARDS CONFORMANCE

t_bind(): SVID2, XPG3, XPG4

t

NAME

t_close() - close a transport endpoint

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
/* or */
#include <tiuser.h>         /* for Transport Layer Interface - TLI  */

int t_close (fd)
int  fd;
```

DESCRIPTION

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

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

Thread-Safeness

The **t_close()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

Valid States

All - apart from T_UNINIT

RETURN VALUE

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

ERRORS

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TPROTO]	(XTI only) This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no suitable XTI (**t_errno**).

SEE ALSO

t_getstate(3), t_open(3), t_unbind(3).

STANDARDS CONFORMANCE

t_close(): SVID2, XPG3, XPG4

t

NAME

t_connect() - establish a connection with another transport user

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
/* or */
#include <tiuser.h>         /* for Transport Layer Interface - TLI  */

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

DESCRIPTION

This function enables a transport user to request a connection to the specified destination transport user. This function can only be issued in the T_IDLE state. *fd* identifies the local transport endpoint where communication will be established. *sndcall* and *rcvcall* point to a **t_call** structure which contains the following members:

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

The type **netbuf** structure is defined in the **<xti.h>** or **<tiuser.h>** header file. This structure, which is used to define buffer parameters, has the following members:

unsigned int maxlen	maximum byte length of the data buffer
unsigned int len	actual byte length of data written to buffer
char *buf	points to buffer location

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

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

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

The *opt* argument permits users to define the options that may be passed to the transport provider. These options are specific to the underlying protocols of the transport provider and are described for ISO and TCP protocols in Appendix A, "ISO Transport Protocol Information," Appendix B, "Internet Protocol-specific Information," and Appendix F, "Headers and Definitions" of the *CAE Specification X/Open Transport Interface (XTI)* manual. The user may choose not to negotiate protocol options by setting the *len* field of *opt* to zero. In this case, the provider may use default options.

If the *opt* argument is used, the *sndcall->opt.buf* structure must point to the corresponding options structures. For XTI over the OSI transport provider, the options buffer should be a struct of type **isoco_options** or **tcp_options**. For TLI, see the documentation for the transport provider being used. The *maxlen* and *buf* fields of the **netbuf** structure pointed by *rcvcall->addr* and *rcvcall->opt* must be set before the call.

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

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

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

When a synchronous **t_connect()** call is interrupted by the arrival of a signal, the state of the corresponding transport endpoint is T_OUTCON, allowing a further call to either **t_rcvconnect()**, **t_rcvdis()**, or **t_snddis()**.

Thread-Safeness
The **t_connect()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

Valid States
T_IDLE

RETURN VALUE

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

ERRORS

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TOUTSTATE]	The function was issued in the wrong sequence.
[TNODATA]	**O_NONBLOCK** was set, so the function successfully initiated the connection establishment procedure, but did not wait for a response from the remote user.
[TBADADDR]	The specified protocol address was in an incorrect format or contained illegal information.
[TBADOPT]	The specified protocol options were in incorrect format or contained illegal information.
[TBADDATA]	The amount of user data specified was not within the bounds allowed by the transport provider.
[TACCES]	The user does not have permission to use the specified address or options.
[TBUFOVFLW]	The number of bytes allocated for an incoming argument (*maxlen*) is greater than zero but not sufficient to store the value of that argument. If executed in synchronous mode, the provider's state, as seen by the user, changes to T_DATAXFER, and the connect indication information to be returned in *rcvcall* is discarded.
[TLOOK]	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.
[TADDRBUSY]	This transport provider does not support multiple connections with the same local and remote addresses. This error indicates that a connection already exists.
[TPROTO]	(XTI only) This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no suitable XTI (**t_errno**).

FILES

/usr/include/xti.h	XTI data structures
/usr/include/tiuser.h	TLI data structures

SEE ALSO
fcntl(2), t_accept(3), t_alloc(3), t_getinfo(3), t_listen(3), t_open(3), t_rcvconnect(3).

STANDARDS CONFORMANCE
t_connect(): SVID2, XPG3, XPG4

NAME

t_error() - produce error message

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
/* or */
#include <tiuser.h>         /* for Transport Layer Interface - TLI  */

void t_error (errmsg)
char *errmsg;
extern int t_errno
extern char *t_errlist[]
extern int t_nerr;
```

DESCRIPTION

The **t_error()** function produces a language-dependent message on the standard error output which describes the last error encountered during a call to a transport function. The argument string *errmsg* is a user-supplied error message that gives context to the error.

The error message is written as follows:

First if *errmsg* is not a null pointer and the character pointed to be *errmsg* is not the null character, the string pointed to by *errmsg* is written followed by a colon and a space.

Then a standard error message string for the current error defined in **t_errno** is written. If **t_errno** has a value different from [TSYSERR], the standard error message string is followed by a newline character. If, however, **t_errno** is equal to [TSYSERR], the **t_errno** string is followed by the standard error message string for the current error defined in **errno** followed by a newline.

The language for error message strings written by **t_error()** is implementation-defined. If it is in English, the error message string describing the value in **t_errno** is identical to the comments following the **t_errno** codes defined in **<xti.h>**. The contents of the error message strings describing the value in **errno** are the same as those returned by the **strerror()** function with an argument of **errno**.

To simplify variant formatting of messages, the array of message strings **t_errlist** is provided; **t_errno** can be used as an index in this table to get the message string without the newline. The variable **t_nerr** is the largest message number provided for in the **t_errlist** table.

The error number, **t_errno**, is only set when an error occurs and it is not cleared on successful calls.

Thread-Safeness

The **t_error()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

Valid States

All - apart from T_UNINIT

RETURN VALUE

For XTI, upon completion, a value of 0 is returned. TLI does not return a value.

ERRORS

No errors are defined for the **t_error()** function.

EXAMPLE

If a **t_connect()** function fails on transport endpoint *fd2* because a bad address was given, the following call might follow the failure:

```
t_error("t_connect failed on fd2");
```

The diagnostic message to be printer would look like:

```
t_connect failed on fd2:   Incorrect address format
```

where **Incorrect address format** identifies the specific error that occurred, and **t_connect failed on fd2** tells the user which function failed on which transport endpoint.

FILES

/usr/lib/nls/msg/C/libnsl_s.cat	NLS message catalog for TLI

STANDARDS CONFORMANCE

t_error(): SVID2, XPG3, XPG4

NAME

t_free() - free a library structure

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
/* or */
#include <tiuser.h>         /* for Transport Layer Interface - TLI  */

int t_free (ptr, struct_type)
char *ptr;
int struct_type;
```

DESCRIPTION

The **t_free()** function frees memory previously allocated by **t_alloc()**. This function will free memory for the specified structure and will also free memory for buffers referenced by the structure.

The argument *ptr* points to one of the seven structure types described for **t_alloc()**. *struct_type* identifies the type of that structure which must be one of the following:

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

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

t_free() will check the *addr*, *opt*, and *udata* fields of the given structure (as appropriate) and free the buffers pointed to by the *buf* field of the **netbuf** structure. If *buf* is a null pointer, **t_free()** will not attempt to free memory. After all buffers are freed, **t_free()** will free the memory associated with the structure pointed to by *ptr*.

Undefined results will occur if *ptr* or any of the *buf* pointers points to a block of memory that was not previously allocated by **t_alloc()**.

Thread-Safeness

The **t_free()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

Valid States

All - apart from T_UNINIT

RETURN VALUE

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

ERRORS

On failure, **t_errno** is set to the following:

[TSYSERR] A system error has occurred during execution of this function.

[TNOSTRUCTYPE] Unsupported *struct_type* requested.

[TPROTO] (XTI only) This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no suitable XTI (**t_errno**).

SEE ALSO

t_alloc(3).

STANDARDS CONFORMANCE

t_free(): SVID2, XPG3, XPG4

NAME

t_getinfo() - get protocol-specific service information

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
/* or */
#include <tiuser.h>         /* for Transport Layer Interface - TLI  */

int t_getinfo (fd, info)
int fd;
struct info *info;
```

DESCRIPTION

The **t_getinfo()** function returns the current characteristics of the underlying transport protocol associated with file descriptor *fd*. The *info* structure is used to return the same information returned by **t_open()**. This function enables a transport user to access this information during any phase of communication.

This argument points to a **t_info** structure which contains the following members:

```
t_scalar_t addr;     /* max size of the transport protocol address */
t_scalar_t options;  /* max number of bytes of protocol-specific options */
t_scalar_t tsdu;     /* max size of a transport service data unit,TSDU */
t_scalar_t etsdu;    /* max size of expedited transport service data unit,ETSDU */
t_scalar_t connect;  /* max data allowed on connection establishment functions */
t_scalar_t discon;   /* max data allowed on t_snddis and t_rcvdis functions */
t_scalar_t servtype; /* service type supported by the transport provider */
t_scalar_t flags;    /* other info about the transport provider */
```

The values of the fields have the following meanings:

addr A value greater than or equal to zero indicates the maximum size of a transport protocol address.

A value of –1 specifies that there is no limit on the address size.

A value of –2 specifies that the transport provider does not provide user access to transport protocol addresses.

options A value greater than or equal to zero indicates the maximum number of bytes of protocol-specific options supported by the provider.

A value of –1 specifies that there is no limit on the option size.

A value of –2 specifies that the transport provider does not support user-settable options.

tsdu A value greater than zero specifies the maximum size of a transport service data unit (TSDU)

A value of zero specifies that the transport provider does not support the concept of TSDU although it does support the sending of a data stream with no logical boundaries preserved across a connection.

A value of –1 specifies that there is no limit on the size of a TSDU.

A value of –2 specifies that the transfer of normal data is not supported by the transport provider.

etsdu A value greater than zero specifies the maximum size of an expedited transport service data unit (ETSDU).

A value of zero specifies that the transport provider does not support the concept of ETSDU although it does support the sending of an expedited data stream with no logical boundaries preserved across a connection.

A value of –1 specifies that there is no limit on the size of an ETSDU.

A value of –2 specifies that the transfer of expedited data is not supported by the transport provider.

connect A value greater than or equal to zero specifies the maximum amount of data that may be associated with the connection establishment functions **t_connect()** and

t_rcvconnect().

A value of –1 specifies that there is no limit on the amount of data sent during connection establishment.

A value of –2 specifies that the transport provider does not allow data to be sent with connection establishment functions.

discon A value greater than or equal to zero specifies the maximum amount of data that may be associated with the **t_snddis()** and **t_rcvdis()** functions.

A value of –1 specifies that there is no limit on the amount of data sent with these abortive release functions.

A value of –2 specifies that the transport provider does not allow data to be sent with the abortive release functions.

servtype This field specifies the service type supported by the transport provider, as described below.

flags This is a bit field used to specify other information about the transport provider. If the T_SENDZERO bit is set in flags, this indicates that the underlying transport provider supports the sending of zero-length TSDUs. See Appendix A, "ISO Transport Protocol Information" of the *CAE Specification X/Open Transport Interface (XTI)* manual for a discussion of the separate issue of zero-length fragments within a TSDU. Note: HP currently does not support T_SENDZERO flag within the **timod** module.

If a transport user is concerned with protocol independence, the above sizes may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the **t_alloc()** function may be used to allocate these buffers. An error will result if a transport user exceeds the allowed data size on any function. The value of each field may change as a result of option negotiation, and **t_getinfo()** enables a user to retrieve the current characteristics of the underlying transport protocol.

The *servtype* field of *info* specifies one of the following values on return:

T_COTS The transport provider supports a connection-mode service but does not support the optional orderly release facility.

T_COTS_ORD The transport provider supports a connection-mode service with the optional orderly release facility.

T_CLTS The transport provider supports a connectionless-mode service. For this service type, **t_open()** will return –2 for *etsdu*, *connect*, and *discon*.

Thread-Safeness

The **t_getinfo()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

Valid States

All - apart from T_UNINIT

t

RETURN VALUE

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

ERRORS

On failure, **t_errno** is set to the following

[TBADF] The specified identifier does not refer to a transport endpoint.

[TSYSERR] A system error has occurred during execution of this function.

[TPROTO] (XTI only) This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no suitable XTI (**t_errno**).

SEE ALSO

t_alloc(3), t_open(3).

STANDARDS CONFORMANCE

t_getinfo(): SVID2, XPG3, XPG4

NAME

t_getprotaddr() - get the protocol address

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
int t_getprotaddr (fd, boundaddr, perraddr)
int fd;
struct t_bind *boundaddr;
struct t_bind *peeraddr;
```

DESCRIPTION

The **t_getprotaddr()** function returns local and remote protocol addresses currently associated with the transport endpoint specified by *fd*. In *boundaddr* and *peeraddr* the user specifies *maxlen*, which is the maximum size of the address buffer, and *buf* which points to the buffer where the address is to be placed. On return, the *buf* field of *boundaddr* points to the address, if any, currently bound to *fd*, and the *len* field specifies the length of the address. If the transport endpoint is in the T_UNBND state, zero is returned in the *len* field of *boundaddr*. The *buf* field of *peeraddr* points to the address, if any, currently connected to *fd*, and the *len* field specifies the length of the address. If the transport endpoint is not in the T_DATAXFER state, zero is returned in the *len* field of *peeraddr*.

Thread-Safeness

The **t_getprotaddr()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

Valid States

All - apart from T_UNINIT

RETURN VALUE

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

ERRORS

On failure, **t_errno** is set to the following

[TBADF]	The specified identifier does not refer to a transport endpoint.
[TBUFOVFLW]	The number of bytes allocated for an incoming argument (*maxlen*) is greater than 0 but not sufficient to store the value of that argument.
[TSYSERR]	A system error has occurred during execution of this function.
[TPROTO]	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (**t_errno**).

SEE ALSO

t_bind(3).

NAME

t_getstate() - get the current state

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
/* or */
#include <tiuser.h>         /* for Transport Layer Interface - TLI  */

int t_getstate (fd)
int fd;
```

DESCRIPTION

The **t_getstate()** function returns the current state of the provider as seen by the application associated with the transport endpoint specified by *fd*.

Thread-Safeness

The **t_getstate()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

RETURN VALUE

State is returned upon successful completion. Otherwise, a value of –1 it returned and **t_errno** is set to indicate the error. The current state is one of the following:

T_UNBND	unbound
T_IDLE	idle
T_OUTCON	outgoing connection pending
T_INCON	incoming connection pending
T_DATAXFER	data transfer
T_OUTREL	outgoing orderly release (waiting for an orderly release indication).
T_INREL	incoming orderly release (waiting for an orderly release request).

If the provider is undergoing a state transition when **t_getstate()** is called, the function will fail.

ERRORS

On failure, **t_errno** is set to the following:

[TBADF]	The specified identifier does not refer to a transport endpoint.
[TSTATECHNG]	The transport provider is undergoing a transient state change.
[TSYSERR]	A system error has occurred during execution of this function.
[TPROTO]	(XTI only) This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no suitable XTI (**t_errno**).

SEE ALSO

t_open(3).

STANDARDS CONFORMANCE

t_getstate(): SVID2, XPG3, XPG4

NAME

t_listen() - listen for a connect request

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
/* or */
#include <tiuser.h>         /* for Transport Layer Interface - TLI  */

int t_listen (fd, call)
int fd;
struct t_call *call;
```

DESCRIPTION

The **t_listen()** function listens for a connect request from a calling transport user. *fd* identifies the local transport endpoint where connect indications arrive. On return, *call* contains information describing the connect indication. The parameter *call* points to a **t_call** structure which contains the following members:

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

The type **netbuf** structure is defined in the **<xti.h>** or **<tiuser.h>** header file. This structure, which is used to define buffer parameters, has the following members:

unsigned int maxlen	maximum byte length of the data buffer
unsigned int len	actual byte length of data written to buffer
char *buf	points to buffer location

In *call*, *addr* returns the protocol address of the calling transport user. This address is in a format usable in future calls to **t_connect()**. Note, however that **t_connect()** may fail for other reasons, for example [TADDRBUSY].

opt returns protocol-specific parameters associated with the connect request. For XTI over the OSI transport provider, struct **isoco_options** should be used. For TLI, see the documentation for the transport provider being used.

udata returns any user data sent by the caller on the connect request.

sequence is a number that uniquely identifies the returned connect indication. The value of *sequence* enables the user to listen for multiple connect indications before responding to any of them.

Since this function returns values for the *addr*, *opt*, and *udata* fields of *call*, the *maxlen* field of each must be set before issuing the **t_listen** to indicate the maximum size of the buffer for each.

By default, **t_listen** executes in synchronous mode and waits for a connect indication to arrive before returning to the user. However, if **O_NONBLOCK** is set (via **t_open()** or **fcntl()**), **t_listen()** executes asynchronously, reducing to a poll for existing connect indications. If none are available, it returns –1 and sets **t_errno** to [TNODATA].

Thread-Safeness

The **t_listen()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

Valid States

T_IDLE, T_INCON

Caveats

Some transport providers do not differentiate between a connect indication and the connection itself. If this is the case, a successful return of **t_listen()** indicates an existing connection (see Appendix B, "Internet Protocol-specific Information," of the *CAE Specification X/Open Transport Interface (XTI)* manual).

RETURN VALUE

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

ERRORS

On failure, **t_errno** is set to the following:

[TBADF] The specified identifier does not refer to a transport endpoint.

[TBADQLEN] The argument *qlen* of the endpoint referenced by *fd* is zero.

[TBUFOVFLW] The number of bytes allocated for an incoming argument (*maxlen*) is not sufficient to store the value of that argument. The provider's state, as seen by the user, changes to T_INCON, and the connect indication information to be returned in *call* is discarded. For XTI only, the value of *sequence* returned can be used to do a **t_snddis()**.

[TNODATA] **O_NONBLOCK** was set, but no connect indication has been queued.

[TLOOK] An asynchronous event has occurred on this transport endpoint and requires immediate attention.

[TOUTSTATE] This function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

[TSYSERR] A system error has occurred during execution of this function.

[TQFULL] The maximum number of outstanding indications has been reached for the endpoint referenced by *fd*.

[TPROTO] (XTI only) This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no suitable XTI (**t_errno**).

FILES

/usr/include/xti.h XTI data structures

/usr/include/tiuser.h TLI data structures

SEE ALSO

fcntl(2), t_accept(3), t_alloc(3), t_bind(3), t_connect(3), t_open(3), t_snddis(3), t_rcvconnect(3).

STANDARDS CONFORMANCE

t_listen(): SVID2, XPG3, XPG4

t

NAME

t_look() - look at the current event on a transport endpoint

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
/* or */
#include <tiuser.h>         /* for Transport Layer Interface - TLI  */

int t_look (fd)
int fd;
```

DESCRIPTION

The **t_look()** function returns the current event on the transport endpoint specified by *fd*. This function enables a transport provider to notify a transport user of an asynchronous event when the user is calling functions in synchronous mode. Certain events require immediate notification of the user and are indicated by a specific error, [TLOOK], on the current or next function to be executed.

This function also enables a transport user to poll a transport endpoint periodically for asynchronous events.

Thread-Safeness

The **t_look()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

Valid States

All - apart from T_UNINIT

XTI Internet Protocol-specific Information

As soon as a segment with the TCP urgent pointer set enters the TCP receive buffer, the event T_EXDATA is indicated. T_EXDATA remains set until all data up to the byte pointed to by the TCP urgent pointer has been received. If the urgent pointer is updated, and the user has not yet received the byte previously pointed to by the urgent pointer, the update is transparent to the user.

RETURN VALUE

Upon success, **t_look()** returns a value that indicates which of the following allowable events has occurred, or returns zero if no event exists. One of the following events is returned:

T_LISTEN	connection indication received.
T_CONNECT	connect confirmation received.
T_DATA	normal data received.
T_ERROR	(TLI ONLY) fatal error occurred.
T_EXDATA	expedited data received.
T_DISCONNECT	disconnect received.
T_UDERR	datagram error indication.
T_ORDREL	orderly release indication.
T_GODATA	(XTI only) flow control restrictions on normal data flow have been lifted. Normal data may be sent again.
T_GOEXDATA	(XTI only) flow control restrictions on expedited data flow have been lifted. Expedited data may be sent again.

ERRORS

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

[TBADF]	The specified identifier does not refer to a transport endpoint.
[TSYSERR]	A system error has occurred during execution of this function.
[TPROTO]	(XTI only) This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no suitable XTI (**t_errno**).

SEE ALSO
t_open(3), t_snd(3), t_sndudata(3).

STANDARDS CONFORMANCE
t_look(): SVID2, XPG3, XPG4

NAME

t_open() - establish a transport endpoint

SYNOPSIS

```
#include <fcntl.h>
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
/* or */
#include <tiuser.h>         /* for Transport Layer Interface - TLI  */

int t_open (name, oflag, info)
char *name;
int  oflag;
struct t_info *info;
```

DESCRIPTION

The **t_open()** function must be called as the first step in the initialization of a transport endpoint. This function establishes a transport endpoint by opening a file that identifies a particular transport provider and returning a file descriptor that identifies that endpoint.

The argument *name* points to a file name that identifies a transport provider. When using HP XTI to connect to the OSI protocol stack, *name* must be **/dev/ositpi** for the connection-mode service, or **/dev/osicltpi** for the connectionless service. When using HP XTI to connect to the TCP protocol stack, *name* should be **/dev/tcp**. For UDP, *name* should be **/dev/udp**. (See also the *OBSOLESCENT INTERFACES* section.) For TLI, use the device file name of the transport provider desired. Note that HP TCP/UDP/IP and HP OSI COTS and CLTS only support XTI.

oflag identifies any open flags (as in **open()**) and is constructed from **O_RDWR** optionally or-ed with **O_NONBLOCK**. These flags are defined by the header file <**fcntl.h**>. **t_open()** returns a file descriptor that will be used by all subsequent functions to identify the particular local transport endpoint.

This function also returns various default characteristics of the underlying transport protocol by setting fields in the **t_info** structure. This argument points to a **t_info** which contains the following members:

```
t_scalar_t addr;      /* max size of the transport protocol */
                      /* address */
t_scalar_t options;   /* max number of bytes of */
                      /* protocol-specific options */
t_scalar_t tsdu;      /* max size of a transport service data */
                      /* unit (TSDU) */
t_scalar_t etsdu;     /* max size of an expedited transport */
                      /* service data unit (ETSDU) */
t_scalar_t connect;   /* max amount of data allowed on */
                      /* connection establishment functions */
t_scalar_t discon;    /* max amount of data allowed on */
                      /* t_snddis and t_rcvdis functions */
t_scalar_t servtype;  /* service type supported by the */
                      /* transport provider */
t_scalar_t flags;     /* other info about the transport provider */
```

The values of the fields have the following meanings:

addr A value greater than or equal to zero indicates the maximum size of a transport protocol address.

A value of –1 specifies that there is no limit on the address size.

A value of –2 specifies that the transport provider does not provide user access to transport protocol addresses.

options A value greater than or equal to zero indicates the maximum number of bytes of protocol-specific options supported by the provider.

A value of –1 specifies that there is no limit on the option size.

A value of –2 specifies that the transport provider does not support user-settable options.

tsdu A value greater than zero specifies the maximum size of a transport service data unit (TSDU).

A value of zero specifies that the transport provider does not support the concept of TSDU, although it does support the sending of a data stream with no logical boundaries preserved across a connection.

A value of –1 specifies that there is no limit on the size of a TSDU.

A value of –2 specifies that the transfer of normal data is not supported by the transport provider.

etsdu A value greater than zero specifies the maximum size of an expedited transport service data unit (ETSDU).

A value of zero specifies that the transport provider does not support the concept of ETSDU, although it does support the sending of an expedited data stream with no logical boundaries preserved across a connection.

A value of –1 specifies that there is no limit on the size of an ETSDU.

A value of –2 specifies that the transfer of expedited data is not supported by the transport provider.

connect A value greater than or equal to zero specifies the maximum amount of data that may be associated with the connection establishment functions **t_connect()** and **t_rcvconnect()**.

A value of –1 specifies that there is no limit on the amount of data sent during connection establishment.

A value of –2 specifies that the transport provider does not allow data to be sent with connection establishment functions.

discon A value greater than or equal to zero specifies the maximum amount of data that may be associated with the and functions.

A value of –1 specifies that there is no limit on the amount of data sent with these abortive release functions.

A value of –2 specifies that the transport provider does not allow data to be sent with the abortive release functions.

servtype This field specifies the service type supported by the transport provider, as described below.

flags This is a bit field used to specify other information about the transport provider. If the T_SENDZERO bit is set in flags, this indicates the underlying transport provider supports the sending of zero-length TSDUs. See Appendix A, "ISO Transport Protocol Information," of the *CAE Specification X/Open Transport Interface (XTI)* manual from X/Open Company Limited for a discussion of the separate issue of zero-length fragments within a TSDU.

If a transport user is concerned with protocol independence, the above sizes may be accessed to determine how large the buffer must be to hold each piece of information. Alternatively, the **t_alloc()** function may be used to allocate these buffers. An error will result if a transport user exceeds the allowed data size on any function.

The *servtype* field of *info* specifies one of the following values on return:

T_COTS The transport provider supports a connection-mode service but does not support the optional orderly release facility.

T_COTS_ORD The transport provider supports a connection-mode service with the optional orderly release facility.

T_CLTS The transport provider supports a connectionless-mode service. For this service type, **t_open()** will return –2 for *etsdu*, *connect*, and *discon*.

A single transport endpoint may support only one of the above services at one time.

If *info* is set to a null pointer by the transport user, no protocol information is returned by **t_open()**.

Thread-Safeness

The **t_open()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

Obsolescent Interfaces
When using TCP and UDP through HP XTI, *name* can be **/dev/inet_cots** and **/dev/inet_clts**, respectively. These names are supported for backward-compatibility purposes. They might not be supported in future releases, and they should be replaced with **/dev/tcp** and **/dev/udp**, respectively.

RETURN VALUE

A valid endpoint identifier is returned upon successful completion. Otherwise, a value of –1 is returned and **t_errno** is set to indicate an error.

HP OSI does not support the **T_COTS_ORD** servtype.

ERRORS

On error, **t_errno** is set to one of the following:

[TBADFLAG] An invalid flag is specified.

[TBADNAME] Invalid transport provider name.

[TSYSERR] A system error has occurred during execution of this function.

[TPROTO] (XTI only) This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no suitable XTI (**t_errno**).

SEE ALSO

open(2), t_sync(3).

STANDARDS CONFORMANCE

t_open(): SVID2, XPG3, XPG4

NAME

t_optmgmt() - manage options for a transport endpoint

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
/* or */
#include <tiuser.h>         /* for Transport Layer Interface - TLI  */

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

DESCRIPTION

The **t_optmgmt()** function enables a transport user to retrieve, verify or negotiate protocol options with the transport provider. The argument *fd* identifies a bound transport endpoint.

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

```
struct netbuf opt;
t_scalar_t flags;
```

The *opt* field identifies protocol options. The *flags* field is used to specify the action to take with those options.

The options are represented by a **netbuf** structure in a manner similar to the address in **t_bind()**. The argument *req* is used to request a specific action of the provider and to send options to the provider. The argument *len* specifies the number of bytes in the options. *buf* points to the options buffer. *maxlen* has no meaning for the *req* argument. For XTI over the OSI transport provider, the options buffer should be of struct **isoco_options** for connection-oriented service, or struct **isocl_options** for connectionless service. For TLI, see the documentation of the transport provider being used.

The transport provider may return options and flag values to the user through *ret*. For *ret*, maxlen specifies the maximum size of the options buffer, and *buf* points to the buffer where the options are to be placed. For XTI over the OSI transport provider, the options buffer should be of struct **isoco_options** for connection-oriented service, or struct **isocl_options** for connectionless service. For TLI, see the documentation of the transport provider being used. On return, *len* specifies the number of bytes of options returned. The value in *maxlen* has no meaning for the *req* argument, but must be set in the *ret* argument to specify the maximum number of bytes the options buffer can hold. The actual content of the options is imposed by the transport provider.

Each option in the options buffer is of the form **struct t_opthdr** possibly followed by an option value.

The *level* field of struct **t_opthdr** identifies the XTI level or a protocol of the transport provider. The *name* field identifies the option within the level. *len* contains its total length; i.e. the length of the option header **t_opthdr** plus the length of the option value. If **t_optmgmt()** is called with the action T_NEGOTIATE set, the *status* field of the returned options contains information about the success or failure of a negotiation.

Each option in the input or output option buffer must start at a **t_uscalar_t** boundary. The macro **OPT_NEXTHDR(pbuf, buflen, option)** can be used for that purpose. The parameter *pbuf* denotes a pointer to an option buffer *opt.buf*, and *buflen* is its length. The parameter *option* points to a current option in the option buffer. **OPT_NEXTHDR** returns a pointer to the position of the next option or returns a null pointer if the option buffer is exhausted. The macro is helpful for writing and reading. See **<xti.h>** in the *CAE Specification X/Open Transport Interface (XTI)* manual from X/Open Company Limited for the exact definition.

If the transport user specifies several options on input, all options must address the same level.

If any option in the options buffer does not indicate the same level as the first option, or the level specified is unsupported, then the **t_optmgmt()** request will fail with [TBADOPT]. If the error is detected, some options have possibly been successfully negotiated. The transport user can check the current status by calling **t_optmgmt()** with the T_CURRENT flag set.

The *flags* field of *req* must specify one of the following actions:

T_NEGOTIATE This action enables the transport user to negotiate option values.

The user specifies the options of interest and their values in the buffer specified by *req->opt.buf* and *req->opt.len*. The negotiated option values are returned in the buffer pointed to by *ret->opt.buf*. The status field of each returned option is set to indicate the result of the negotiation. The status is one of the following:

T_SUCCESS if the proposed value was negotiated.

T_PARTSUCCESS if a degraded value was negotiated.

T_FAILURE if the negotiation failed according to the negotiation rules.

T_NOTSUPPORT if the transport provider does not support this option or illegally requests negotiation of a privileged option.

T_READONLY if modification of a read-only option was requested.

If the status is T_SUCCESS, T_FAILURE, T_NOTSUPPORT or T_READONLY, the returned option value is the same as the one requested on input.

The overall result of the negotiation is returned in *ret->flags*.

This field contains the worst single result, whereby the rating is done according to the order of T_NOTSUPPORT, T_READONLY, T_FAILURE, T_PARTSUCCESS, T_SUCCESS. The value T_NOTSUPPORT is the worst result and T_SUCCESS is the best.

For each level, the option T_ALLOPT can be requested on input. No value is given with this option; only the **t_opthdr** part is specified. This input requests to negotiate all supported options of this level to their default values. The result is returned option by option in *ret->opt.buf*. (Note that depending on the state of the transport endpoint, not all requests to negotiate the default may be successful.)

T_CHECK This action enables the user to verify whether the options specified in *req* are supported by the transport provider.

If an option is specified with no option value (it consists only of a **t_opthdr** structure), the option is returned with its *status* field set to T_SUCCESS if it is supported, T_NOTSUPPORT if it is not or needs additional user privileges, and T_READONLY if it is read-only (in the current state). No option value is returned.

If an option is specified with an option value, the *status* field of the returned option has the same value, as if the user had tried to negotiate this value with T_NEGOTIATE. If the status is T_SUCCESS, T_FAILURE, T_NOTSUPPORT, or T_READONLY, the returned option value is the same as the one requested on input.

The overall result of the option checks is returned in *ret->flags*. This field contains the worst single result of the option checks, whereby the rating is the same as for T_NEGOTIATE.

Note that no negotiation takes place. All currently effective option values remain unchanged.

T_DEFAULT This action enables the transport user to retrieve the default option values. The user specifies the option of interest in *req->opt.buf*. The option values are irrelevant and will be ignored; it is sufficient to specify the **t_opthdr** part of an option only. The default values are then returned in *ret->opt.buf*.

The *status* field returned is one of the following:

T_NOTSUPPORT if the protocol level does not support this option or if the transport user illegally requested a privileged option.

T_READONLY if the option is read-only.

T_SUCCESS in all other cases.

The overall result of the request is returned in *ret->flags*. This field contains the worst single result, whereby the rating is the same as for T_NEGOTIATE.

For each level, the option T_ALLOPT can be requested on input. All supported options of this level with their default values are then returned. In this case, *ret->opt.maxlen* must be given at least the value *info->options* (see *t_getinfo*(3), *t_open*(3)) before the call.

T_CURRENT This action enables the transport user to retrieve the currently effective option values. The user specifies the options of interest in *req->opt.buf*. The option values are irrelevant and will be ignored; it is sufficient to specify the **t_opthdr** part of an option only. The default values are then returned in *ret->opt.buf*.

The *status* field returned is one of the following:

T_NOTSUPPORT if the protocol level does not support this option or if the transport user illegally requested a privileged option.

T_READONLY if the option is read-only.

T_SUCCESS in all other cases.

The overall result of the request is returned in *ret->flags*. This field contains the worst single result, whereby the rating is the same as for T_NEGOTIATE.

For each level, the option T_ALLOPT can be requested on input. All supported options of this level with their currently effective values are then returned.

The option T_ALLOPT can only be used with **t_optmgmt()** and the actions T_NEGOTIATE, T_DEFAULT, and T_CURRENT. It can be used with any supported level and addresses all supported options of this level. The option has no value; it consists of a **t_opthdr** only. Since in a **t_optmgmt()** call only options of one level may be addressed, this option should not be requested together with other options. The function returns as soon as this option has been processed.

Options are independently processed in the order they appear in the input option buffer. If an option is multiply input, it depends on the implementation whether it is multiply output or whether it is returned only once.

Transport providers may not be able to provide an interface capable of supporting T_NEGOTIATE and/or T_CHECK functionalities. When this is the case, the error [T_NOTSUPPORT] is returned.

The function **t_optmgmt()** may block under various circumstances and depending on the implementation. The function will block, for instance, if the protocol addressed by the call resides on a separate controller. It may also block due to flow control constraints, i.e. if data set previously across this transport endpoint has not yet been fully processed. If the function is interrupted by a signal, the option negotiations that have been done so far may remain valid. The behavior of the function is not changed if O_NONBLOCK is set.

XTI-Level Options

XTI level options are not specific for a particular transport provider. An XTI implementation supports none, all or any subset of the options defined below. An implementation may restrict the use of any of these options by offering them only in the privileged or read-only mode, or if *fd* relates to specific transport providers.

The subsequent options are not association-related. They may be negotiated in all XTI states except T_UNINIT.

The protocol level is XTI_GENERIC. For this level, the following options are defined. A request for XTI_DEBUG is an absolute requirement. A request to activate XTI_LINGER is an absolute requirement; the timeout value to this option is not. XTI_RCVBUF, XTI_RECVLOWAT, XTI_SNDBUF, and XTI_SNDLOWAT are not absolute requirements.

XTI_DEBUG This option enables debugging. The values of this option are implementation- defined. Debugging is disabled if the option is specified with "no value", i.e. with an option header only.

The system supplies utilities to process the traces. Note that an implementation may also provide other means for debugging.

XTI_LINGER This option is used to linger the execution of a **t_close()** or **close()** if send data is still queued in the send buffer. The option value specifies the linger period. If a **close()** or **t_close()** is issued and the send buffer is not empty, the system

attempts to send the pending data within the linger period before closing the endpoint. Data still pending after the linger period has elapsed is discarded.

Depending on the implementation, **t_close()** or **close()** either block for at maximum the linger period, or immediately return, whereupon the system holds the connection in existence for at most the linger period.

The option constists of a structure **t_linger** declared as:

```
struct t_linger{
     t_uscalar_t l_onoff;  /* switch option on/off     */
     t_uscalar_t l_linger; /* linger period in seconds */
}
```

Legal values for the *l_onoff* are:

T_NO switch option off

T_YES activate option

The value of *l_onoff* is an absolute requirement.

The field *l_linger* determines the linger period in seconds. The transport user can request the default value by setting the field to T_UNSPEC. The default timeout value depends on the underlying transport provider (it is often T_INFINITE). Legal values for this field are T_UNSPEC, T_INFINITE, and all non-negative numbers.

The *l_linger* value is not an absolute requirement. The implementation may place upper and lower limits to this value. Requests that fall short of the lower limit are negotiated to the lower limit.

Note that this option does not linger the execution of **t_snddis()**.

XTI_RCVBUF This option is used to adjust the internal buffer size allocated for the receive buffer. The buffer size may be increased for high-volume connections, or decreased to limit the possible backlog of incoming data.

This request is not an absolute requirement. The implementation may place upper and lower limits on the option value. Requests that fall short of the lower limit are negotiated to the lower limit.

Legal values are all positive numbers.

XTI_RCVLOWAT

This option is used to set a low-water mark in the receive buffer. The option values gives the minimal number of bytes that must have accumulated in the receive buffer before they become visible to the transport user. If and when the amount of accumulated receive data exceeds the low-water mark, a T_DATA event is created, an event mechanism (e.g. **poll()** or **select()**) indicates the data, and the data can be read by **t_rcv()** or **t_rcvudata()**.

This request is not an absolute requirement. The implementation may place upper and lower limits on the option value. Requests that fall short of the lower limit are negotiated to the lower limit.

Legal values are all positive numbers.

XTI_SNDBUF This option is used to adjust the internal buffer size allocated for the send buffer.

This request is not an absolute requirement. The implementation may place upper and lower limits on the option value. Requests that fall short of the lower limit are negotiated to the lower limit.

Legal values are all positive numbers.

XTI_SNDLOWAT

This option is used to set a low-water mark in the send buffer. The option value gives the minimal number of bytes that must have accumulated in the send buffer before they are sent.

This request is not an absolute requirement. The implementation may place upper and lower limits on the option value. Requests that fall short of the lower limit are negotiated to the lower limit.

Legal values are all positive numbers.

Thread-Safeness
The **t_optmgmt()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

Valid States
All - apart from T_UNINIT

RETURN VALUE

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

TLI supports any transport provider which is compliant with TPI (Transport Provider Interface). Users can access TLI versions of the **t_*** routines by linking with **/usr/lib/libnsl_s.a**. For more information on TLI, see the TLI section of *STREAMS/UX for HP 9000 Reference Manual*.

ERRORS

On failure, **t_errno** is set to the following:

[TBADF]	The specified identifier does not refer to a transport endpoint.
[TOUTSTATE]	The function was issued in the wrong sequence.
[TACCES]	The user does not have the permission to negotiate the specified options.
[TBADOPT]	The specified protocol options were in an incorrect format or contained illegal information.
[TBADFLAG]	An invalid flag was specified.
[TBUFOVFLW]	The number of bytes allowed for an incoming argument (*maxlen*) is greater than 0 and not sufficient to store the value of that argument. The information to be returned in *ret* will be discarded.
[TSYSERR]	A system error has occurred during execution of this function.
[TPROTO]	(XTI only) This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no existing XTI **t_errno**.
[TNOTSUPPORT]	(XTI only) This action is not supported by the underlying transport provider.

SEE ALSO

t_accept(3), t_alloc(3), t_connect(3), t_getinfo(3), t_listen(3), t_open(3), t_rcvconnect(3).

STANDARDS CONFORMANCE

t_optmgmt(): SVID2, XPG3, XPG4

t

NAME

t_rcv() - receive data or expedited data sent over a connection

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
/* or */
#include <tiuser.h>         /* for Transport Layer Interface - TLI  */

int t_rcv (fd, buf, nbytes, flags)
int fd;
char *buf;
unsigned nbytes;
int *flags;
```

DESCRIPTION

The **t_rcv()** function receives either normal or expedited data. *fd* identifies the local transport endpoint through which data will arrive. *buf* points to a receive buffer where user data will be placed. *nbytes* specifies the size of the receive buffer. *flags* may be set on return from **t_rcv()** and specifies optional flags as described below.

By default, **t_rcv()** operates in synchronous mode and will wait for data to arrive if none is currently available. However, if **O_NONBLOCK** is set (via **t_open()** or **fcntl()**), **t_rcv()** will execute in asynchronous mode and will fail if no data is available. (See [TNODATA] below.)

On return from the call, if T_MORE is set in *flags*, this indicates that there is more data. Thus, the current transport service data unit (TSDU) or expedited transport service data unit (ETSDU) must be received in multiple **t_rcv()** calls. In the asynchronous mode, the T_MORE flag may be set on return from the **t_rcv()** call even when the number of bytes received is less than the size of the the receive buffer specified. Each **t_rcv()** with the T_MORE flag set indicates that another **t_rcv()** must follow immediately to get more data for the current TSDU. The end of the TSDU is identified by the return of a **t_rcv()** call with the T_MORE flag not set. If the transport provider does not support the concept of a TSDU as indicated in the *info* argument on return from **t_open()** or **t_getinfo()**, the T_MORE flag is not meaningful and should be ignored. If *nbytes* is greater than zero on the call to **t_rcv()**, **t_rcv()** will return 0 only if the end of a TSDU is being returned to the user.

On return, the data returned is expedited data if T_EXPEDITED is set in *flags*. If the number of bytes of expedited data exceeds *nbytes*, **t_rcv()** will set T_EXPEDITED and T_MORE on return from the initial call. Subsequent calls to retrieve the remaining ETSDU will have T_EXPEDITED set on return. The end of the ETSDU is identified by the return of a **t_rcv()** call with the T_MORE flag not set.

If expedited data arrives after part of a TSDU has been retrieved, receipt of the remainder of the TSDU will be suspended until the ETSDU has been processed. Only after the full ETSDU has been retrieved (T_MORE not set) will the remainder of the TSDU be available to the user.

In synchronous mode, the only way for the user to be notified of the arrival of normal or expedited data is to issue this function or check for the T_DATA or T_EXDATA events using the **t_look()** function.

Thread-Safeness

The **t_rcv()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

XTI Internet Protocol-specific Information

The T_MORE flag should be ignored if normal data is delivered. If a byte in the data stream is pointed to by the TCP urgent pointer, as many bytes as possible preceding this marked byte and the marked byte itself are denoted as urgent data and are received with the T_EXPEDITED flag set. If the buffer supplied by the user is too small to hold all urgent data, the T_MORE flag will be set, indicating that urgent data still remains to be read. Note that the number of bytes received with the T_EXPEDITED flag set is not necessarily equal to the number of bytes sent by the peer user with T_EXPEDITED flag set.

RETURN VALUE

Upon successful completion, **t_rcv()** returns the number of bytes received. Otherwise, it return –1 and **t_errno** is set to indicate the error.

ERRORS

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

[TBADF]	The specified identifier does not refer to a transport endpoint.
[TNODATA]	**O_NONBLOCK** was set, but no data is currently available from the transport provider.
[TLOOK]	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TOUTSTATE]	(XTI only) The function was issued in the wrong sequence on the endpoint referenced by *fd*.
[TSYSERR]	A system error has occurred during execution of this function.
[TPROTO]	(XTI only) This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (**t_errno**).

SEE ALSO

fcntl(2), t_getinfo(3), t_look(3), t_open(3), t_snd(3).

STANDARDS CONFORMANCE

t_rcv(): SVID2, XPG3, XPG4

NAME

t_rcvconnect() - receive the confirmation from a connect request

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
/* or */
#include <tiuser.h>         /* for Transport Layer Interface - TLI  */

int t_rcvconnect (fd, call)
int fd;
struct t_call *call;
```

DESCRIPTION

The **t_rcvconnect()** function enables a calling transport user to determine the status of a previously sent connect request. **t_rcvconnect()** is also used in conjunction with **t_connect()** to establish a connection in asynchronous mode. The connection will be established on successful completion of this function.

fd identifies the local transport endpoint where communication will be established. *call* contains information associated with the newly established connection. *call* points to a **t_call** structure which contains the following members:

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

The type **netbuf** structure is defined in the **<xti.h>** or **<tiuser.h>** header file. This structure, which is used to define buffer parameters, has the following members:

unsigned int maxlen maximum byte length of the data buffer

unsigned int len actual byte length of data written to buffer

char *buf points to buffer location

In *call*, *addr* returns the protocol address associated with the responding transport endpoint. *opt* presents any protocol-specific information associated with the connection. For XTI over the OSI transport provider, struct **isoco_options** should be used. For TLI, see the documentation for the transport provider being used. *udata* points to optional user data that may be returned by the destination transport user during connection establishment. *sequence* has no meaning for this function.

The *maxlen* field of each argument must be set before issuing this function to indicate the maximum size of the buffer for each. However, *call* may be a null pointer, in which case no information is given to the user on return from **t_rcvconnect()**. By default, **t_rcvconnect()** executes in synchronous mode and waits for the connection to be established before returning. On return, the *addr*, *opt*, and *udata* fields reflect values associated with the connection.

If **O_NONBLOCK** is set (via **t_open()** or **fcntl()**), **t_rcvconnect()** executes in asynchronous mode, and reduces to a poll for existing connect confirmations. If none are available, **t_rcvconnect()** fails and returns immediately without waiting for the connection to be established. (See [TNODATA] below.) **t_rcvconnect()** must be re-issued at a later time to complete the connection establishment phase and retrieve the information returned in *call*.

Thread-Safeness

The **t_rcvconnect()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

Valid States

T_OUTCON

RETURN VALUE

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

ERRORS

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

[TBADF]	The specified identifier does not refer to a transport endpoint.
[TBUFOVFLW]	The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument and the connect information to be returned in *call* will be discarded. The provider's state, as seen by the user, will be changed to T_DATAXFER.
[TNODATA]	**O_NONBLOCK** was set, but a connect confirmation has not yet arrived.
[TLOOK]	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TOUTSTATE]	This function was issued in the wrong sequence on the transport endpoint referenced by *fd*.
[TSYSERR]	A system error has occurred during execution of this function.
[TPROTO]	(XTI only) This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no existing XTI (**t_errno**).

FILES

/usr/include/xti.h XTI data structure

/usr/include/tiuser.h TLI data structure

SEE ALSO

fcntl(2), t_accept(3), t_alloc(3), t_bind(3), t_connect(3), t_listen(3), t_open(3).

STANDARDS CONFORMANCE

t_rcvconnect(): SVID2, XPG3, XPG4

t

NAME

t_rcvdis() - retrieve information from disconnect

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
/* or */
#include <tiuser.h>         /* for Transport Layer Interface - TLI  */

int t_rcvdis (fd, discon)
int fd;
struct t_discon *discon;
```

DESCRIPTION

The **t_rcvdis()** function is used to identify the cause of a disconnect, and to retrieve any user data sent with the disconnect. *fd* identifies the local transport endpoint where the connection existed. *discon* points to a **t_discon** structure containing the following members:

```
struct netbuf udata;
int reason;
int sequence;
```

The type **netbuf** structure is defined in the **<xti.h>** or **<tiuser.h>** header file. This structure, which is used to define buffer parameters, has the following members:

unsigned int maxlen	maximum byte length of the data buffer
unsigned int len	actual byte length of data written to buffer
char *buf	points to buffer location

reason specifies the reason for the disconnect through a protocol-dependent reason code. For HP XTI over the OSI transport provider, these codes are described in the OTS/9000 manual section under "Transport Errors". For TLI, see the documentation for the transport provider being used. *udata* identifies any user data that was sent with the disconnect. *sequence* may identify an outstanding connect indication with which the disconnect is associated. *sequence* is only meaningful when **t_rcvdis()** is issued by a passive transport user who has executed one or more **t_listen()** functions and is processing the resulting connect indications. If a disconnect indication occurs, *sequence* can be used to identify which of the outstanding connect indications is associated with the disconnect.

If a user does not care that there is incoming data and does not need to know the value of *reason* or *sequence*, *discon* may be a null pointer and any user data associated with the disconnect will be discarded. However, if a user has retrieved more than one outstanding connect indication (via **t_listen()**) and *discon* is a null pointer, the user will be unable to identify with which connect indication the disconnect is associated.

Thread-Safeness

The **t_rcvdis()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

Valid States

T_DATAXFER, T_OUTCON, T_OUTREL, T_INREL, T_INCON (ocnt > 0)

RETURN VALUE

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

ERRORS

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

[TBADF]	The specified identifier does not refer to a transport endpoint.
[TNODIS]	No disconnect indication currently exists on the specified transport endpoint.
[TBUFOVFLW]	The number of bytes allocated for incoming data is not sufficient to store the data. If *fd* is a passive endpoint with *ocnt* (number of outstanding connections) > 1, it remains in state T_INCON (see **t_getstate**). Otherwise, the endpoint state is set to T_IDLE.

[TNOTSUPPORT]
This function is not supported by the underlying transport provider.

[TOUTSTATE] (XTI only) This function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

[TSYSERR] A system error has occurred during execution of this function.

[TPROTO] (XTI only) This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no existing XTI (**t_errno**).

SEE ALSO

t_alloc(3), t_connect(3), t_listen(3), t_open(3), t_snddis(3).

STANDARDS CONFORMANCE

t_rcvdis(): SVID2, XPG3, XPG4

t

NAME

t_rcvrel() - acknowledge receipt of an orderly release indication at a transport endpoint

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
/* or */
#include <tiuser.h>         /* for Transport Layer Interface - TLI  */

int t_rcvrel (fd)
int fd;
```

DESCRIPTION

The **t_rcvrel()** function is used in connection-oriented mode to acknowledge receipt of an orderly release indication at a transport endpoint. The released endpoint is specified by *fd*, which is a file descriptor previously returned by the **t_open()** function.

After receipt of this orderly release indication at the transport endpoint specified by *fd*, the transport user should not try to receive additional data from that transport endpoint. Any attempt to receive more data from a released transport endpoint blocks continuously. However, the transport user may continue to send data across the connection until a release is sent by the transport user by invoking the **t_sndrel()** function call.

The **t_rcvrel()** function should not be used unless the *servtype* type-of-service returned by the **t_open()** or **t_getinfo()** functions is T_COTS_ORD (supports connection-mode service with the optional orderly release facility).

Thread-Safeness

The **t_rcvrel()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

Valid States

T_DATAXFER, T_OUTREL

Note

HP OSI XTI does not support **t_rcvrel()**.

RETURN VALUE

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

ERRORS

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

[TBADF]	The specified identifier does not refer to a transport endpoint.
[TNOREL]	No orderly release indication currently exists on the specified transport endpoint.
[TBUFOVFLW]	The number of bytes allocated for incoming data is not sufficient to store the data.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TLOOK]	An asynchronous event has occurred on the transport endpoint referenced by *fd* and requires immediate attention.
[TSYSERR]	A system error has occurred during execution of this function.
[TPROTO]	(XTI only) This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (**t_errno**).
[TOUTSTATE]	(XTI only) The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

SEE ALSO

t_getinfo(3), t_open(3), t_sndrel(3).

STANDARDS CONFORMANCE

t_rcvrel(): SVID2, XPG3, XPG4

NAME

t_rcvudata() - receive a data unit from remote transport provider user

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
/* or */
#include <tiuser.h>         /* for Transport Layer Interface - TLI  */

int t_rcvudata (fd, unitdata, flags)
inf fd;
struct t_unitdata *unitdata;
int *flags;
```

DESCRIPTION

The **t_rcvudata()** function is used in connectionless-mode to receive a data unit from a remote transport provider user. The argument *fd* identifies the local transport endpoint through which data will be received. *unitdata* holds information associated with the received data unit. *flags* is set on return to indicate that the complete data unit was not received. The argument *unitdata* points to a **t_unitdata** structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

The type **netbuf** structure is defined in the **<xti.h>** or **<tiuser.h>** header file. This structure, which is used to define buffer parameters, has the following members:

unsigned int maxlen	maximum byte length of the data buffer
unsigned int len	actual byte length of data written to buffer
char *buf	points to buffer location

The *maxlen* field of *addr*, *opt* and *udata* must be set before calling this function to indicate the maximum size of the buffer for each.

On return from this call, *addr* specifies the protocol address of the sending user, *opt* identifies protocol-specific options that were associated with this data unit, and *udata* specifies the user data that was received.

By default, **t_rcvudata()** executes in the synchronous operating mode. The **t_rcvudata()** function waits for data to arrive at the transport endpoint specified by *fd* before returning control to the transport user who called this function. However, when the transport endpoint, specified by the *fd* parameter, has the **O_NONBLOCK** option set by **t_open()** or **fcntl()** function, the **t_rcvudata()** function executes in asynchronous mode. In asynchronous mode, when a data unit is unavailable, control is immediately returned to the caller.

If the buffer defined in the *udata* field of *unitdata* is not large enough to hold the current data unit, the buffer will be filled and T_MORE will be set in *flags* on return to indicate that another **t_rcvudata()** should be called to retrieve the rest of the data unit. Subsequent calls to **t_rcvudata()** will return zero for the length of the address and options until the full data unit has been received.

Thread-Safeness

The **t_rcvudata()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

Valid States

T_IDLE

RETURN VALUE

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

ERRORS

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

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

[TNODATA]	**O_NONBLOCK** was set, but no data units are currently available from the transport provider.
[TBUFOVFLW]	The number of bytes allocated for the incoming protocol address or protocol options is not sufficient to store the information. The unit data information normally returned in *unitdata* is discarded.
[TLOOK]	An asynchronous event has occurred on this transport endpoint required immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TOUTSTATE]	(XTI only) The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.
[TSYSERR]	A system error occurred during execution of this function.
[TPROTO]	(XTI only) This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (**t_errno**).

SEE ALSO

fcntl(2), t_alloc(3), t_open(3), t_optmgmt(3), t_rcvuderr(3), t_sndudata(3).

STANDARDS CONFORMANCE

t_rcvudata(): SVID2, XPG3, XPG4

t

NAME

t_rcvuderr() - receive a unit data error indication

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
/* or */
#include <tiuser.h>         /* for Transport Layer Interface - TLI  */

int t_rcvuderr (fd, uderr)
int fd;
struct t_uderr *uderr;
```

DESCRIPTION

The **t_rcvuderr()** function is used in connectionless mode to receive information concerning an error on a previously sent data unit. This function should only be issued following a unit data error indication. It informs the transport user that a data unit with a specific destination address and protocol options produced an error. The argument *fd* identifies the local transport endpoint through which the error report will be received. *uderr* points to a type **t_uderr** structure used to specify the protocol address, protocol options, and the nature of the error associated with the data unit sent through the transport endpoint specified by the *fd* parameter. The **t_uderr** structure has the following members:

```
struct netbuf addr;
struct netbuf opt;
t_scalar_t error;
```

The type **netbuf** structure is defined in the **<xti.h>** or **<tiuser.h>** header file. This structure, which is used to define buffer parameters, has the following members:

unsigned int maxlen	maximum byte length of the data buffer
unsigned int len	actual byte length of data written to buffer
char *buf	points to buffer location

The *maxlen* field of *addr* and *opt* must be set before calling this function to indicate the maximum size of the buffer for each.

On return from this call, the *addr* structure specifies the destination protocol address of the erroneous data unit the *opt* structure identified protocol-specific options that were associated with the data unit and *error* specifies a protocol dependent error code.

If the user does not care to identify the data unit that produced an error, *uderr* may be set to a null pointer, and **t_rcvuderr()** will simply clear the error indication without reporting any information to the user.

Thread-Safeness

The **t_rcvuderr()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

Valid States

T_IDLE

RETURN VALUE

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

ERRORS

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

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TNOUDERR]	No unit data error indication currently exists at the specified transport endpoint.
[TBUFOVFLW]	The number of bytes allocated for the incoming protocol address or options information is not sufficient to store that information. The unit data error information to be returned in *uderr* will be discarded.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.

[TSYSERR] A system error occurred during execution of this function.

[TPROTO] (XTI only) This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (**t_errno**).

SEE ALSO

t_look(3), t_rcvudata(3), t_sndudata(3).

STANDARDS CONFORMANCE

t_rcvuderr(): SVID2, XPG3, XPG4

NAME

t_snd() - send data or expedited data over a connection

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
/* or */
#include <tiuser.h>         /* for Transport Layer Interface - TLI  */

int t_snd (fd, buf, nbytes, flags)
int fd;
char *buf;
unsigned nbytes;
int flags;
```

DESCRIPTION

This function is used to send either normal or expedited data. *fd* identifies the local transport endpoint over which data should be sent, *buf* points to the user data, *nbytes* specifies the number of bytes of user data to be sent, and *flags* specifies any optional flags described below:

T_EXPEDITED
if set in *flags*, the data will be sent as expedited data and will be subject to the interpretations of the transport provider.

T_MORE if set in *flags*, this indicates to the transport provider that the transport service data unit, TSDU (or expedited transport service data unit - ETSDU), is being sent through multiple **t_snd()** calls. Each **t_snd()** with the T_MORE flag set indicates that another **t_snd()** will follow with more data for the current TSDU. The end of the TSDU (or ETSDU) is identified by a **t_snd()** call with the T_MORE flag not set. Use of T_MORE enables a user to break up large logical data units without losing the boundaries of those units at the other end of the connection. The flag implies nothing about how the data is packaged for transfer below the transport interface. If the transport provider does not support the concept of a TSDU as indicated in the *info* argument on return from **t_open()** or **t_getinfo()**, the T_MORE flag is not meaningful and should be ignored.

The sending of a zero-length fragment of a TSDU or ETSDU is only permitted where this is used to indicate the end of a TSDU or ETSDU, i.e. when the T_MORE flag is not set. Some transport providers also forbid zero-length TSDUs and ETSDUs.

By default, **t_snd()** operates in synchronous mode and may wait if flow control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if **O_NONBLOCK** is set (via **t_open()** or **fcntl()**), **t_snd()** will execute in asynchronous mode, and will fail immediately if there are flow control restrictions. For XTI only, the process can arrange to be informed when the flow control restrictions are cleared via **t_look()**.

t_snd() will wait if STREAMS internal resources are not available, even if **O_NONBLOCK** is set. **O_NONBLOCK** non-blocking behavior applies only to flow control conditions.

On successful completion, **t_snd()** returns the number of bytes accepted by the transport provider. In synchronous mode, this will equal the number of bytes specified in *nbytes*. However, if **O_NONBLOCK** (asynchronous mode) is set, it is possible that only part of the data will actually be accepted by the transport provider. In this case, **t_snd()** will set T_MORE for the data that can be accepted by the provider, and return a value that is less than the value of *nbytes*.

If *nbytes* is zero and sending of zero octets is not supported by the underlying transport service, **t_snd()** will return –1 with **t_errno** set to [TBADDATA].

The size of each TSDU or ETSDU must not exceed the limits of the transport provider as returned in the TSDU or ETSDU fields of the *info* argument of **t_open()** or **t_getinfo()**. Failure to comply will result in a protocol error. (See [TSYSERR] below.)

For XTI only, the error [TLOOK] may be returned to inform the process that an event (e.g., a disconnect) has occurred.

For TLI only, if the transport endpoint is in any state but T_DATAXFER or T_INREL, the transport provider will set **t_errno** to [TSYSERR] and set the system **errno** to [EPROTO].

Thread-Safeness

The **t_snd()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

Caveats

It is important to remember that the transport provider treats all users of a transport endpoint as a single user. Therefore, if several processes issue concurrent **t_snd()** calls then the different data may be intermixed.

Multiple sends which exceed the maximum TSDU or ETSDU size may not be discovered by XTI. In this case, an implementation-dependent error will result (generated by the transport provider) perhaps on a subsequent XTI call. This error may take the form of a connection abort, [TSYSERR], [TBADDATA], or a [TPROTO].

If multiple sends which exceed the maximum TSDU or ETSDU size are detected by XTI, **t_snd()** fails with [TBADDATA].

RETURN VALUE

On successful completion, **t_snd()** returns the number of bytes accepted by the transport provider. Otherwise, –1 is returned on failure and **t_errno** is set to indicate the error.

ERRORS

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

[TBADF]	The specified identifier does not refer to a transport endpoint.
[TBADDATA]	Illegal amount of data: A single send was attempted specifying a TSDU (ETSDU) or fragment TSDU (ETSDU) greater than that specified by the current values of the TSDU or ETSDU fields in the info argument. A send of zero byte TSDU (ETSDU) or zero byte fragment of TSDU (ETSDU) is not supported by the provider. Multiple sends were attempted resulting in a TSDU (ETSDU) larger than that specified by the current value of the TSDU or ETSDU fields in the *info* argument. The ability of XTI to catch such an error case is implementation-dependent.
[TBADFLAG]	An invalid flag was specified.
[TFLOW]	**O_NONBLOCK** was set, but the flow control mechanism prevented the transport provider from accepting all or part of the data at this time.
[TLOOK]	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TOUTSTATE]	The function was issued in the wrong sequence on the endpoint referenced by *fd*.
[TSYSERR]	A system error has occurred during execution of this function.
[TPROTO]	(XTI only) This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (**t_errno**).

SEE ALSO

t_getinfo(3), t_open(3), t_rcv(3).

STANDARDS CONFORMANCE

t_snd(): SVID2, XPG3, XPG4

NAME

t_snddis() - send user-initiated disconnect request

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
/* or */
#include <tiuser.h>         /* for Transport Layer Interface - TLI  */

int t_snddis (fd, call)
int fd;
struct t_call *call;
```

DESCRIPTION

The **t_snddis()** function is used to initiate an abortive release on an already established connection or to reject a connect request. *fd* identifies the local transport endpoint of the connection, and *call* specifies information associated with the abortive release. *call* points to a **t_call** structure which contains the following members:

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

The type **netbuf** structure is defined in the **<xti.h>** or **<tiuser.h>** header file. This structure, which is used to define buffer parameters, has the following members:

unsigned int maxlen	maximum byte length of the data buffer
unsigned int len	actual byte length of data written to buffer
char *buf	points to buffer location

The values in *call* have different semantics, depending on the context of the call to **t_snddis()**. When rejecting a connect request, *call* must be a non-null pointer and contain a valid value of *sequence* to uniquely identify the rejected connect indication to the transport provider. The *sequence* parameter is only meaningful, if the transport connection is in the T_INCON state (see *t_getstate*(3)). The *addr* and *opt* fields of *call* are ignored. In all other cases, *call* need only be used when data is being sent with the disconnect request. The *addr*, *opt*, and *sequence* fields of the **t_call** structure are ignored. If the user does not wish to send data to the remote user, the value of *call* may be a null pointer.

udata specifies the user data to be sent to the remote user. The amount of user data must not exceed the limits supported by the transport provider as returned in the *discon* field of the *info* argument of **t_open()** or **t_getinfo()**. If the *len* field of *udata* is zero, no data will be sent to the remote user.

Thread-Safeness

The **t_snddis()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

Valid States

T_DATAXFER, T_OUTCON, T_OUTREL, T_INREL, T_INCON (ocnt > 0)

Caveats

t_snddis() is an abortive disconnect. Therefore a **t_snddis()** issued on a connection endpoint may cause data previously sent via **t_snd()** or data not yet received to be lost (even if an error is returned).

RETURN VALUE

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

ERRORS

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

[TBADF]	The specified identifier does not refer to a transport endpoint.
[TOUTSTATE]	This function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

[TBADDATA]	(XTI only) The amount of user data specified was not within the bounds allowed by the transport provider. Some outbound data queued for this endpoint may be lost.
[TBADSEQ]	An invalid sequence was specified, or a null *call* pointer was specified when rejecting a connect request. Some outbound data queued for this endpoint may be lost.
[TLOOK]	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.
[TPROTO]	(XTI only) This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (**t_errno**).

SEE ALSO

t_connect(3), t_getinfo(3), t_listen(3), t_open(3).

STANDARDS CONFORMANCE

t_snddis(): SVID2, XPG3, XPG4

t

NAME

t_sndrel() - initiate an orderly release

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
/* or */
#include <tiuser.h>         /* for Transport Layer Interface - TLI  */

int t_sndrel (fd)
int fd;
```

DESCRIPTION

The **t_sndrel()** function is used in connection-oriented mode to initiate an orderly release at a transport endpoint specified by *fd*, which is a file descriptor previously returned by the **t_open()** function.

After this orderly release is indicated, the transport user should not try to send more data through that transport endpoint. An attempt to send more data to a released transport endpoint may block continuously. However, the transport user may continue to receive data over the connection until an orderly release indication is received. This function is an optional service of the transport provider and is only supported if the transport provider returned service type **T_COTS_ORD** on **t_open()** or **t_getinfo()**.

Thread-Safeness

The **t_sndrel()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

Note

HP OSI XTI does not support **t_sndrel()**.

RETURN VALUE

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

ERRORS

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

[TBADF]	The specified identifier does not refer to a transport endpoint.
[TFLOW]	Asynchronous mode is indicated because **O_NONBLOCK** was set, but the transport provider cannot accept a release because of flow-control restrictions.
[TLOOK]	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.
[TPROTO]	(XTI only) This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (**t_errno**).

SEE ALSO

t_getinfo(3), t_open(3), t_rcvrel(3).

STANDARDS CONFORMANCE

t_sndrel(): SVID2, XPG3, XPG4

NAME

t_sndudata() - send a data unit

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
/* or */
#include <tiuser.h>         /* for Transport Layer Interface - TLI )

int t_sndudata(fd, unitdata)
int fd;
struct t_unitdata *unitdata;
```

DESCRIPTION

The **t_sndudata()** function is used in connectionless mode to send a data unit to another transport user. The argument *fd* identifies the local transport endpoint through which data is sent. The argument *unitdata* points to a type **t_unitdata** structure used to specify a data unit being sent through the transport endpoint specified by the *fd* parameter. The **t_unitdata** structure has the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

The type netbuf structure is defined in the **<xti.h>** or **<tiuser.h>** header file. This structure, which is used to define buffer parameters, has the following members:

unsigned int maxlen	maximum byte length of the data buffer
unsigned int len	actual byte length of data written to buffer
char *buf	points to buffer location

In *unitdata*, *addr* specifies the protocol address of the destination user, *opt* identifies protocol-specific options that the user wants associated with this request and *udata* specifies the user data to be sent. The user may choose not to specify what protocol options are associated with the transfer by setting the *len* field of *opt* to zero. In this case, the provider may use default options.

If the *len* field of *udata* is zero, and sending of zero octets is not supported by the underlying transport service, the **t_sndudata()** will return –1 with **t_errno** set to [TBADDATA].

By default, **t_sndudata()** executes in the synchronous operating mode and may wait if flow control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if **O_NONBLOCK** is set (via **t_open()** or **fcntl()**), **t_sndudata()** will execute in asynchronous mode and will fail under such conditions. The process can arrange to be notified of the clearance restriction via either **t_look()** or the EM interface.

If the amount of data specified in *udata* exceeds the TSDU size as returned in the *tsdu* field of the *info* argument of **t_open()** or **t_getinfo()**, a [TBADDATA] error will be generated. If **t_sndudata()** is called before the destination user has activated its transport endpoint, the data unit may be discarded.

If it is not possible for the transport provider to immediately detect the conditions that cause errors [TBADADDR] and [TBADOPT], then these errors will alternatively be returned by **t_rcvuderr()**. Therefore, an application must be prepared to receive these errors both of these ways.

Thread-Safeness

The **t_sndudata()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

RETURN VALUE

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

ERRORS

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

[TBADDATA]	Illegal amount of data. Zero octets is not supported.
[TBADF]	The specified file descriptor does not refer to a transport endpoint.

[TFLOW]	Asynchronous mode is indicated because **O_NONBLOCK** was set, but the transport provider cannot accept the data because of flow-control restrictions.
[TLOOK]	(XTI only) An asynchronous event has occurred on this transport endpoint and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TOUTSTATE]	(XTI only) The **t_sndudata()** function was issued in the wrong sequence on the transport endpoint referenced by the *fd* parameter.
[TSYSERR]	A system error occurred during execution of this function. A protocol error may not cause the **t_sndudata()** function to fail until a subsequent call is made to access the transport endpoint specified by the *fd* parameter.
[TBADADDR]	The specified protocol address was in an incorrect format or contained illegal information.
[TBADOPT]	The specified options were in an incorrect format or contained illegal information.
[TPROTO]	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (**t_errno**).

SEE ALSO

fcntl(2), t_alloc(3), t_open(3), t_rcvudata(3), t_rcvuderr(3).

STANDARDS CONFORMANCE

t_sndudata(): SVID2, XPG3, XPG4

NAME

t_strerror() - produce an error message string

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
int *t_strerror (errnum)
int errnum;
struct info *info;
```

DESCRIPTION

The **t_strerror()** function maps the error number in *errnum* that corresponds to an XTI error to a language-dependent error message string and returns a pointer to the string. The string pointed to will not be modified by the program, but may be overwritten by a subsequent call to the **t_strerror()** function. The string is not terminated by a newline character. The language for error message strings written by **t_strerror()** is implementation-defined. If it is English, the error message string describing the value in **t_errno** is identical to the comments following the **t_errno** defined in **<xti.h>**. If an error code is unknown, and the language is English, **t_strerror()** returns the string:

"*error*: error unknown"

where *error* is the error number supplied as input. In other languages, an equivalent text is provided.

Thread-Safeness

The **t_strerror()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

Valid Status

All - apart from T_UNINIT

RETURN VALUE

The function **t_strerror()** returns a pointer to the generated message string.

SEE ALSO

t_error(3).

t

NAME

t_sync() - synchronize transport library

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
/* or */
#include <tiuser.h>         /* for Transport Layer Interface - TLI  */

int t_sync (fd)
int fd;
```

DESCRIPTION

For the transport endpoint specified by *fd*, the **t_sync()** function synchronizes the data structures managed by the transport library with information from the underlying transport provider. In doing so, it can convert an uninitialized file descriptor (obtained via **open()**, **dup()** or as a result of a **fork()** and **exec()**) to an initialized endpoint, assuming that the file descriptor referenced a transport endpoint, by updating and allocating the necessary library data structures. This function also allows two cooperating processes to synchronize their interaction with a transport provider.

For example, if a process forks a new process and issues an **exec()**, the new process must issue a **t_sync()** to build the private library data structure associated with a transport endpoint and to synchronize the data structure with the relevant provider information.

It is important to remember that the transport provider treats all users of a transport endpoint as a single user. If multiple processes are using the same endpoint, they should coordinate their activities so as not to violate the state of the transport endpoint. The function **t_sync()** returns the current state of the transport endpoint to the user, thereby enabling the user to verify the state before taking further action. This coordination is only valid among cooperating processes; it is possible that a process or an incoming event could change the endpoint's state after a **t_sync()** is issued.

If the transport endpoint is undergoing a state transition when **t_sync()** is called, the function will fail.

Thread-Safeness

The **t_sync()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

Valid States

All - apart from T_UNINIT

RETURN VALUE

t_sync returns the state of the transport connection endpoint on successful completion and –1 on failure, and **t_errno** is set to indicate the error. The state returned is one of the following:

T_UNBND	Unbound
T_IDLE	Idle
T_OUTCON	Outgoing connection pending
T_INCON	Incoming connection pending
T_DATAXFER	Data transfer
T_OUTREL	Outgoing orderly release (waiting for an orderly release indication)
T_INREL	Incoming orderly release (waiting for an orderly release request)

ERRORS

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

[TBADF]	The specified file descriptor does not refer to a transport endpoint. This error may be returned when the *fd* has been previously closed or an erroneous number may have been passed to the call.
[TSTATECHNG]	The transport endpoint is undergoing a state change.
[TSYSERR]	A system error has occurred during execution of this function.
[TPROTO]	(XTI only) This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (**t_errno**).

SEE ALSO
t_open(3), t_getstate(3), dup(2), exec(2), fork(2), open(2).

STANDARDS CONFORMANCE
t_sync(): SVID2, XPG3, XPG4

NAME

t_unbind() - disable a transport endpoint

SYNOPSIS

```
#include <xti.h>            /* for X/OPEN Transport Interface - XTI */
/* or */
#include <tiuser.h>         /* for Transport Layer Interface - TLI  */

int t_unbind (fd)
int fd;
```

DESCRIPTION

The **t_unbind()** function disables the transport endpoint specified by *fd* which was previously bound by **t_bind()**. On completion of this call, no further data or events destined for this transport endpoint will be accepted by the transport provider.

Thread-Safeness

The **t_unbind()** function is safe to be called by multithreaded applications, and it is thread-safe for both POSIX Threads and DCE User Threads. It has a cancellation point. It is neither async-cancel safe nor async-signal safe. Finally, it is not fork-safe.

Note

Users can access XTI versions of the **t_*** routines by linking with **/usr/lib/libxti.a**. For more information on XTI, see *HP-UX/9000 XTI Programmer's Guide*.

TLI supports any transport provider which is compliant with TPI (Transport Provider Interface). Users can access TLI versions of the t_* routines by linking with **/usr/lib/libnsl_s.a**. For more information on TLI, see the TLI section of the *STREAMS/UX for HP 9000 Reference Manual*.

Valid States

T_IDLE

RETURN VALUE

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

ERRORS

On failure, **t_errno** is set to the following:

[TBADF]	The specified identifier does not refer to a transport endpoint.
[TOUTSTATE]	The function was issued in the wrong sequence.
[TLOOK]	An asynchronous event has occurred on this transport endpoint.
[TSYSERR]	A system error has occurred during execution of this function.

SEE ALSO

t_bind(3).

STANDARDS CONFORMANCE

t_unbind(): SVID2, XPG3, XPG4

NAME

tan(), tanf() - tangent functions

SYNOPSIS

```
#include <math.h>
double tan(double x);
float tanf(float x);
```

DESCRIPTION

tan() returns the tangent of *x* (*x* specified in radians).

tan() may lose accuracy when *x* is far from zero.

tanf() is a **float** version of **tan()**; it takes a **float** argument and returns a **float** result. To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options.

tanf() is not specified by any standard, but it is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, make sure your program includes <**math.h**>, and link in the math library by specifying **-lm** on the compiler or linker command line.

Millicode versions of the **tan()** and **tanf()** functions are available. Millicode versions of math library functions are usually faster than their counterparts in the standard library. To use these versions, compile your program with the **+Olibcalls** or the **+Oaggressive** optimization option.

If an error occurs, the millicode versions return the value described in the *RETURN VALUE* section, but do not set **errno**.

For more information, see the *HP-UX Floating-Point Guide*.

RETURN VALUE

If *x* is NaN or ±INFINITY, **tan()** returns NaN.

If the correct value after rounding would be smaller in magnitude than **MINDOUBLE**, **tan()** returns zero.

If the correct value would overflow, **tan()** returns ±**HUGE_VAL** and sets **errno** to [ERANGE].

ERRORS

If **tan()** fails, **errno** is set to the following value.

[ERANGE] The correct value would overflow.

SEE ALSO

acos(3M), asin(3M), atan(3M), atan2(3M), cos(3M), sin(3M), tand(3M), math(5), values(5).

STANDARDS CONFORMANCE

tan(): SVID3, XPG4.2, ANSI C

t

NAME

tand(), tandf() - degree-valued tangent functions

SYNOPSIS

```
#include <math.h>
double tand(double x);
float tandf(float x);
```

DESCRIPTION

tand() is a degree-valued version of the **tan()** function. It returns the tangent of *x* (*x* specified in degrees).

tand() may lose accuracy when *x* is far from zero.

tandf() is a **float** version of **tand()**; it takes a **float** argument and returns a **float** result.

tand() and **tandf()** are not specified by any standard, but **tandf()** is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

If *x* is NaN or ±INFINITY, **tand()** returns NaN.

If the correct value after rounding would be smaller in magnitude than **MINDOUBLE**, **tand()** returns zero.

If the correct value would overflow, **tand()** returns ±**HUGE_VAL** and sets **errno** to [ERANGE].

ERRORS

If **tand()** fails, **errno** is set to the following value.

[ERANGE] The correct value would overflow.

SEE ALSO

acosd(3M), asind(3M), atand(3M), atan2d(3M), cosd(3M), sind(3M), tan(3M), math(5), values(5).

t

NAME

tanh(), tanhf() - hyperbolic tangent functions

SYNOPSIS

```
#include <math.h>
double tanh(double x);
float tanhf(float x);
```

DESCRIPTION

The **tanh()** function returns the hyperbolic tangent of its argument.

tanhf() is a **float** version of **tanh()**; it takes a **float** argument and returns a **float** result. To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options.

tanhf() is not specified by any standard, but it is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, make sure your program includes <**math.h**>, and link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

If *x* is ±INFINITY, **tanh()** returns ±1.0 respectively.

If *x* is NaN, **tanh()** returns NaN.

If the correct value after rounding would be smaller in magnitude than **MINDOUBLE**, **tanh()** returns zero.

ERRORS

No errors are defined.

SEE ALSO

atanh(3M), cosh(3M), sinh(3M), math(5), values(5).

STANDARDS CONFORMANCE

tanh(): SVID3, XPG4.2, ANSI C

t

NAME

tcgetattr(), tcsetattr() - control tty device

SYNOPSIS

```
#include <termios.h>

int tcgetattr(int fildes, struct termios *termios_p);

int tcsetattr(
     int fildes,
     int optional_actions,
     const struct termios *termios_p
);
```

DESCRIPTION

tcgetattr() gets the parameters associated with *fildes* and stores them in the *termios* structure referenced by *termios_p*. If the terminal device does not support split baud rates, the input baud rate stored in the *termios* structure is zero. This function is allowed from a background process (see *termio*(7)). However, the terminal attributes can be subsequently changed by a foreground process.

tcsetattr() sets the parameters associated with *fildes* (unless support is required from underlying hardware that is not available) from the *termios* structure referenced by *termios_p* as follows:

- If *optional_actions* is **TCSANOW**, the change is immediate.
- If *optional_actions* is **TCSADRAIN**, the change occurs after all output written to *fildes* is transmitted.
- If *optional_actions* is **TCSAFLUSH**, the change occurs after all output written to *fildes* is transmitted, and all input that has been received but not read is discarded.

APPLICATION USAGE

tcgetattr() and **tcsetattr()** are thread-safe and async-cancel-safe.

RETURN VALUE

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

ERRORS

tcgetattr() and **tcsetattr()** fail if any of the following conditions are encountered:

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

[EINVAL] The *optional_actions* argument is not a proper value or no part of a **tcsetattr()** request could be performed.

[ENOTTY] The file associated with *fildes* is not a terminal.

WARNINGS

tcsetattr() will attempt to perform as much of the request as possible. However, the hardware being used may not support all possible c_cflag values. If any part of the request can be performed, those values will be set, any values not supported by the hardware will be ignored, and **tcsetattr()** will complete successfully. If no part of the request can be performed, **tcsetattr()** will fail and set **errno** to [EINVAL].

For any hardware that does not support separate input and output baud rates, an attempt to set different input and output baud rates does not affect either baud rate.

tcgetattr() always returns the actual values set in hardware. For any hardware that does not support separate input and output baud rates, **tcgetattr()** returns zero for the input baud rate.

SEE ALSO

cfspeed(3C), tccontrol(3C), termio(7).

STANDARDS CONFORMANCE

tcgetattr(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

tcsetattr(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

tcsendbreak(), tcdrain(), tcflush(), tcflow() - tty line control functions

SYNOPSIS

```
#include <termios.h>
int tcsendbreak(int fildes, int duration);
int tcdrain(int fildes);
int tcflush(int fildes, int queue_selector);
int tcflow(int fildes, int action);
```

DESCRIPTION

If the terminal is using asynchronous serial data transmission, **tcsendbreak()** causes transmission of a continuous stream of zero-valued bits for a specific duration. If *duration* is zero, it causes transmission of zero-valued bits for at least 0.25 seconds, but not more than 0.5 seconds. If *duration* is not zero, zero-valued bits are not transmitted. For all HP-UX implementations, *duration* is ignored.

tcdrain() waits until all output written to *fildes* has been transmitted.

tcflush() discards data written to *fildes* but not transmitted, or data received but not read, depending on the value of *queue_selector*:

- If *queue_selector* is **TCIFLUSH**, data received but not read is flushed.
- If *queue_selector* is **TCOFLUSH**, data written but not transmitted is flushed.
- If *queue_selector* is **TCIOFLUSH**, both data received but not read, and data written but not transmitted is flushed.

tcflow() suspends transmission of data to *fildes* or reception of data from *fildes*, depending on the value of *action*:

- If *action* is **TCOOFF**, output is suspended.
- If *action* is **TCOON**, suspended output is restarted.
- If *action* is **TCIOFF**, a STOP character is transmitted which is intended to cause the terminal to stop transmitting data to the system.
- If *action* is **TCION**, a START character is transmitted which is intended to cause the terminal to start transmitting data to the system.

APPLICATION USAGE

tcsendbreak(), **tcdrain()**, **tcflush()** and **tcflow()** are thread-safe. These interfaces are async-cancel-safe. A cancellation point may occur when a thread is executing **tcdrain()**.

RETURN VALUE

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

ERRORS

These functions fail if one or more of the following is true:

[EBADF]	*fildes* is not a valid file descriptor.
[EINTR]	A signal was received during **tcdrain()**.
[EINVAL]	The *queue_selector* or the *action* argument is not a proper value.
[ENOTTY]	The file associated with *fildes* is not a terminal.

SEE ALSO

tcattribute(3C), tccontrol(3C), termio(7).

STANDARDS CONFORMANCE

tcdrain(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

tcflow(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

tcflush(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1
tcsendbreak(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

t

NAME

tcgetpgrp( ) - get foreground process group id

SYNOPSIS

```
#include <unistd.h>
pid_t tcgetpgrp(int fildes);
```

DESCRIPTION

tcgetpgrp() returns the value of the process group ID of the foreground process group associated with the terminal referenced by *fildes*. **tcgetpgrp()** is allowed from a process that is a member of a background process group (see *termio*(7)); however, the information can be subsequently changed by a process that is a member of a foreground process group.

RETURN VALUE

Upon successful completion, **tcgetpgrp()** returns the value of the process group ID of the foreground process group associated with the terminal referenced by *fildes*. Otherwise, **tcgetpgrp()** returns a value of –1 and sets **errno** to indicate the error.

ERRORS

tcgetpgrp() fails if any of the following conditions are encountered:

[EACCES]	The file associated with *fildes* is the controlling terminal of the calling process, however, there is no foreground process group defined for the controlling terminal.
[EBADF]	*fildes* is not a valid file descriptor.
[ENOTTY]	The file associated with *fildes* is not the controlling terminal or the calling process does not have a controlling terminal.

APPLICATION USAGE

tcgetpgrp() is thread-safe and async-cancel-safe.

WARNING

The error EACCES, which is returned if the controlling terminal has no foreground process group, might not be returned in future releases, depending on the course taken by the POSIX standard. Portable applications therefore should not rely on this error condition.

SEE ALSO

setpgid(2), setsid(2), tcsetpgrp(3C), termio(7).

STANDARDS CONFORMANCE

tcgetpgrp(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

tcgetsid() - get terminal session ID

SYNOPSIS

```
#include <termios.h>

pid_t tcgetsid (int fildes);
```

DESCRIPTION

The **tcgetsid()** function returns the value of the session ID of the foreground process associated with the terminal referenced by *fildes*. **tcgetsid()** is allowed from a process that is a member of a background process group (see *termio*(7)).

RETURN VALUE

Upon successful completion, **tcgetsid()** returns the value of the session ID of the foreground process associated with the terminal referenced by *fildes*. Otherwise, **tcgetsid()** returns a value of **−1** and sets **errno** to indicate the error.

ERRORS

If the **tcgetsid()** function fails, it sets **errno** (see *errno*(2)) to one of the following values:

[EACCES] The file associated with *fildes* is the controlling terminal of the calling process; however, there is no foreground process group defined for the controlling terminal.

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

[ENOTTY] The file associated with *fildes* is not the controlling terminal or the calling process does not have a controlling terminal.

APPLICATION USAGE

tcgetsid() is thread-safe and async-cancel-safe.

SEE ALSO

getsid(2), setsid(2), tcgetpgrp(3C).

t

NAME

tcsetpgrp() - set foreground process group id

SYNOPSIS

```
#include <unistd.h>
int tcsetpgrp(int fildes, pid_t pgrp_id);
```

DESCRIPTION

If the calling process has a controlling terminal, **tcsetpgrp()** sets the foreground process group ID associated with the terminal referenced by *fildes* to *pgrp_id*. The file associated with *fildes* must be the controlling terminal of the calling process and the controlling terminal must be currently associated with the session of the calling process. The value of *pgrp_id* must match a process group ID of a process in the same session as the calling process.

RETURN VALUE

Upon successful completion, **tcsetpgrp()** returns zero. Otherwise, **tcsetpgrp()** returns –1 and sets **errno** to indicate the error.

ERRORS

tcsetpgrp() fails if any of the following conditions are encountered:

[EBADF]	*fildes* is not a valid file descriptor.
[EINVAL]	The value of the *pgrp_id* argument is not supported.
[ENOTTY]	The calling process does not have a controlling terminal, or the *fildes* is not the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.
[EPERM]	The value of *pgrp_id* is a supported value but does not match the process group ID of a process in the same session as the calling process.

APPLICATION USAGE

tcsetpgrp() is thread-safe and async-cancel-safe.

SEE ALSO

setpgid(2), setsid(2), tcgetpgrp(3C), termio(7).

STANDARDS CONFORMANCE

tcsetpgrp(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

t

(ENHANCED CURSES)

NAME

termattrs, term_attrs — get supported terminal video attributes

SYNOPSIS

```
#include <curses.h>
chtype termattrs(void);
attr_t term_attrs(void);
```

DESCRIPTION

The **termattrs()** function extracts the video attributes of the current terminal. That are supported by the **chtype** data type.

The **term_attrs()** function extracts the video attributes of the current terminal. That are supported for a **cchar_t** data type.

RETURN VALUE

The **termattrs()** function returns a logical OR of A_ values of all video attributes supported by the terminal.

The **term_attrs()** function returns a logical OR of WA_ values of all video attributes supported by the terminal.

ERRORS

No errors are defined.

SEE ALSO

Attributes in curses_intro(3X), attroff(3X), attr_get(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

t

NAME

tgetent(), tgetnum(), tgetflag(), tgetstr(), tgoto(), tputs() - emulate /usr/share/lib/termcap access routines

SYNOPSIS

```
#include <curses.h>
int tgetent(char *bp, const char *name);
int tgetnum(const char *id);
int tgetflag(const char *id);
char *tgetstr(const char *id, char **area);
char *tgoto(char *cm, int destcol, int destline);
int tputs(char *cp, int affcnt, int (*outc)(int));
```

DESCRIPTION

These functions extract and use capabilities from the compiled terminal capability data bases (see *terminfo*(4)). They are emulation routines that are provided as a part of the *curses*(3X) library.

tgetent() Extracts the compiled entry for terminal *name* into buffers accessible by the programmer. Unlike previous termcap routines, all capability strings (except cursor addressing and padding information) are already compiled and stored internally upon return from **tgetent()**. The buffer pointer *bp* is redundant in the emulation, and is ignored. It should not be relied upon to point to meaningful information. **tgetent()** returns -1 if it cannot access the *terminfo* directory, 0 if there is no capability file for *name*, and 1 if all goes well. If a **TERMINFO** environment variable is set, **tgetent()** first looks for **TERMINFO/?/***name* (where **?** is the first character of *name*), and if that file is not accessible, it looks for **/usr/share/lib/terminfo/?/***name*.

tgetnum() Gets the numeric value of capability *id*, returning -1 if it is not given for the terminal. **tgetnum()** is useful only with capabilities having numeric values.

tgetflag() Returns 1 if the specified capability is present in the terminal's entry, and 0 if it is not. **tgetflag()** is useful only with capabilities that are boolean in nature (i.e. either present or missing in *terminfo*(4)).

tgetstr() Returns a pointer to the string value of capability *id*. In addition, if *area* is not a NULL pointer, **tgetstr()** places the capability in the buffer at *area* and advances the area pointer. The returned string capability is compiled except for cursor addressing and padding information. **tgetstr()** is useful only with capabilities having string values.

tgoto() Returns a cursor addressing string decoded from *cm* to go to column *destcol* in line *destline*. (Programs that call **tgoto()** should be sure to turn off the TAB3 bit or bits, since **tgoto()** can now output a tab. See *termio*(7)). Note that programs using **termcap** should in general turn off TAB3 anyway since some terminals use Ctrl-I for other functions, such as nondestructive space.) If a **%** sequence is given that is not understood, **tgoto()** returns [OOPS].

tputs() Decodes the padding information of the string *cp*. *affcnt* gives the number of lines affected by the operation, or 1 if this is not applicable. *outc* is a routine that is called with each character in turn. The **terminfo** variable *pad_char* should contain a pad character to be used (from the *pc* capability) if a null (^@) is inappropriate.

WARNINGS

These routines are not meant to be used by programs running in the background.

FILES

/usr/lib/libcurses.a -lcurses library
/usr/share/lib/terminfo/?/* data bases

SEE ALSO

ex(1), terminfo(4), termio(7).

(ENHANCED CURSES)

NAME

termname — get terminal name

SYNOPSIS

```
#include <curses.h>
char *termname(void);
```

DESCRIPTION

The **termname()** function obtains the terminal name as recorded by **setupterm()**.

RETURN VALUE

The **termname()** function returns a pointer to the terminal name.

ERRORS

No errors are defined.

SEE ALSO

Minimum Guaranteed Limits in terminfo(4), del_curterm(3X), getenv(3C) (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification), initscr(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

t

(ENHANCED CURSES)

NAME

tgetent, tgetflag, tgetnum, tgetstr, tgoto — **termcap** database emulation (**TO BE WITHDRAWN**)

SYNOPSIS

```
#include <term.h>
int tgetent(char *bp, const char *name);
int tgetflag(char id[2]);
int tgetnum(char id[2]);
char *tgetstr(char id[2], char **area);
char *tgoto(char *cap, int col, int row);
```

DESCRIPTION

The **tgetent()** function looks up the **termcap** entry for *name*. The emulation ignores the buffer pointer *bp*.

The **tgetflag()** function gets the boolean entry for *id*.

The **tgetnum()** function gets the numeric entry for *id*.

The **tgetstr()** function gets the string entry for *id*. If *area* is not a null pointer and does not point to a null pointer, **tgetstr()** copies the string entry into the buffer pointed to by **area* and advances the variable pointed to by *area* to the first byte after the copy of the string entry.

The **tgoto()** function instantiates *col* and *row* into *cap* and returns a pointer to the resulting string.

All of the information available in the **terminfo** database need not be available through these functions.

RETURN VALUE

Upon successful completion, functions that return an integer return OK. Otherwise, they return ERR.

Functions that return pointers return a null pointer on error.

ERRORS

No errors are defined.

APPLICATION USAGE

These functions are included as a conversion aid for programs that use the **termcap** library. Their arguments are the same and the functions are emulated using the **terminfo** database.

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the A_ prefix.

Any terminal capabilities from the **terminfo** database that cannot be retrieved using these interfaces can be retrieved using the interfaces described on the **tigetflg()** page.

Portable applications must use **tputs()** to output the strings returned by **tgetstr()** and **tgoto()**.

SEE ALSO

putc(3S), del_curterm(3X), tigetflag(3X), <term.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

(ENHANCED CURSES)

NAME

tigetflag, tigetnum, tigetstr, tparm — retrieve capabilities from the **terminfo** database

SYNOPSIS

```
#include <term.h>

int tigetflag(char *capname);

int tigetnum(char *capname);

char *tigetstr(char *capname);

char *tparm(char *cap, long p1, long p2, long p3, long p4,
            long p5, long p6, long p7, long p8, long p9);
```

DESCRIPTION

The **tigetflag()**, **tigetnum()**, and **tigetstr()** functions obtain boolean, numeric and string capabilities, respectively, from the selected record of the **terminfo** database. For each capability, the value to use as *capname* appears in the **Capname** column in the table in *Defined Capabilities* in terminfo(4).

The **tparm()** function takes as *cap* a string capability. If *cap* is parameterised (as described in *Parameterised Strings* in terminfo(4)), **tparm()** resolves the parameterisation. If the parameterised string refers to parameters **%p1** through **%p9**, then **tparm()** substitutes the values of *p1* through *p9*, respectively.

RETURN VALUE

Upon successful completion, **tigetflg()**, **tigetnum()** and **tigetstr()** return the specified capability. The **tigetflag()** function returns –1 if *capname* is not a boolean capability. The **tigetnum()** function returns –2 if *capname* is not a numeric capability. The **tigetstr()** function returns (**char ***)–1 if *capname* is not a string capability.

Upon successful completion, **tparm()** returns *str* with parameterisation resolved. Otherwise, it returns a null pointer.

ERRORS

No errors are defined.

APPLICATION USAGE

For parameterised string capabilities, the application should pass the return value from **tigetstr()** to **tparm()**, as described above.

Applications intending to send terminal capabilities directly to the terminal (which should only be done using **tputs()** or **putp()**) instead of using Curses, normally should obey the following rules:

- Call **reset_shell_mode()** to restore the display modes before exiting.
- If using cursor addressing, output **enter_ca_mode** upon startup and output **exit_ca_mode** before exiting.
- If using shell escapes, output **exit_ca_mode** and call **reset_shell_mode()** before calling the shell; call **reset_prog_mode()** and output *enter_ca_mode* after returning from the shell.

All parameterised terminal capabilities defined in this document can be passed to **tparm()**. Some implementations create their own capabilities, create capabilities for non-terminal devices, and redefine the capabilities in this document. These practices are non-conforming because it may be that **tparm()** cannot parse these user-defined strings.

SEE ALSO

def_prog_mode(3X), tgetent(3X), putp(3X), <term.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

tmpfile() - create a temporary file

SYNOPSIS

```
#include <stdio.h>
FILE *tmpfile(void);
```

DESCRIPTION

tmpfile() creates a temporary file by generating a name through **tmpnam()** (see *tmpnam*(3S)), and returns a corresponding **FILE** pointer. If the file cannot be opened a NULL pointer is returned. The file is automatically deleted when the process using it terminates. The file is opened for update (**wb+**).

ERRORS

The **tmpfile()** function will fail if:

[EOVERFLOW] The named file is a regular file and the size of the file cannot be represented correctly in an object of size **off_t** in this environment.

Additional **errno** values may be set by the underlying **fopen()** function (see *fopen*(3S)).

APPLICATION USAGE

tmpfile() is thread-safe. It is not async-cancel-safe. A cancellation point may occur when a thread is executing **tmpfile()**.

NOTES

On HP-UX systems, the **wb+** mode is equivalent to the **w+** mode.

SEE ALSO

creat(2), unlink(2), mktemp(3C), fopen(3S), fgetpos64(3S), tmpnam(3S).

STANDARDS CONFORMANCE

tmpfile(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

t

NAME

tmpnam(), tempnam() - create a name for a temporary file

SYNOPSIS

```
#include <stdio.h>
char *tmpnam(char *s);
char *tempnam(const char *dir, const char *pfx);
```

DESCRIPTION

tmpnam() and **tempnam()** 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()** destroys 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*. For multi-thread applications, if *s* is a NULL pointer, the operation is not performed and a NULL pointer is returned.

tempnam() allows the user to control the choice of a directory. The argument *dir* points to the name of the directory in which the file is to be created. If *dir* is NULL or points to a string that is not an appropriate directory name, the path-prefix defined as **P_tmpdir** in the <**stdio.h**> header file is used. If that directory is not accessible, **/tmp** is 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.

In order to request the default behavior for either **tempnam()** or **tmpnam()**, a NULL value must be passed in *dir* and *pfx* for **tempnam()**, or in *s* for **tmpnam()**. If valid parameters are not passed in, behavior is undefined.

Many applications are written such that temporary files have certain initial character sequences in their names. Use the *pfx* argument to define a given prefix. The argument can be NULL or point to a string of up to five characters to be used as the first characters in the temporary-file name.

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

APPLICATION USAGE

tmpnam() and **tempnam()** are thread-safe. These interfaces are not async-cancel-safe.

NOTES

tmpnam() and **tempnam()** generate a different file name each time they are called, but start recycling previously used names if called more than **TMP_MAX** times in a single process.

Files created using these functions and either **fopen()** or **creat()** (see *fopen*(3S) and *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 it is no longer needed.

WARNINGS

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 such that duplication by other means is unlikely.

SEE ALSO

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

STANDARDS CONFORMANCE

tmpnam(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

tempnam(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

NAME
touchwin — window refresh control function

SYNOPSIS
```
#include <curses.h>
int touchwin(WINDOW *win);
```

DESCRIPTION
The **touchwin()** function touches the specified window (that is, marks it as having changed more recently than the last refresh operation).

RETURN VALUE
Upon successful completion, the this function return OK. Otherwise, it returns ERR.

ERRORS
No errors are defined.

APPLICATION USAGE
Calling **touchwin()** or **touchline()** is sometimes necessary when using overlapping windows, since a change to one window affects the other window, but the records of which lines have been changed in the other window do not reflect the change.

SEE ALSO
Screens, Windows and Terminals in curs_intro, doupdate(3X), is_linetouched(3X), <curses.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

t

NAME

trunc( ) - truncation function

SYNOPSIS

```
#include <math.h>
double trunc(double x);
```

DESCRIPTION

trunc() rounds its argument to the integral value, in floating-point format, nearest to but no larger in magnitude than the argument.

The ISO/ANSI C committee has approved the **trunc()** function for inclusion in the C9X draft standard.

To use this function, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

The **trunc()** function returns the truncated integral value.

ERRORS

No errors are defined.

SEE ALSO

ceil(3M), floor(3M), fabs(3M), fmod(3M), fegetround(3M), fesetround(3M), lrint(3M), llrint(3M), lround(3M), llround(3M), rint(3M), round(3M), math(5), fenv(5).

t

NAME

tsearch(), tfind(), tdelete(), twalk() - manage binary search trees

SYNOPSIS

```
#include <search.h>

void *tsearch(
     const void *key,
     void **rootp,
     int (*compar)(const void *, const void *)
);

void *tfind(
     const void *key,
     void * const *rootp,
     int (*compar)(const void *, const void *)
);

void *tdelete(
     const void *key,
     void **rootp,
     int (*compar)(const void *, const void *)
);

void twalk(
     const void *root,
     void (*action)(const void *, VISIT, int)
);
```

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, *compar*. 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 an entry to be accessed or stored. If there is an entry in the tree equal to the value pointed to by *key*, a pointer to the previous key associated with this found entry is returned. Otherwise, *key* is inserted, and a pointer to it returned. Note that since the value returned is a pointer to *key* and *key* itself is a pointer, the value returned is a pointer to a pointer. 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 is set to point to the entry which will be at the root of the new tree.

Like **tsearch()**, **tfind()** searches for an entry in the tree, returning a pointer to it if found. However, if it is not found, **tfind()** returns a NULL pointer. The arguments for **tfind()** are the same as for **tsearch()**.

tdelete() deletes a node from a binary search tree. Arguments are the same as for **tsearch()**. The variable pointed to by *rootp* is 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:

- First argument is the address of the node being visited.
- 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.
- Third argument is the level of the node in the tree, with the root being level zero.

APPLICATION USAGE

tsearch(), **tfind()**, **tdelete()** and **twalk()** are thread-safe. These interfaces are not async-cancel-safe.

EXAMPLE

The following code reads 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 <stdlib.h>
#include <search.h>
#include <stdio.h>
#include <string.h>

struct element    /* pointers to these are stored in the tree */
{
    char *string;
    int length;
 };
char string_space[10000];   /* space to store strings */
struct element elements[500];           /* elements to store */
struct element *root = NULL;    /* this points to the root */
void print_node(void *, VISIT, int);
int element_compare(const void *, const void *);

main( )
{
    char *strptr = string_space;
    struct element *element_ptr = elements;
    struct element **ts_retval;

    int i = 0;

    while (gets(strptr) != NULL && i++ < 500)
    {
        /* set element */
        element_ptr->string = strptr;
        element_ptr->length = strlen(strptr);

        /* put element into the tree */
        ts_retval = (struct element **) tsearch((void *) element_ptr,
                    (void **) &root, element_compare);

        if (*ts_retval == element_ptr)
        {
            (void) printf("The element \"%s\" ",
                          (*ts_retval)->string);
            (void) printf("has now been inserted into the tree\n");
        }
        else
        {
            (void) printf("The element \"%s\" ",
                          (*ts_retval)->string);
            (void) printf("already existed in the tree\n");
        }

        /* adjust pointers, so we don't overwrite tree */
        strptr += element_ptr->length + 1;
        element_ptr++;
    }
    twalk((void *) root, print_node);
}
/* This routine compares two elements, based on an
      alphabetical ordering of the string field.  */
int
element_compare(elem1, elem2)
void *elem1, *elem2;
```

```
{
    return strcmp(((struct element *) elem1)->string,
                  ((struct element *) elem2)->string);
}
/* This routine prints out a node, the first time
     twalk encounters it.  */
void
print_node(element, order, level)
void *element;
VISIT order;
int level;
{
    if (order == preorder || order == leaf)
    {
        (void) printf("string = %20s,  length = %d\n",
                      (*(struct element **) element)->string,
                      (*(struct element **) element)->length);
    }
}
```

RETURN VALUE

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

A NULL pointer is returned by **tsearch()**, **tfind()**, and **tdelete()** if *rootp* is NULL on entry.

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

WARNINGS

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

Two nomenclatures are used to refer to the order in which tree nodes are visited. **tsearch()** uses preorder, postorder and endorder to respectively refer to visiting a node before any of its children, after its left child and before its right and after both its children. The alternate nomenclature uses preorder, inorder, and postorder to refer to the same visits, which could result in some confusion over the meaning of postorder. If the calling function alters the pointer to the root, results are unpredictable.

SEE ALSO

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

STANDARDS CONFORMANCE

tsearch(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

tdelete(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

tfind(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

twalk(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

NAME

ttyname(), ttyname_r(), isatty() - find name of a terminal

SYNOPSIS

```
#include <unistd.h>
char *ttyname(int fildes);
int ttyname_r(int fildes, char *buffer, size_t buflen);
int isatty(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.

Reentrant Interfaces

ttyname_r() returns the result string in the supplied buffer. The *buffer* is *buflen* characters long and should have space for the name and the terminating null character. The maximum length of the terminal name is **TTY_NAME_MAX**.

RETURN VALUE

ttyname() returns a NULL pointer if *fildes* does not describe a terminal device in directory **/dev**.
ttyname_r() returns a zero upon success and an error number upon failure.

ERRORS

isatty() and **ttyname()** fail if any of the following conditions are encountered:

[EBADF]	The *fildes* argument is invalid.
[ENOTTY]	An inappropriate I/O control operation has been attempted.

APPLICATION USAGE

The return value for **ttyname()** points to static data whose content is overwritten by each call. **ttyname()** is not thread-safe. **ttyname_r()** is thread-safe. **ttyname()** and **ttyname_r()** are not async-cancel-safe. A cancellation point may occur when a thread is executing **ttyname()** or **ttyname_r()**.

WARNINGS

For streams ptys, **ttyname()** and **isatty()** do not consider master ptys to be tty devices. It should also be noted that **ttyname()** returns a pointer to the master pty name for all master pty devices. This is a result of device files being linked together.

Users of **ttyname_r()** should also note that the prototype of this function has changed in this release for conformance with the POSIX.1c Threads standard. The old prototype of **ttyname_r()** is supported for compatibility with existing DCE applications only.

FILES

/dev/*
/dev/pty/*

STANDARDS CONFORMANCE

ttyname(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

ttyname_r(): POSIX.1c

isatty(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

ttyslot() - find the slot in the utmp file of the current user

SYNOPSIS

```
#include <stdlib.h>
int ttyslot(void);
```

DESCRIPTION

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

RETURN VALUE

ttyslot() returns –1 if an error was encountered while searching for the terminal name or if none of file descriptors 0, 1, or 2 is associated with a terminal device.

APPLICATION USAGE

ttyslot() is thread-safe. It is not async-cancel-safe. A cancellation point may occur when a thread is executing **ttyslot()**.

FILES

/etc/utmp

SEE ALSO

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

STANDARDS CONFORMANCE

ttyslot(): XPG2

t

NAME
typeahead — control checking for typeahead

SYNOPSIS
```
#include <curses.h>
int typeahead(int fildes);
```

DESCRIPTION
The **typeahead()** function controls the detection of typeahead during a refresh, based on the value of *fildes*:

- If *fildes* is a valid file descriptor, typeahead is enabled during refresh; Curses periodically checks *fildes* for input and aborts the refresh if any character is available. (This is the initial setting, and the typeahead file descriptor corresponds to the input file associated with the screen created by **initscr()** or **newterm()**.) The value of *fildes* need not be the file descriptor on which the refresh is occurring.
- If *fildes* is –1, Curses does not check for typeahead during refresh.

RETURN VALUE
Upon successful completion, **typeahead()** returns OK. Otherwise, it returns ERR.

ERRORS
No errors are defined.

SEE ALSO
Input Processing in curses_intro(3X), doupdate(3X), getch(3X), initscr(3X), <curses.h>, *X/Open System Interface Definitions, Issue 4, Version 2* specification, Section 9.2, *Parameters That Can Be Set*.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The entry is rewritten for clarity. The *RETURN VALUE* section now states that the function returns OK on success and ERR on failure. No return values were defined in previous issues.

t

NAME
unctrl — generate printable representation of a character

SYNOPSIS
```
#include <unctrl.h>
char *unctrl(chtype c);
```

DESCRIPTION
The **unctrl()** function generates a character string that is a printable representation of *c*. If *c* is a control character, it is converted to the ^*X* notation. If *c* contains rendition information, the effect is undefined.

RETURN VALUE
Upon successful completion, **unctrl()** returns the generated string. Otherwise, it returns a null pointer.

ERRORS
No errors are defined.

SEE ALSO
keyname(3X), wunctrl(3X), <unctrl.h>.

CHANGE HISTORY
First released in X/Open Curses, Issue 2.

X/Open Curses, Issue 4
The entry is rewritten for clarity. The RETURN VALUE section now states that the function may return a null pointer. This condition was not specified in previous issues.

u

NAME

ungetc() - push character back into input stream

SYNOPSIS

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

Obsolescent Interface

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

DESCRIPTION

ungetc() inserts the character *c* (converted to an unsigned char) into the buffer associated with an input *stream*. That character, *c*, is returned by the next call to **getc()** (see *getc*(3S)) on that *stream*. A successful intervening call to a file positioning function with *stream* (**fseek()**, **fsetpos()**, or **rewind()**) erases all memory of the inserted characters.

ungetc() affects only the buffer associated with the input *stream*. It does not affect the contents of the file corresponding to *stream*.

One character of pushback is guaranteed.

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

Obsolescent Interface

ungetc_unlocked() pushes character back into input stream.

APPLICATION USAGE

ungetc() is a thread-safe interface. It is not async-cancel-safe. A cancellation point may occur when a thread is executing **ungetc()**.

RETURN VALUE

If successful, **ungetc()** and **ungetc_unlocked()** return *c* and clear the end-of-file indicator for the stream. **ungetc()** and **ungetc_unlocked()** return EOF if they cannot insert the character.

WARNINGS

ungetc_unlocked() is an obsolescent interface supported only for compatibility with existing DCE applications. New multithreaded applications should use **ungetc()**.

SEE ALSO

flockfile(3S), fseek(3S), fgetpos(3S), getc(3S), setbuf(3S).

STANDARDS CONFORMANCE

ungetc(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

(ENHANCED CURSES)

NAME

ungetch, unget_wch — push a character onto the input queue

SYNOPSIS

```
#include <curses.h>
int ungetch(int ch);
int unget_wch(const wchar_t wch);
```

DESCRIPTION

The **ungetch()** function pushes the single-byte character *ch* onto the head of the input queue.

The **unget_wch()** function pushes the wide character *wch* onto the head of the input queue.

One character of push-back is guaranteed. If these functions are called too many times without an intervening call to **getch()** or **get_wch()**, the operation may fail.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

SEE ALSO

Input Processing in curses_intro(3X), getch(3X), get_wch(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

ungetwc() - push a wide character back into an input stream

SYNOPSIS

```
#include <wchar.h>

wint_t ungetwc(wint_t wc, FILE *stream);
```

Obsolescent Interface

```
wint_t ungetwc_unlocked(wint_t wc, FILE *stream);
```

Remarks:

This function is compliant with the XPG4 Worldwide Portability Interface wide-character I/O functions. It parallels the 8-bit character I/O function defined in *ungetc*(3S).

DESCRIPTION

ungetwc() pushes the character corresponding to the wide-character code *wc* into the buffer associated with an input *stream*. That wide-character code, *wc*, is returned by the next call to **getwc()** (see *getwc*(3C)) on that *stream*. A successful intervening call to a file positioning function with *stream* (**fseek()**, **fsetpos()**, or **rewind()**) erases all memory of the pushed-back characters.

ungetwc() affects only the buffer associated with the input *stream*. It does not affect the contents of the file corresponding to *stream*.

One character of pushback is guaranteed.

If *wc* equals **WEOF**, **ungetwc()** does nothing to the buffer and returns **WEOF**.

The definition for this function, the type **wint_t** and the value **WEOF** are provided in the <**wchar.h**> header.

Obsolescent Interface

ungetwc_unlocked() pushes a wide character back into an input stream.

APPLICATION USAGE

ungetwc() is a thread-safe interface. It is not async-cancel-safe. A cancellation point may occur when a thread is executing **ungetwc()**.

EXTERNAL INFLUENCES

Locale

The **LC_CTYPE** category determines how wide character conversions are done.

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

If successful, **ungetwc()** and **ungetwc_unlocked()** return *wc* and clear the end-of-file indicator for the stream. **ungetwc()** and **ungetwc_unlocked()** return **WEOF** if they cannot insert the wide character.

WARNINGS

ungetwc_unlocked() is an obsolescent interface supported only for compatibility with existing DCE applications. New multithreaded applications should use **ungetwc()**.

AUTHOR

ungetwc() and **ungetwc_unlocked()** were developed by OSF and HP.

SEE ALSO

flockfile(3S), fseek(3S), fgetpos(3S), getwc(3C), setbuf(3S).

STANDARDS CONFORMANCE

ungetwc(): XPG4

NAME

unlockpt - unlock a STREAMS pty master and slave pair

SYNOPSIS

```
int unlockpt (int fildes);
```

DESCRIPTION

The passed parameter, *fildes*, is a file descriptor that is returned from a successful open of a master pty (pseudo-terminal) device. The **unlockpt()** function unlocks a slave pty from its associated master counterpart. It does this by clearing a lock flag so that the slave pty can be opened. For security reason, *grantpt*(3C) must be executed before *unlockpt*(3C).

RETURN VALUE

Upon successful completion, the **unlockpt()** function returns a value of 0 (zero). Otherwise, it returns a value of -1.

Failure may result under the following conditions:

- The file descriptor specified by the *fildes* parameter is not an open file descriptor.
- The file descriptor specified by the *fildes* parameter is not associated with a STREAMS pty master device.

EXAMPLES

The following example shows how **unlockpt** is typically used.

```
int fd_master, fd_slave;
char *slave;
    ...
    fd_master = open("/dev/ptmx", O_RDWR);
    grantpt(fd_master);
    unlockpt(fd_master);
    slave = ptsname(fd_master);
    fd_slave = open(slave, O_RDWR);
    ioctl(fd_slave, I_PUSH, "ptem");
    ioctl(fd_slave, I_PUSH, "ldterm");
```

AUTHOR

unlockpt() was developed by HP and OSF.

SEE ALSO

open(2), grantpt(3C), ptsname(3C), ptm(7), pts(7), ptem(7).

u

(ENHANCED CURSES)

NAME

use_env — specify source of screen size information

SYNOPSIS

```
#include <curses.h>
void use_env(bool boolvalue);
```

DESCRIPTION

The **use_env()** function specifies the technique by which the implementation determines the size of the screen. If *boolvalue* is FALSE, the implementation uses the values of *lines* and *columns* specified in the **terminfo** database. If *boolvalue* is TRUE, the implementation uses the **LINES** and **COLUMNS** environment variables. The initial value is TRUE.

Any call to **use_env()** must precede calls to **initscr()**, **newterm()** or **setupterm()**.

RETURN VALUE

The function does not return a value.

ERRORS

No errors are defined.

SEE ALSO

del_curterm(3X), initscr(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

u

(ENHANCED CURSES)

NAME

vidattr, vid_attr, vidputs, vid_puts — output attributes to the terminal

SYNOPSIS

```
#include <curses.h>

int vidattr(chtype attr);

int vid_attr(attr_t attr, short color_pair_number, void *opt);

int vidputs(chtype attr, int (*putfunc)(int));

int vid_puts(attr_t attr, short color_pair_number, void *opt, int
(*putwfunc)(int));
```

DESCRIPTION

These functions output commands to the terminal that change the terminal's attributes.

If the **terminfo** database indicates that the terminal in use can display characters in the rendition specified by *attr*, then **vidattr()** outputs one or more commands to request that the terminal display subsequent characters in that rendition. The function outputs by calling **putchar()**. The **vidattr()** function neither relies on nor updates the model Curses maintains of the prior rendition mode.

The **vidputs()** function computes the same terminal output string that **vidattr()** does, based on *attr*, but **vidputs()** outputs by calling the user-supplied function *putfunc*.

The user-supplied function *putfunc* (specified as an argument to **vidputs()**) is either **putchar()** or some other function with the same prototype. The **vidputs()** function ignores the return value of *putfunc*.

The **vid_attr()** and **vid_puts()** functions correspond to **vidattr()** and **vidputs()**, respectively, but take a set of arguments, one of type **attr_t** for the attributes **short** for the color pair number and **void *** and thus support the attribute constants with the WA_ prefix.

The *opts* argument is reserved for definition in a future edition of this document. Currently, the application must provide a null pointer as *opts*.

The user-supplied function *putwfunc* (specified as an argument to **vid_puts()**) is either **putwchar()** or some other function with the same prototype. The **vid_puts()** function ignores the return value of *putwfunc*.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

After use of any of these functions, the model Curses maintains of the state of the terminal might not match the actual state of the terminal. The application should touch and refresh the window before resuming conventional use of Curses.

Use of these functions requires that the application contain so much information about a particular class of terminal that it defeats the purpose of using Curses.

On some terminals, a command to change rendition conceptually occupies space in the screen buffer (with or without width). Thus, a command to set the terminal to a new rendition would change the rendition of some characters already displayed.

SEE ALSO

doupdate(3X), is_linetouched(3X), putchar() (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification), putwchar() (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification), tigetflag(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

NAME

vprintf(), vfprintf(), vsprintf(), vsnprintf() - print formatted output of a varargs argument list

SYNOPSIS

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

int vprintf(const char *format, va_list ap);

int vfprintf(FILE *stream, const char *format, va_list ap);

int vsprintf(char *s, const char *format, va_list ap);

int vsnprintf(char *s, size_t maxsize, const char *format, va_list ap);
```

DESCRIPTION

vprintf(), **vfprintf()**, **vsprintf()**, and **vsnprintf()** are the same as **printf()**, **fprintf()**, **sprintf()**, and **snprintf()** respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by **<stdarg.h>**.

EXAMPLE

The following demonstrates how **vfprintf()** could be used to write an error routine:

```
#include <stdarg.h>
#include <stdio.h>
    .
    .
    .
/*
 *  error should be called using the form:
 *      error(function_name, format, arg1, arg2...);
 */

/*VARARGS0*/
void
error(va_alist)
va_dcl
{
    va_list args;
    char *fmt;

    va_start(args);

    /* print out name of function causing error */
    (void)fprintf(stderr, "ERROR in %s: ", va_arg(args, char *));
    fmt = va_arg(args, char *);

    /* print out remainder of message */
    (void)vfprintf(stderr, fmt, args);
    va_end(args);
    (void)abort( );
}
```

APPLICATION USAGE

The interfaces **vprintf()**, **vfprintf()**, **vsnprintf()** and **vsprintf()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing **vprintf()** or **vfprintf()**.

SEE ALSO

printf(3S), setlocale(3C), varargs(5).

STANDARDS CONFORMANCE

vprintf(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

vfprintf(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
vsprintf(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

V

NAME

vscanf(), vfscanf(), vsscanf() - formatted input conversion to a varargs argument list, read from stream file

SYNOPSIS

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

int vscanf(const char *format, va_list ap);

int vfscanf(FILE *stream, const char *format, va_list ap);

int vsscanf(char *s, const char *format, va_list ap);
```

DESCRIPTION

vscanf(), **vfscanf()**, and **vsscanf()** are the same as **scanf()**, **fscanf()**, and **sscanf()** 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).

APPLICATION USAGE

vscanf(), **vfscanf()** and **vsscanf()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing **vscanf()** or **vfscanf()**.

SEE ALSO

scanf(3S), setlocale(3C), varargs(5).

V

(ENHANCED CURSES)

NAME

vw_printw — print formatted output in window (**TO BE WITHDRAWN**)

SYNOPSIS

```
#include <stdarg.h> #include <curses.h>
int vw_printw(WINDOW *win, char *fmt, va_list varglist);
```

DESCRIPTION

The **vw_printw()** function achieves the same effect as **wprintw()** using a variable argument list. The third argument is a **va_list**, as defined in <stdarg.h>.

RETURN VALUE

Upon successful completion, **vw_printw()** returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

SEE ALSO

mvprintw(3X), fprintf() (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification), <curses.h>, <stdarg.h> (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification).

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

V

(ENHANCED CURSES)

NAME

vw_scanw — convert formatted input from a window (**TO BE WITHDRAWN**)

SYNOPSIS

```
#include <stdarg.h> #include <curses.h>
int vw_scanw(WINDOW *win, char *fmt, va_list varglist);
```

DESCRIPTION

The **vw_scanw()** function achieves the same effect as **wscanw()** using a variable argument list. The third argument is a **va_list**, as defined in <stdarg.h>.

RETURN VALUE

Upon successful completion, **vw_scanw()** returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

SEE ALSO

fscanf() (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification), mvscanw(3X), <curses.h>, <stdarg.h> (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification).

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

V

(ENHANCED CURSES)

NAME

vwprintw — print formatted output in window

SYNOPSIS

```
#include <stdarg.h> #include <curses.h>
int vwprintw(WINDOW *win, char *fmt, va_list varglist);
```

DESCRIPTION

The **vwprintw()** function achieves the same effect as **wprintw()** using a variable argument list. The third argument is a **va_list**, as defined in <stdarg.h>.

RETURN VALUE

Upon successful completion, **vwprintw()** returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

The **vwprintw()** function is deprecated because it relies on deprecated functions in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification. The **vw_printw()** function is preferred.

SEE ALSO

mvprintw(3X), fprintf() (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification), vw_printw(3X), <curses.h>, <stdarg.h> (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification).

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

V

(ENHANCED CURSES)

NAME

vwscanw — convert formatted input from a window

SYNOPSIS

```
#include <stdarg.h> #include <curses.h>
int vwscanw(WINDOW *win, char *fmt, va_list varglist);
```

DESCRIPTION

The **vwscanw()** function achieves the same effect as **wscanw()** using a variable argument list. The third argument is a **va_list**, as defined in <stdarg.h>.

RETURN VALUE

Upon successful completion, **vwscanw()** returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

APPLICATION USAGE

The **vwscanw()** function is deprecated because it relies on deprecated functions in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification. The **vw_scanw()** function is preferred.

SEE ALSO

fscanf() (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification), mvscanw(3X), vw_scanw(), <curses.h>, <stdarg.h> (in the *X/Open System Interfaces and Headers, Issue 4, Version 2* specification).

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

V

NAME

towupper(), towlower() - translate wide characters

SYNOPSIS

```
#include <wchar.h>
wint_t towupper(wint_t wc);
wint_t towlower(wint_t wc);
```

Remarks:

These functions are compliant with the XPG4 Worldwide Portability Interface wide-character conversion functions. They parallel the 8-bit character conversion functions defined in *conv*(3C).

DESCRIPTION

towupper() and **towlower()** have as domain a **wint_t**, the value of which is representable as a **wchar_t** or the value **WEOF**. If the argument has any other value, the behavior is undefined. If the argument of **towupper()** represents a lowercase letter, the result is the corresponding uppercase letter. If the argument of **towlower()** represents an uppercase letter, the result is the corresponding lowercase letter. All other arguments are returned unchanged.

Definitions for these functions, the types **wint_t**, **wchar_t**, and the value **WEOF** are provided in the <**wchar.h**> header.

APPLICATION USAGE

towupper() and **towlower()** are thread-safe and async-cancel-safe.

EXTERNAL INFLUENCES

Locale

The **LC_CTYPE** category determines the translations to be done.

International Code Set Support

Single-byte and multi-byte character code sets are supported.

AUTHOR

wconv() was developed by IBM, OSF, and HP.

SEE ALSO

conv(3C), multibyte(3C), wctype(3C), setlocale(3C), lang(5).

STANDARDS CONFORMANCE

towlower(): XPG4

towupper(): XPG4

W

NAME

wcsftime() - convert date and time to wide-character string

SYNOPSIS

```
#include <wchar.h>

size_t wcsftime(
   wchar_t *ws,
   size_t maxsize,
   const char *format,
   const struct tm *timeptr
);
```

Remarks:

This function is compliant with the XPG4 Worldwide Portability Interface wide-character formatting functions. It parallels the 8-bit character formatting function defined in *strftime*(3C).

DESCRIPTION

wcsftime() converts the contents of a **tm** structure (see *ctime*(3C)) to a formatted date and time wide-character string.

wcsftime() places wide characters into the array pointed to by *ws* as controlled by the string pointed to by *format*. The *format* string consists of zero or more directives and ordinary characters. A directive consists of a **%** character, an optional field width and precision specification, and a terminating character that determines the directive's behavior. All ordinary characters (including the terminating null character) are converted into corresponding wide characters and are copied into the array. No more than *maxsize* wide characters are placed into the array. Each directive is replaced by the appropriate wide characters as described in the following list. The appropriate wide characters are determined by the program's locale, by the values contained in the structure pointed to by *timeptr*, and by the **TZ** environment variable (see External Influences below).

The definition for this function and the type **wchar_t** are provided in the <**wchar.h**> header.

Directives

The following directives, shown without the optional field width and precision specification, are replaced by the corresponding wide characters as indicated:

%a	Locale's abbreviated weekday name.
%A	Locale's full weekday name.
%b	Locale's abbreviated month name.
%B	Locale's full month name.
%c	Locale's appropriate date and time representation.
%C	The century number (the year divided by 100 and truncated to an integer) as a decimal number [00-99].
%d	Day of the month as a decimal number [01,31].
%D	Equivalent to the directive string *%m/%d/%y*.
%e	Day of the month as a decimal number [1,31]; a single digit is preceded by a space.
%h	Equivalent to %b.
%H	Hour (24-hour clock) as a decimal number [00,23].
%I	Hour (12-hour clock) as a decimal number [01,12].
%j	Day of the year as a decimal number [001,366].
%m	Month as a decimal number [01,12].
%M	Minute as a decimal number [00,59].
%n	The New-line character.
%p	Locale's equivalent of either AM or PM.
%r	The time in AM and PM notation; in the POSIX locale this is equivalent to *%I:%M:%S %p*.
%R	The time in 24 hour notation (%H:%M).
%S	Second as a decimal number [00,61].
%t	The Tab character.
%T	The time in hours, minutes, and seconds (%H:%M:%S).
%u	The weekday as a decimal number [1(Monday),7].
%U	Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.

%V The week number of the year (Monday as the first day of the week) as a decimal number [01,53]. If the week containing January 1st has four or more days in the new year, then it is considered week 1; otherwise, it is week 53 of the previous year, and the next week is week 1.

%w Weekday as a decimal number [0(Sunday),6].

%W Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.

%x Locale's appropriate date representation.

%X Locale's appropriate time representation.

%y Year without century as a decimal number [00,99].

%Y Year with century as a decimal number.

%Z Time zone name (or by no characters if no time zone exists).

%% The percent (%) character.

The following directives are provided for backward compatibility with the directives supported by *date*(1) and the *ctime*(3C) functions. These directives may be removed in a future release. It is recommended that the directives above be used in preference to those below.

%E Locale's combined Emperor/Era name and year (use **%EC%Ey** instead).

%F Locale's full month name (use **%B** instead).

%N Locale's Emperor/Era name (use **%EC** instead).

%o Locale's Emperor/Era year (use **%Ey** instead).

%z Time zone name (or by no characters if no time zone exists) (use **%Z** instead).

If a directive is not one of the above, the behavior is undefined.

Modified Conversion Specifiers

Some conversion specifiers can be modified by the E or O modifier characters to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified conversion specifier. If the alternative format or specification does not exist for the current locale, the behavior will be as if the unmodified conversion specification were used. Alternative numeric symbols refers to those symbols defined by the **ALT_DIGIT** (see *langinfo*(5)) in the locale.

%Ec The locales alternative appropriate date and time representation.

%EC The name of the base year (period/Emperor/Era) in the locale's alternative representation.

%Ex The locale's alternative date representation

%EX The locale's alternative time representation.

%Ey The offset from %EC (year only) in the locale's alternative representation.

%EY The full alternative year representation.

%Od The day of the month, using the locale's alternative numeric symbols, filled as needed with leading zeros if there is any alternative symbol for zero, otherwise with leading spaces.

%Oe the day of the month, using the locale's alternative numeric symbols, filled as needed with leading spaces.

%OH The hour (24-hour clock) using the locale's alternative numeric symbols.

%OI The hour (12-hour clock) using the locale's alternative numeric symbols.

%Om The month using the locale's alternative numeric symbols.

%OM The minutes using the locale's alternative numeric symbols.

%OS The seconds using the locale's alternative numeric symbols.

%Ou The weekday as a number in the locale's alternative representation (Monday=1).

%OU The week number of the year (Sunday as the first day of the week, rules corresponding to %U) using the locale's alternative numeric symbols.

%OV The week number of the year (Monday as the first day of the week, rules corresponding to %V) using tht locale's alternative numeric symbols.

%Ow The number of the weekday (Sunday=0) using the locale's alternative numeric symbols.

%OW The week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.

%Oy The year (offset from %C) in the locale's alternative representation and using the locale's alternative symbols.

Field Width and Precision

An optional field width and precision specification can immediately follow the initial **%** of a directive in the following order:

[**-**|**0**]*w* The decimal digit string *w* specifies a minimum field width in which the result of the conversion is right- or left-justified. It is right-justified (with space padding) by default. If the optional **-** character is specified, it is left-justified with space padding on the right. If the optional **0** character is specified, it is right-justified and padded with zeros on the left.

.*p* The decimal digit string *p* specifies the minimum number of digits to appear for the **d**, **H**, **I**, **j**, **m**, **M**, **o**, **S**, **U**, **w**, **W**, **y** and **Y** directives, and the maximum number of corresponding wide characters to be used from the **a**, **A**, **b**, **B**, **c**, **D**, **E**, **F**, **h**, **n**, **N**, **p**, **r**, **t**, **T**, **x**, **X**, **z**, **Z**, and **%** directives. In the first case, if a directive supplies fewer digits than specified by the precision, it is expanded with leading zeros. In the second case, if a directive supplies more characters than specified by the precision, excess characters are truncated on the right.

If no field width or precision is specified for a **d**, **H**, **I**, **m**, **M**, **S**, **U**, **W**, **y**, or **j** directive, a default of **.2** is used for all but **j** for which **.3** is used.

APPLICATION USAGE

wcsftime() is thread-safe. It is not async-cancel-safe.

EXTERNAL INFLUENCES

Locale

The **LC_TIME** category determines the characters to be substituted for those directives described above as being from the locale.

The **LC_CTYPE** category determines the interpretation of the bytes within *format* as single and/or multibyte characters as well as how wide-character conversions are done.

The **LC_NUMERIC** category determines the characters used to form numbers for those directives that produce numbers in the output. If **ALT_DIGITS** (see *langinfo*(5)) is defined for the locale, the characters so specified are used in place of the default ASCII characters. If both **ALT_DIGITS** and **ALT_DIGIT** is defined for the locale, **ALT_DIGITS** will take precedence over **ALT_DIGIT**.

Environment Variables

TZ determines the time zone name substituted for the **%Z** and **%z** directives. The time zone name is determined by calling the function **tzset()** which sets the external variable **tzname** (see *ctime*(3C)).

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

If the total number of resulting wide characters including the terminating null wide character is not more than **maxsize**, **wcsftime()** returns the number of wide characters placed into the array pointed to by *ws*, not including the terminating null wide character. Otherwise, zero is returned and the contents of the array are indeterminate.

EXAMPLES

If the *timeptr* argument contains the following values:

```
timeptr->tm_sec = 4;
timeptr->tm_min = 9;
timeptr->tm_hour = 15;
timeptr->tm_mday = 4;
timeptr->tm_mon = 6;
```

```
timeptr->tm_year = 88;
timeptr->tm_wday = 1;
timeptr->tm_yday = 185;
timeptr->tm_isdst = 1;
```

the following combinations of the **LC_TIME** category and format strings produce the indicated output:

LC_TIME	Format String	Output
en_US.roman8	%x	Mon, Jul 4, 1988
de_De.roman8	%x	Mo., 4. Juli 1988
en_US.roman8	%X	03:09:04 PM
fr_FR.roman8	%X	15h09 04
any *	%H:%M:%S	15:09:04
any *	%.1H:%.1M:%.1S	15:9:4
any *	%2.1H:%-3M:%03.1S	15:9 :004

* The directives used in these examples are not affected by the **LC_TIME** category of the locale.

WARNINGS

The function **tzset()** is called upon every invocation of **wcsftime()** (whether or not the time zone name is copied to the output array).

The range of values for **%S** ([0,61]) extends to 61 to allow for the occasional one or two leap seconds. However, the system does not accumulate leap seconds and the **tm** structure generated by the functions **localtime()** and **gmtime()** (see *ctime*(3C)) never reflects any leap seconds.

Results are undefined if values contained in the structure pointed to by *timeptr* exceed the ranges defined for the **tm** structure (see *ctime*(3C)) or are not consistent (such as if the **tm_yday** element is set to 0, indicating the first day of January, while the **tm_mon** element is set to 11, indicating a day in December).

AUTHOR

wcsftime() was developed by OSF and HP.

SEE ALSO

date(1), ctime(3C), setlocale(3C), environ(5), langinfo(5), hpnls(5).

STANDARDS CONFORMANCE

wcsftime(): XPG4

W

NAME

wcstod() - convert wide character string to double-precision number

SYNOPSIS

```
#include <wchar.h>
double wcstod(const wchar_t *nptr, wchar_t **endptr);
```

Remarks:

This function is compliant with the XPG4 Worldwide Portability Interface wide-character formatting functions. It parallels the 8-bit character formatting function defined in *strtod*(3C).

DESCRIPTION

wcstod() returns, as a double-precision floating-point number, the value represented by the wide character string pointed to by *nptr*. The wide character string is scanned (leading white-space characters as defined by **iswspace()** in *wctype*(3C) are ignored) up to the first unrecognized character. If no conversion can take place, zero is returned.

wcstod() recognizes wide characters in the following sequence:

1. An optional string of "white-space" wide characters which are ignored,
2. An optional sign,
3. A string of digits optionally containing a radix character,
4. An optional **e** or **E** followed by an optional sign or space, followed by an integer.

The radix character is determined by the current NLS environment (see *setlocale*(3C)). If **setlocale()** has not been called successfully, the default NLS environment, "C", is used (see *lang*(5)). The default environment specifies a period (.) as the radix character.

If the value of *endptr* is not **(wchar_t **)NULL**, the variable to which it points is set to point at the wide character after the last number, if any, that was recognized. If no number can be formed, ***endptr* is set to *nptr*, and zero is returned.

The definition for this function and the type **wchar_t** are provided in the <**wchar.h**> header.

APPLICATION USAGE

wcstod() is thread-safe and async-cancel-safe.

EXTERNAL INFLUENCES

Locale

The **LC_NUMERIC** category determines the value of the radix character within the currently loaded NLS environment.

The **LC_CTYPE** category determines how wide character codes are interpreted.

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

If the correct value would cause overflow, **+HUGE_VAL** or **-HUGE_VAL** 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.

If **wcstod()** encounters an input wide character string equal to **inf** or **infinity** (both case insensitive) it will return **HUGE_VAL**. If **wcstod()** encounters an input wide character string equal to **nan** (case insensitive) it will return **_DNANQ**.

AUTHOR

wcstod() was developed by AT&T and HP.

SEE ALSO

wctype(3C), setlocale(3C), scanf(3S), wcstol(3C), hpnls(5), lang(5).

STANDARDS CONFORMANCE

wcstod(): XPG4

NAME

wcstol(), wcstoul() - convert wide character string to long integer

SYNOPSIS

```
#include <wchar.h>

long int wcstol(const wchar_t *nptr, wchar_t **endptr, int base);

unsigned long int wcstoul(const wchar_t *nptr, wchar_t **endptr,
    int base);
```

Remarks:

These functions are compliant with the XPG4 Worldwide Portability Interface wide-character formatting functions. They parallel the 8-bit character formatting functions defined in *strtol*(3C).

DESCRIPTION

wcstol() (**wcstoul()**) converts the wide character string pointed to by *nptr* to **long int** (**unsigned long int**) representation. The wide character string is scanned up to the first wide character inconsistent with the base. Leading "white-space" wide characters (as defined by **iswspace()** in *wctype*(3C)) are ignored. If no conversion can take place, zero is returned.

If *base* is greater than or equal to 2 and less than or equal to 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 wide character string itself determines the base as follows: After an optional leading sign, a leading zero indicates octal conversion; a leading **0x** or **0X** hexadecimal conversion. Otherwise, decimal conversion is used.

If the value of *endptr* is not **(wchar_t **)NULL**, a pointer to the wide character terminating the scan is returned in the location pointed to by *endptr*. If no integer can be formed, the location pointed to by *endptr* is set to *nptr*, and zero is returned.

Definitions for these functions and the type **wchar_t** are provided in the <**wchar.h**> header.

APPLICATION USAGE

wcstol(), and **wcstoul()** are thread-safe and async-cancel-safe.

EXTERNAL INFLUENCES

Locale

The **LC_CTYPE** category determines how wide character codes are interpreted.

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

Upon successful completion, both functions return the converted value, if any. If the correct value would cause overflow, **wcstol()** returns **LONG_MAX** or **LONG_MIN** (according to the sign of the value), and sets **errno** to ERANGE; **wcstoul()** returns **ULONG_MAX** and sets **errno** to ERANGE.

For all other errors, zero is returned and **errno** is set to indicate the error.

ERRORS

wcstol() and **wcstoul()** fail and **errno** is set if any of the following conditions are encountered:

W

[EINVAL]	The value of *base* is not supported.
[ERANGE]	The value to be returned would have caused overflow.

AUTHOR

wcstol() and **wcstoul()** were developed by OSF and HP.

SEE ALSO

wctype(3C), wcstod(3C), scanf(3S).

STANDARDS CONFORMANCE

wcstol(): XPG4

wcstoul(): XPG4

NAME

wcscat(), wcsncat(), wcscmp(), wcsncmp(), wcscpy(), wcsncpy(), wcslen(), wcschr(), wcsrchr(), wcspbrk(), wcsspn(), wcscspn(), wcswcs(), wcstok(), wcscoll(), wcwidth(), wcswidth(), wcsxfrm() - wide character string operations

SYNOPSIS

```
#include <wchar.h>
wchar_t *wcscat(wchar_t *ws1, const wchar_t *ws2);
wchar_t *wcsncat(wchar_t *ws1, const wchar_t *ws2, size_t n);
int wcscmp(const wchar_t *ws1, const wchar_t *ws2);
int wcsncmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);
wchar_t *wcscpy(wchar_t *ws1, const wchar_t *ws2);
wchar_t *wcsncpy(wchar_t *ws1, const wchar_t *ws2, size_t n);
size_t wcslen(const wchar_t *ws);
wchar_t *wcschr(const wchar_t *ws, wint_t wc);
wchar_t *wcsrchr(const wchar_t *ws, wint_t wc);
wchar_t *wcspbrk(const wchar_t *ws1, const wchar_t *ws2);
size_t wcsspn(const wchar_t *ws1, const wchar_t *ws2);
size_t wcscspn(const wchar_t *ws1, const wchar_t *ws2);
wchar_t *wcswcs(const wchar_t *ws1, const wchar_t *ws2);
wchar_t *wcstok(wchar_t *ws1, const wchar_t *ws2);
int wcscoll(const wchar_t *ws1, const wchar_t *ws2);
int wcwidth(wint_t wc);
int wcswidth(const wchar_t *ws, size_t n);
size_t wcsxfrm(wchar_t *ws1, const wchar_t *ws2, size_t n);
```

Obsolescent Interface

```
wchar_t *wcstok_r(wchar_t *ws1, const wchar_t *ws2, wchar_t **wlast);
```

Remarks:

These functions are compliant with the XPG4 Worldwide Portability Interface wide-character string handling functions. They parallel the 8 bit string functions defined in *string*(3C).

DESCRIPTION

The arguments *ws1*, *ws2*, and *ws* point to wide character strings (arrays of type **wchar_t** terminated by a null value).

wcscat() appends a copy of wide string *ws2* to the end of wide string *ws1*. **wcsncat()** appends a maximum of *n* characters; fewer if *ws2* is shorter than *n* characters. Each returns a pointer to the null-terminated result (the value of *ws1*).

wcscmp() compares its arguments and returns an integer less than, equal to, or greater than zero, depending on whether *ws1* is lexicographically less than, equal to, or greater than *ws2*. The comparison of corresponding wide characters is done by comparing numeric values of the wide character codes. Null pointer values for *ws1* and *ws2* are treated the same as pointers to empty wide strings. **wcsncmp()** makes the same comparison but examines a maximum of *n* characters (*n* less than or equal to zero yields equality).

wcscpy() copies wide string *ws2* to *ws1*, stopping after the null value has been copied. **wcsncpy()** copies up to *n* characters from *ws2*, adding null values to *ws1* if necessary, until a total of *n* have been copied. The result is not null-terminated if the length of *ws2* is *n* or more. Each function returns *ws1*. Note that **wcsncpy()** should not be used to copy an arbitrary structure. If that structure contains sizeof(wchar_t) consecutive null bytes, **wcsncpy()** may not copy the entire structure. Use the **memcpy()** function (see *memory*(3C)) to copy arbitrary binary data.

wcslen() returns the number of wide characters in *ws*, not including the terminating null wide character.

wcschr() (**wcsrchr()**) returns a pointer to the first (last) occurrence of wide character *wc* in wide string *ws*, or a null pointer if *wc* does not occur in the wide string. The null wide character terminating a wide string is considered to be part of the wide string.

wcspbrk() returns a pointer to the first occurrence in wide string *ws1* of any wide character from wide string *ws2*, or a null pointer if no wide character from *ws2* exists in *ws1*.

wcsspn() (**wcscspn()**) returns the length of the maximum initial segment of wide string *ws1*, which consists entirely of wide characters from (not from) wide string *ws2*.

wcswcs() returns a pointer to the first occurrence of wide string *ws2* in wide string *ws1*, or a null pointer if *ws2* does not occur in the wide string. If *ws2* points to a wide string of zero length, **wcswcs()** returns *ws1*.

wcstok() considers the wide string *ws1* to consist of a sequence of zero or more text tokens separated by spans of one or more wide characters from the separator wide string *ws2*. The first call (with a non-null pointer *ws1* specified) returns a pointer to the first wide character of the first token, and writes a null wide character into *ws1* immediately following the returned token. The function keeps track of its position in the wide string *ws1* between separate calls, so that subsequent calls made with the first argument a null pointer work through the wide string immediately following that token. In this way subsequent calls work through the wide string *ws1* until no tokens remain. The separator wide string *ws2* can be different from call to call. When no token remains in *ws1*, a null pointer is returned.

wcscoll() returns an integer greater than, equal to, or less than zero, according to whether the wide string pointed to by *ws1* is greater than, equal to, or less than the wide string pointed to by *ws2*. The comparison is based on wide strings interpreted as appropriate to the program's locale (see Locale below). In the "C" locale **wcscoll()** works like **wcscmp()**.

wcwidth() returns the number of column positions required for the wide character *wc*, or 0 if *wc* is a null wide character, or -1 if *wc* is an unprintable wide character.

wcswidth() returns the number of column positions required for *n* wide characters (or fewer than *n* wide characters if a null wide character is encountered before *n* wide characters are exhausted) in the wide string pointed to by *ws*. **wcswidth()** returns 0 if *ws* points to a null wide character, or -1 if *ws* contains an unprintable wide character.

wcsxfrm() transforms the wide character string pointed to by *ws2* and places the resulting wide character string into the array pointed to by *ws1*. The transformation is such that if **wcscmp()** is applied to two transformed wide strings, it returns a value greater than, equal to or less than 0, corresponding to the result of **wcscoll()** applied to the same two original wide character strings. No more than *n* wide-character codes are placed into the resulting array pointed to by *ws1* including the terminating null wide-character. If *n* is 0, *ws1* is permitted to be a null pointer. If copying takes place between objects that overlap, the behaviour is undefined. **wcsxfrm()** returns the length of the transformed wide character string (not including the terminating null wide-character code). If the value returned is *n* or more, the contents of the array pointed to by *ws1* are indeterminate. On error, **wcsxfrm()** returns (size_t) -1, and sets **errno** to **EINVAL** (if the wide character string pointed to by *ws2* contains wide-character codes outside the domain of the collating sequence) or to **ENOSYS** (if the function is not supported).

Definitions for these functions and the type **wchar_t** are provided in header file <**wchar.h**>.

Obsolescent Interface

wcstok_r() performs wide character string operations.

W

EXTERNAL INFLUENCES

Locale

The **LC_COLLATE** category determines the collation ordering used by the **wcscoll()** function.

The **LC_CTYPE** category determines how widths are calculated by the **wcwidth()** and **wcswidth()** functions.

EXAMPLE

The following sample piece of code finds the tokens, separated by blanks, that are in the string *s* (assuming that there are at most **MAXTOK** tokens):

```
int i = 0;
wchar_t *ws, *wlast, *wtok[MAXTOK];

wtok[0] = wcstok_r(ws, L" ", &wlast);
while (wtok[++i] = wcstok_r(NULL, L" ", &wlast));
```

APPLICATION USAGE

The interfaces **wcscat()**, **wcsncat()**, **wcscmp()**, **wcsncmp()**, **wcscpy()**, **wcsncpy()**, **wcslen()**, **wcschr()**, **wcsrchr()**, **wcspbrk()**, **wcsspn()**, **wcscspn()**, **wcswcs()**, **wcstok()**, **wcscoll()**, **wcwidth()**, **wcswidth()** and **wcsxfrm()** are thread-safe. All these interfaces, with the exception of **wcscoll()**, **wcwidth()**, **wcswidth()** and **wcsxfrm()** are async-cancel-safe.

WARNINGS

The functions **wcscat()**, **wcsncat()**, **wcscpy()**, **wcsncpy()**, **wcstok()**, and **wcstok_r()** alter the contents of the array to which *ws1* points. They do not check for overflow of the array.

Null pointers for destination wide strings cause undefined behavior.

Wide character movement is performed differently in different implementations, so copying that involves overlapping source and destination wide strings may yield unexpected results.

For the **wcscoll()** function, the results are undefined if the languages specified by the **LC_COLLATE** and **LC_CTYPE** categories use different code sets.

wcstok_r() is an obsolescent interface supported only for compatibility with existing DCE applications. New multithreaded applications should use **wcstok()**.

wcswidth() will return an incorrect negative value if the number of column positions required for *n* wide characters in the wide string pointed to by *ws* exceeds **INT_MAX**.

AUTHOR

wcstring functions were developed by OSF and HP.

SEE ALSO

wconv(3C), memory(3C), multibyte(3C), setlocale(3C), string(3C), hpnls(5).

STANDARDS CONFORMANCE

wcscat(): XPG4
wcschr(): XPG4
wcscmp(): XPG4
wcscoll(): XPG4
wcscpy(): XPG4
wcscspn(): XPG4
wcslen(): XPG4
wcsncat(): XPG4
wcsncmp(): XPG4
wcsncpy(): XPG4
wcspbrk(): XPG4
wcsrchr(): XPG4
wcsspn(): XPG4
wcstok(): XPG4
wcswcs(): XPG4

W

wcswidth(): XPG4
wcwidth(): XPG4
wcsxfrm(): XPG4

NAME

iswalpha(), iswupper(), iswlower(), iswdigit(), iswxdigit(), iswalnum(), iswspace(), iswpunct(), iswprint(), iswgraph(), iswcntrl(), wctype(), iswctype() - classify wide characters

SYNOPSIS

```
#include <wchar.h>

wctype_t wctype(const char *charclass);
int iswctype(wint_t wc, wctype_t prop);
int iswalnum(wint_t wc);
int iswalpha(wint_t wc);
int iswcntrl(wint_t wc);
int iswdigit(wint_t wc);
int iswgraph(wint_t wc);
int iswlower(wint_t wc);
int iswprint(wint_t wc);
int iswpunct(wint_t wc);
int iswspace(wint_t wc);
int iswupper(wint_t wc);
int iswxdigit(wint_t wc);
```

Remarks:

These functions are compliant with the XPG4 Worldwide Portability Interface wide-character classification functions. They parallel the 8-bit character classification functions defined in *ctype*(3C).

DESCRIPTION

These functions classify wide character values according to the rules of the coded character set identified by the last successful call to **setlocale()** (see *setlocale*(3C)).

If **setlocale()** has not been called successfully, characters are classified according to the rules of the default ASCII 7-bit coded character set (see *setlocale*(3C)).

Each of the classification functions is a predicate that returns non-zero for true, zero for false.

wctype() is defined for valid character class names as defined in the current locale. *charclass* is a string identifying a generic character class for which codeset-specific type information is required. The following class names are defined in all locales: **alnum**, **alpha**, **blank**, **cntrl**, **digit**, **graph**, **lower**, **print**, **punct**, **space**, **upper**, and **xdigit**. User-defined class names may be specified if supported by the current locale as defined by **setlocale()** (see *setlocale*(3C)). **wctype()** returns a value of type **wctype_t** that can be used in a subsequent call to **iswctype()**, or **(wctype_t)-1** if *charclass* is not valid in the current locale.

The classification functions return non-zero under the following circumstances, and zero otherwise:

iswctype(*wc*,*prop***)** *wc* has the property defined by *prop*.

iswalpha(*wc***)** *wc* is a letter.

iswupper(*wc***)** *wc* is an uppercase letter.

iswlower(*wc***)** *wc* is a lowercase letter.

iswdigit(*wc***)** *wc* is a decimal digit (in ASCII: characters [0-9]).

iswxdigit(*wc***)** *wc* is a hexadecimal digit (in ASCII: characters [0-9], [A-F] or [a-f]).

iswalnum(*wc***)** *wc* is an alphanumeric (letters or digits).

iswspace(*wc***)** *wc* is a character that creates "white space" in displayed text (in ASCII: space, tab, carriage return, new-line, vertical tab, and form-feed).

iswpunct(*wc***)** *wc* is a punctuation character (in ASCII: any printing character except the space character (040), digits, letters).

iswprint(*wc***)** *wc* is a printing character.

iswgraph(*wc***)** *wc* is a visible character (in ASCII: printing characters, excluding the space character (040)).

W

iswcntrl(*wc* **)** *wc* is a control character (in ASCII: character codes less than 040 and the delete character (0177)).

If the argument to any of these functions is outside the domain of the function, the result is 0 (false).

Definitions for these functions and the types **wint_t**, **wchar_t**, and **wctype_t** are provided in the <**wchar.h**> header.

APPLICATION USAGE

The interfaces **wctype()**, **iswctype()**, **iswalnum()**, **iswalpha()**, **iswcntrl()**, **iswdigit()**, **iswgraph()**, **iswlower()**, **iswprint()**, **iswpunct()**, **iswspace()**, **iswupper()** and **iswxdigit()** are thread-safe and async-cancel-safe.

EXTERNAL INFLUENCES

Locale

The **LC_CTYPE** category determines the classification of character type.

International Code Set Support

Single-byte and multi-byte character code sets are supported.

AUTHOR

wctype() was developed by IBM, OSF, and HP.

SEE ALSO

ctype(3C), multibyte(3C), setlocale(3C), ascii(5).

STANDARDS CONFORMANCE

wctype(): XPG4

iswalnum(): XPG4

iswalpha(): XPG4

iswcntrl(): XPG4

iswctype(): XPG4

iswdigit(): XPG4

iswgraph(): XPG4

iswlower(): XPG4

iswprint(): XPG4

iswpunct(): XPG4

iswspace(): XPG4

iswupper(): XPG4

iswxdigit(): XPG4

W

NAME

wordexp(), wordfree() - perform word expansions

SYNOPSIS

```
#include <wordexp.h>
int wordexp(const char *words, wordexp_t *pwordexp, int flags);
void wordfree(wordexp_t *pwordexp);
```

DESCRIPTION

wordexp() performs word expansions and places the list of expanded words into the structure pointed to by *pwordexp*.

The *words* argument is a pointer to a string containing one or more words to be expanded. The expansions are the same as would be performed by the shell (see *sh-posix*(1)), if words were the part of a command line representing the arguments to a utility. Therefore, *words* must not contain an unquoted newline character or any of the unquoted shell special characters **|**, **&**, **;**, **<** or **>**, except in the context of shell command substitution. If *words* contains an unquoted comment character, **#**, it is treated as the beginning of a token that **wordexp()** interprets as a comment indicator, causing the remainder of *words* to be ignored.

The structure type **wordexp_t** is defined in the header <**wordexp.h**>, and includes the following members:

we_wordc	A **size_t** used to keep count of words matched by *words*.
we_wordv	A **char**** used as a pointer to a list of expanded words.
we_offs	Also a **size_t** used to indicate the number of slots to reserve at the beginning of **pwordexp->we_wordv**.

wordexp() stores the number of generated words into **pwordexp->we_wordc**. Each individual field created during field splitting or path name expansion is a separated word in the **pwordexp->we_wordv** list. The words are in order as described in shell word expansions. The first pointer after the last word pointer is a null pointer. The expansion of special parameters (such as $$ or $*) is unspecified.

It is the caller's responsibility to allocate the storage pointed to by *pwordexp*. **wordexp()** allocates other space as needed, including memory pointed to by **pwordexp->we_wordv**.

wordfree() frees any memory associated with *pwordexp* from a previous call to **wordexp()**.

The *flags* argument is used to control the behavior of **wordexp()**. The value of flags is the bitwise inclusive OR of zero or more of the following constants, which are defined in <**wordexp.h**>:

WRDE_APPEND	Append words generated to the ones from a previous call to **wordexp()**.
WRDE_DOOFFS	Make use of **pwordexp->we_offs**. If this flag is set, **pwordexp->we_offs** is used to specify how many null pointers to add to the beginning of **pwordexp->we_wordv**. In other words, **pwordexp->we_wordv** points to **pwordexp->we_offs** null pointers, followed by **pwordexp->we_wordc** word pointers, followed by a null pointer.
WRDE_NOCMD	Fail if command substitution is requested.
WRDE_REUSE	The *pwordexp* argument was passed to a previous successful call to **wordexp()**, and has not been passed to **wordfree()**. The result is the same as if the application had called **wordfree()** and then called **wordexp()** without **WRDE_REUSE**.
WRDE_SHOWERR	Do not redirect *stderr* to **/dev/null**.
WRDE_UNDEF	Report error on an attempt to expand an undefined shell variable.

The **WRDE_APPEND** flag can be used to append a new set of words to those generated by a previous call to **wordexp()**. The following rules apply when two or more calls to **wordexp()** are made with the same value of *pwordexp* and without intervening calls to **wordfree()**:

- The first call must not set **WRDE_APPEND**. All subsequent calls must set it.
- All of the calls must set **WRDE_DOOFFS**, or all must not set it.
- After the second and each subsequent call, **pwordexp->we_wordv** points to a list containing the following:

- Zero or more null pointers, as specified by **WRDE_DOOFFS** and **pwordexp->we_offs**.
- Pointers to the words that were in the **pwordexp->we_wordv** list before the call, in the same order as before.
- Pointers to the new words generated by the latest call, in the specified order.

- The count returned in **pwordexp->we_wordc** is the total number of words from all of the calls.
- The application can change any of the fields after a call to **wordexp()**, but if it does, it must reset them to the original value before a subsequent call, using the same *pwordexp* value, to **wordfree()** or **wordexp()** with the **WRDE_APPEND** or **WRDE_REUSE** flag.

If *words* contains an unquoted newline, **|**, **&**, **;**, **<**, **>**, parentheses, or curly brackets in an inappropriate context, **wordexp()** fails, and the number of expanded words is zero.

Unless **WRDE_SHOWERR** is set in *flags*, **wordexp()** redirects *stderr* to **/dev/null** for any utilities executed as a result of command substitution while expanding *words*. If **WRDE_SHOWERR** is set, **wordexp()** writes messages to *stderr* if syntax errors are detected while expanding *words*.

If **WRDE_DOOFFS** is set, **pwordexp->we_offs** has the same value for each **wordexp()** call and the **wordfree()** call using a given **wordexp()**.

APPLICATION USAGE

wordexp() and **wordfree()** are thread-safe. These interfaces are not async-cancel-safe. A cancellation point may occur when a thread is executing **wordexp()** or **wordfree()**.

RETURN VALUE

Upon successful completion, **wordexp()** returns zero; otherwise, it returns a nonzero value defined in **<wordexp.h>** to indicate the error:

WRDE_BADCHAR	One of the unquoted characters **|**, **&**, **;**, **<**, **>**, parentheses, or braces appears in *words* in an inappropriate context.
WRDE_BADVAL	Reference to undefined shell variable when **WRDE_UNDEF** is set in *flags*.
WRDE_CMDSUB	Command substitution requested when **WRDE_NOCMD** was set in *flags*.
WRDE_NOSPACE	Attempt to allocate memory failed.
WRDE_SYNTAX	Shell syntax error such as unbalanced parentheses or unterminated string.
WRDE_INTERNAL	Internal error.

If **wordexp()** returns the error value **WRDE_NOSPACE**, **pwordexp->we_wordc** and **pwordexp->we_wordv** are updated to reflect any words that were successfully expanded. In other cases, they are not modified.

AUTHOR

wordexp() and **wordfree()** were developed by OSF and HP.

SEE ALSO

sh-posix(1), fnmatch(3C), glob(3C), regexp(5).

STANDARDS CONFORMANCE

wordexp(): XPG4, POSIX.2

wordfree(): XPG4, POSIX.2

W

(ENHANCED CURSES)

NAME

wunctrl — generate printable representation of a wide character

SYNOPSIS

```
#include <curses.h>
wchar_t *wunctrl(cchar_t *wc);
```

DESCRIPTION

The **wunctrl()** function generates a wide character string that is a printable representation of the wide character *wc*.

This function also performs the following processing on the input argument:

- Control characters are converted to the ^*X* notation.
- Any rendition information is removed.

RETURN VALUE

Upon successful completion, **wunctrl()** returns the generated string. Otherwise, it returns a null pointer.

ERRORS

No errors are defined.

SEE ALSO

keyname(3X), unctrl(3X), <curses.h>.

CHANGE HISTORY

First released in X/Open Curses, Issue 4.

W

NAME

xdr - library routines for external data representation

DESCRIPTION

XDR routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Data for remote procedure calls (RPC) are transmitted using these routines.

Index to Routines

The following table lists XDR routines and the manual reference pages on which they are described:

XDR Routine	Manual Reference Page
xdr_array	*xdr_complex*(3N)
xdr_bool	*xdr_simple*(3N)
xdr_bytes	*xdr_complex*(3N)
xdr_char	*xdr_simple*(3N)
xdr_control	*xdr_admin*(3N)
xdr_destroy	*xdr_create*(3N)
xdr_double	*xdr_simple*(3N)
xdr_enum	*xdr_simple*(3N)
xdr_float	*xdr_simple*(3N)
xdr_free	*xdr_simple*(3N)
xdr_getpos	*xdr_admin*(3N)
xdr_hyper	*xdr_simple*(3N)
xdr_inline	*xdr_admin*(3N)
xdr_int	*xdr_simple*(3N)
xdr_long	*xdr_simple*(3N)
xdr_longlong_t	*xdr_simple*(3N)
xdr_opaque	*xdr_complex*(3N)
xdr_pointer	*xdr_complex*(3N)
xdr_quadruple	*xdr_simple*(3N)
xdr_reference	*xdr_complex*(3N)
xdr_setpos	*xdr_admin*(3N)
xdr_short	*xdr_simple*(3N)
xdr_sizeof	*xdr_admin*(3N)
xdr_string	*xdr_complex*(3N)
xdr_u_char	*xdr_simple*(3N)
xdr_u_hyper	*xdr_simple*(3N)
xdr_u_int	*xdr_simple*(3N)
xdr_u_long	*xdr_simple*(3N)
xdr_u_longlong_t	*xdr_simple*(3N)
xdr_u_short	*xdr_simple*(3N)
xdr_union	*xdr_complex*(3N)
xdr_vector	*xdr_complex*(3N)
xdr_void	*xdr_simple*(3N)
xdr_wrapstring	*xdr_complex*(3N)
xdrmem_create	*xdr_create*(3N)
xdrrec_create	*xdr_create*(3N)
xdrrec_endofrecord	*xdr_admin*(3N)
xdrrec_eof	*xdr_admin*(3N)
xdrrec_readbytes	*xdr_admin*(3N)
xdrrec_skiprecord	*xdr_admin*(3N)
xdrstdio_create	*xdr_create*(3N)

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

X

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

SEE ALSO

rpc(3N), xdr_admin(3N), xdr_complex(3N), xdr_create(3N), xdr_simple(3N).

X

NAME

xdr_admin, xdr_control, xdr_getpos, xdr_inline, xdrrec_endofrecord, xdrrec_eof, xdrrec_readbytes, xdrrec_skiprecord, xdr_setpos, xdr_sizeof - library routines for external data representation

SYNOPSIS

```
#include <rpc/xdr.h>
bool_t xdr_control(XDR *xdrs, int req, void *info);
u_int xdr_getpos(const XDR *xdrs);
long *xdr_inline(XDR *xdrs, const int len);
bool_t xdrrec_endofrecord(XDR *xdrs, int sendnow);
bool_t xdrrec_eof(XDR *xdrs);
int xdrrec_readbytes(XDR *xdrs, caddr_t addr, u_int nbytes);
bool_t xdrrec_skiprecord(XDR *xdrs);
bool_t xdr_setpos(XDR *xdrs, const u_int pos);
unsigned long xdr_sizeof(xdrproc_t func, void *data);
```

DESCRIPTION

XDR library routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Protocols such as remote procedure calls (RPC) use these routines to describe the format of the data.

These routines deal specifically with the management of the XDR stream.

Routines

See *rpc*(3N) for the definition of the **XDR** data structure. Note that any buffers passed to the XDR routines must be properly aligned. It is suggested that *malloc*(3C) be used to allocate these buffers or that the programmer insure that the buffer address is divisible evenly by four.

bool_t xdr_control()

A function macro to change or retrieve various information about an XDR stream. *req* indicates the type of operation and *info* is a pointer to the information. The supported values of *req*, their argument types and what they do are:

XDR_GET_BYTES_AVAIL	**xdr_bytesrec ***	return number of bytes left unconsumed in the stream and a flag indicating whether or not this is the last fragment.

u_int xdr_getpos()

A macro that invokes the get-position routine associated with the XDR stream, *xdrs*. The routine returns an unsigned integer, which indicates the position of the XDR byte stream. A desirable feature of XDR streams is that simple arithmetic works with this number, although the XDR stream instances need not guarantee this. Therefore, applications written for portability should not depend on this feature.

long *xdr_inline()

A macro that invokes the in-line routine associated with the XDR stream, *xdrs*. The routine returns a pointer to a contiguous piece of the stream's buffer; *len* is the byte length of the desired buffer. Note: pointer is cast to **long ***.

Warning: **xdr_inline()** may return **NULL** (**0**) if it cannot allocate a contiguous piece of a buffer. Therefore the behavior may vary among stream instances; it exists for the sake of efficiency, and applications written for portability should not depend on this feature.

bool_t xdrrec_endofrecord()

This routine can be invoked only on streams created by **xdrrec_create()** (see *xdr_create*(3N)). The data in the output buffer is marked as a completed record, and the output buffer is optionally written out if *sendnow* is non-zero. This routine returns **TRUE** if it succeeds, **FALSE** otherwise.

bool_t xdrrec_eof()

This routine can be invoked only on streams created by **xdrrec_create()**. After consuming the

rest of the current record in the stream, this routine returns **TRUE** if there is no more data in the stream's input buffer. It returns **FALSE** if there is additional data in the stream's input buffer.

int xdrrec_readbytes()
This routine can be invoked only on streams created by **xdrrec_create**(). It attempts to read *nbytes* bytes from the XDR stream into the buffer pointed to by *addr*. On success this routine returns the number of bytes read, **-1** on failure. A return value of **0** indicates an end of record.

bool_t xdrrec_skiprecord()
This routine can be invoked only on streams created by **xdrrec_create()** (see *xdr_create*(3N)). It tells the XDR implementation that the rest of the current record in the stream's input buffer should be discarded. This routine returns **TRUE** if it succeeds, **FALSE** otherwise.

bool_t xdr_setpos()
A macro that invokes the set position routine associated with the XDR stream *xdrs*. The parameter *pos* is a position value obtained from **xdr_getpos()**. This routine returns **TRUE** if the XDR stream was repositioned, and **FALSE** otherwise.

Warning: it is difficult to reposition some types of XDR streams, so this routine may fail with one type of stream and succeed with another. Therefore, applications written for portability should not depend on this feature.

unsigned long xdr_sizeof()

This routine returns the number of bytes required to encode *data* using the XDR filter function *func*, excluding potential overhead such as RPC headers or record markers. **0** is returned on error. This information might be used to select between transport protocols, or to determine the buffer size for various lower levels of RPC client and server creation routines, or to allocate storage when XDR is used outside of the RPC subsystem.

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

SEE ALSO
malloc(3C), rpc(3N), xdr_complex(3N), xdr_create(3N), xdr_simple(3N).

X

NAME

xdr_complex, xdr_array, xdr_bytes, xdr_opaque, xdr_pointer, xdr_reference, xdr_string, xdr_union, xdr_vector, xdr_wrapstring - library routines for external data representation

SYNOPSIS

```
#include <rpc/xdr.h>

bool_t xdr_array(XDR *xdrs, caddr_t *arrp, u_int *sizep,
    const u_int maxsize, const u_int elsize, const xdrproc_t elproc);

bool_t xdr_bytes(XDR *xdrs, char **sp, u_int *sizep, const u_int maxsize);

bool_t xdr_opaque(XDR *xdrs, caddr_t cp, const u_int cnt);

bool_t xdr_pointer(XDR *xdrs, char **objpp, u_int objsize,
    const xdrproc_t xdrobj);

bool_t xdr_reference(XDR *xdrs, caddr_t *pp, u_int size,
    const xdrproc_t proc);

bool_t xdr_string(XDR *xdrs, char **sp, const u_int maxsize);

bool_t xdr_union(XDR *xdrs, enum_t *dscmp, char *unp,
    const struct xdr_discrim *choices, const xdrproc_t (*defaultarm);

bool_t xdr_vector(XDR *xdrs, char *arrp, const u_int size,
    const u_int elsize, const xdrproc_t elproc);

bool_t xdr_wrapstring(XDR *xdrs, char **sp);
```

DESCRIPTION

XDR library routines allow C programmers to describe complex data structures in a machine-independent fashion. Protocols such as remote procedure calls (RPC) use these routines to describe the format of the data. These routines are the XDR library routines for complex data structures. They require the creation of XDR stream (see *xdr_create*(3N)).

Routines

See *rpc*(3N) for the definition of the **XDR** data structure. Note that any buffers passed to the XDR routines must be properly aligned. It is suggested that **malloc()** be used to allocate these buffers or that the programmer insure that the buffer address is divisible evenly by four.

bool_t xdr_array()
xdr_array() translates between variable-length arrays and their corresponding external representations. The parameter *arrp* is the address of the pointer to the array, while *sizep* is the address of the element count of the array; this element count cannot exceed *maxsize*. The parameter *elsize* is the size of each of the array's elements, and *elproc* is an XDR routine that translates between the array elements' C form and their external representation. If ***arrp* is null when decoding, **xdr_array()** allocates memory and ***aarp* points to it. This routine returns **TRUE** if it succeeds, **FALSE** otherwise.

bool_t xdr_bytes()
xdr_bytes() translates between counted byte strings and their external representations. The parameter *sp* is the address of the string pointer. The length of the string is located at address *sizep*; strings cannot be longer than *maxsize*. If ***sp* is null when decoding, **xdr_bytes()** allocates memory and ***sp* points to it. This routine returns **TRUE** if it succeeds, **FALSE** otherwise.

bool_t xdr_opaque()
xdr_opaque() translates between fixed size opaque data and its external representation. The parameter *cp* is the address of the opaque object, and *cnt* is its size in bytes. This routine returns **TRUE** if it succeeds, **FALSE** otherwise.

bool_t xdr_pointer()
Like **xdr_reference()** except that it serializes **NULL** pointers, whereas **xdr_reference()** does not. Thus, **xdr_pointer()** can represent recursive data structures, such as binary trees or linked lists. If ***objpp* is null when decoding, **xdr_pointer()** allocates memory and ***objpp* points to it.

bool_t xdr_reference()
xdr_reference() provides pointer chasing within structures. The parameter *pp* is the address of the pointer; *size* is the **sizeof** the structure that **pp* points to; and *proc* is an XDR procedure that

translates the structure between its C form and its external representation. If ***pp* is null when decoding, **xdr_reference()** allocates memory and ***pp* points to it. This routine returns **1** if it succeeds, **0** otherwise.

Warning: this routine does not understand **NULL** pointers. Use **xdr_pointer()** instead.

bool_t xdr_string()
xdr_string() translates between C strings and their corresponding external representations. Strings cannot be longer than *maxsize*. Note: *sp* is the address of the string's pointer. If ***sp* is null when decoding, **xdr_string()** allocates memory and ***sp* points to it. This routine returns **TRUE** if it succeeds, **FALSE** otherwise. Note: **xdr_string()** can be used to send an empty string (""), but not a NULL string.

bool_t xdr_union()
xdr_union() translates between a discriminated C **union** and its corresponding external representation. It first translates the discriminant of the union located at *dscmp*. This discriminant is always an **enum_t**. Next the union located at *unp* is translated. The parameter *choices* is a pointer to an array of **xdr_discrim** structures. Each structure contains an ordered pair of [*value*,*proc*]. If the union's discriminant is equal to the associated *value*, then the *proc* is called to translate the union. The end of the **xdr_discrim** structure array is denoted by a routine of value **NULL**. If the discriminant is not found in the *choices* array, then the *defaultarm* procedure is called (if it is not **NULL**). Returns **TRUE** if it succeeds, **FALSE** otherwise.

bool_t xdr_vector()
xdr_vector() translates between fixed-length arrays and their corresponding external representations. The parameter *arrp* is the address of the pointer to the array, while *size* is the element count of the array. The parameter *elsize* is the **sizeof** each of the array's elements, and *elproc* is an XDR routine that translates between the array elements' C form and their external representation. This routine returns **TRUE** if it succeeds, **FALSE** otherwise.

bool_t xdr_wrapstring()
A routine that calls **xdr_string(***xdrs*, *sp*, *maxuint***);** where *maxuint* is the maximum value of an unsigned integer.

Many routines, such as **xdr_array()**, **xdr_pointer()**, and **xdr_vector()** take a function pointer of type **xdrproc_t()**, which takes two arguments. **xdr_string()**, one of the most frequently used routines, requires three arguments, while **xdr_wrapstring()** only requires two. For these routines, **xdr_wrapstring()** is desirable. This routine returns **TRUE** if it succeeds, **FALSE** otherwise.

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

SEE ALSO

rpc(3N), xdr_admin(3N), xdr_create(3N), xdr_simple(3N).

X

NAME

xdr_create, xdr_destroy, xdrmem_create, xdrrec_create, xdrstdio_create - library routines for external data representation stream creation

SYNOPSIS

```
#include <rpc/xdr.h>

void xdr_destroy(XDR *xdrs);

void xdrmem_create(XDR *xdrs, const caddr_t addr, const u_int size,
    const enum xdr_op op);

void xdrrec_create(XDR *xdrs, const u_int sendsz, const u_int recvsz,
    const caddr_t handle, const int (*readit)(const void *read_handle,
    char *buf, const int len),
    const int (*writeit)(const void *write_handle, const char *buf,
    const int len));

void xdrstdio_create(XDR *xdrs, FILE *file, const enum xdr_op op);
```

DESCRIPTION

XDR library routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Protocols such as remote procedure calls (RPC) use these routines to describe the format of the data.

These routines deal with the creation of XDR streams. XDR streams have to be created before any data can be translated into XDR format.

Routines

See *rpc*(3N) for the definition of the **XDR**, **CLIENT**, and **SVCXPRT** data structures. Note that any buffers passed to the XDR routines must be properly aligned. It is suggested that *malloc*(3C) be used to allocate these buffers or that the programmer insure that the buffer address is divisible evenly by four.

void xdr_destroy()

A macro that invokes the destroy routine associated with the XDR stream, *xdrs*. Destruction usually involves freeing private data structures associated with the stream. Using *xdrs* after invoking **xdr_destroy()** is undefined.

void xdrmem_create()

This routine initializes the XDR stream object pointed to by *xdrs*. The stream's data is written to, or read from, a chunk of memory at location *addr* whose length is no less than *size* bytes long. The *op* determines the direction of the XDR stream (either **XDR_ENCODE**, **XDR_DECODE**, or **XDR_FREE**).

void xdrrec_create()

This routine initializes the read-oriented XDR stream object pointed to by *xdrs*. The stream's data is written to a buffer of size *sendsz*; a value of **0** indicates the system should use a suitable default. The stream's data is read from a buffer of size *recvsz*; it too can be set to a suitable default by passing a **0** value. When a stream's output buffer is full, *writeit* is called. Similarly, when a stream's input buffer is empty, *readit* is called. The behavior of these two routines is similar to the system calls **read()** and **write()** (see *read*(2) and *write*(2), respectively), except that an appropriate handle (*read_handle* or *write_handle*) is passed to the former routines as the first parameter instead of a file descriptor. Note: the XDR stream's *op* field must be set by the caller.

Warning: this XDR stream implements an intermediate record stream. Therefore there are additional bytes in the stream to provide record boundary information.

void xdrstdio_create()

This routine initializes the XDR stream object pointed to by *xdrs*. The XDR stream data is written to, or read from, the standard I/O stream *file*. The parameter *op* determines the direction of the XDR stream (either **XDR_ENCODE**, **XDR_DECODE**, or **XDR_FREE**).

Warning: the destroy routine associated with such XDR streams calls **fflush()** on the *file* stream, but never **fclose()** (see *fclose*(3S)).

Failure of any of these functions can be detected by first initializing the *x_ops* field in the **XDR** structure (*xdrs* **->** *x_ops*) to **NULL** before calling the **xdr*_create()** function. After the return from the **xdr*_create()** function, if the *x_ops* field is still **NULL**, the call has failed. If the *x_ops* field contains some other value, the call can be assumed to have succeeded.

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

SEE ALSO

read(2), write(2), malloc(3C), rpc(3N), xdr_admin(3N), xdr_complex(3N), xdr_simple(3N), fclose(3S).

X

NAME

xdr_simple, xdr_bool, xdr_char, xdr_double, xdr_enum, xdr_float, xdr_free, xdr_hyper, xdr_int, xdr_long, xdr_longlong_t, xdr_quadruple, xdr_short, xdr_u_char, xdr_u_hyper, xdr_u_int, xdr_u_long, xdr_u_longlong_t, xdr_u_short, xdr_void - library routines for external data representation

SYNOPSIS

```
#include <rpc/xdr.h>
bool_t xdr_bool(XDR *xdrs, bool_t *bp);
bool_t xdr_char(XDR *xdrs, char *cp);
bool_t xdr_double(XDR *xdrs, double *dp);
bool_t xdr_enum(XDR *xdrs, enum_t *ep);
bool_t xdr_float(XDR *xdrs, float *fp);
void xdr_free(xdrproc_t proc, char *objp);
bool_t xdr_hyper(XDR *xdrs, longlong_t *llp);
bool_t xdr_int(XDR *xdrs, int *ip);
bool_t xdr_long(XDR *xdrs, long *lp);
bool_t xdr_longlong_t(XDR *xdrs, longlong_t *llp);
bool_t xdr_quadruple(XDR *xdrs, long double *pq);
bool_t xdr_short(XDR *xdrs, short *sp);
bool_t xdr_u_char(XDR *xdrs, unsigned char *ucp);
bool_t xdr_u_hyper(XDR *xdrs, u_longlong_t *ullp);
bool_t xdr_u_int(XDR *xdrs, unsigned *up);
bool_t xdr_u_long(XDR *xdrs, unsigned long *ulp);
bool_t xdr_u_longlong_t(XDR *xdrs, u_longlong_t *ullp);
bool_t xdr_u_short(XDR *xdrs, unsigned short *usp);
bool_t xdr_void(void);
```

DESCRIPTION

XDR library routines allow C programmers to describe simple data structures in a machine-independent fashion. Protocols such as remote procedure calls (RPC) use these routines to describe the format of the data.

These routines require the creation of XDR streams (see *xdr_create*(3N)).

Routines

See *rpc*(3N) for the definition of the **XDR** data structure. Note that any buffers passed to the XDR routines must be properly aligned. It is suggested that *malloc*(3C) be used to allocate these buffers or that the programmer insure that the buffer address is divisible evenly by four.

bool_t xdr_bool()
xdr_bool() translates between booleans (C integers) and their external representations. When encoding data, this filter produces values of either **1** or **0**. This routine returns **TRUE** if it succeeds, **FALSE** otherwise.

bool_t xdr_char()
xdr_char() translates between C characters and their external representations. This routine returns **TRUE** if it succeeds, **FALSE** otherwise. Note: encoded characters are not packed, and occupy 4 bytes each. For arrays of characters, it is worthwhile to consider **xdr_bytes()**, **xdr_opaque()**, or **xdr_string()** (see *xdr_complex*(3N)).

bool_t xdr_double()
xdr_double() translates between C **double** precision numbers and their external representations. This routine returns **TRUE** if it succeeds, **FALSE** otherwise.

bool_t xdr_enum()
xdr_enum() translates between C **enum** (actually integers) and their external representations. This routine returns **TRUE** if it succeeds, **FALSE** otherwise.

bool_t xdr_float()
xdr_float() translates between C **float**s and their external representations. This routine returns **TRUE** if it succeeds, **FALSE** otherwise.

void xdr_free()
Generic freeing routine. The first argument is the XDR routine for the object being freed. The second argument is a pointer to the object itself. Note: the pointer passed to this routine is not freed, but what it points to is freed (recursively, depending on the XDR routine).

bool_t xdr_hyper()
xdr_hyper() translates between ANSI C **long long** integers and their external representations. This routine returns **TRUE** if it succeeds, **FALSE** otherwise.

bool_t xdr_int()
xdr_int() translates between C integers and their external representations. This routine returns **TRUE** if it succeeds, **FALSE** otherwise.

bool_t xdr_long()
xdr_long() translates between C **long** integers and their external representations. This routine returns **TRUE** if it succeeds, **FALSE** otherwise.

bool_t xdr_longlong_t()
xdr_longlong_t() translates between ANSI C **long long** integers and their external representations. This routine returns **TRUE** if it succeeds, **FALSE** otherwise. This routine is identical to **xdr_hyper()**.

bool_t xdr_quadruple()
xdr_quadruple() translates between IEEE quadruple precision floating point numbers and their external representations. This routine returns **TRUE** if it succeeds, **FALSE** otherwise.

bool_t xdr_short()
xdr_short() translates between C **short** integers and their external representations. This routine returns **TRUE** if it succeeds, **FALSE** otherwise.

bool_t xdr_u_char()
xdr_u_char() translates between **unsigned** C characters and their external representations. This routine returns **TRUE** if it succeeds, **FALSE** otherwise.

bool_t xdr_u_hyper()
xdr_u_hyper() translates between unsigned ANSI C **long long** integers and their external representations. This routine returns **TRUE** if it succeeds, **FALSE** otherwise.

bool_t xdr_u_int()
A filter primitive that translates between a C **unsigned** integer and its external representation. This routine returns **TRUE** if it succeeds, **FALSE** otherwise.

bool_t xdr_u_long()
xdr_u_long() translates between C **unsigned long** integers and their external representations. This routine returns **TRUE** if it succeeds, **FALSE** otherwise.

bool_t xdr_u_longlong_t()
xdr_u_longlong_t() translates between unsigned ANSI C **long long** integers and their external representations. This routine returns **TRUE** if it succeeds, **FALSE** otherwise. This routine is identical to **xdr_u_hyper()**.

X **bool_t xdr_u_short()**
xdr_u_short() translates between C **unsigned short** integers and their external representations. This routine returns **TRUE** if it succeeds, **FALSE** otherwise.

bool_t xdr_void(void);
This routine always returns **TRUE**. It may be passed to RPC routines that require a function parameter, where nothing is to be done.

MULTITHREAD USAGE
Thread Safe: **Yes**

Cancel Safe: **Yes**
Fork Safe: **No**
Async-cancel Safe: **No**
Async-signal Safe: **No**

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

SEE ALSO

malloc(3C), rpc(3N), xdr_admin(3N), xdr_complex(3N), xdr_create(3N).

X

NAME

y0(), y1(), yn() - Bessel functions of the second kind

SYNOPSIS

```
#include <math.h>
double y0(double x);
double y1(double x);
double yn(int n, double x);
```

DESCRIPTION

y0() and **y1()** return the 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 greater than zero.

To use these functions, compile either with the default **-Ae** option or with the **-Aa** and **-D_HPUX_SOURCE** options. Make sure your program includes <**math.h**>. Link in the math library by specifying **-lm** on the compiler or linker command line.

RETURN VALUE

If *x* is negative or zero, **y0()**, **y1()**, and **yn()** return −**HUGE_VAL**.

If *x* is NaN, **y0()**, **y1()**, and **yn()** return NaN.

If the correct result after rounding would be smaller in magnitude than **MINDOUBLE**, **y0()**, **y1()**, and **yn()** return zero.

If the correct result would overflow, **y0()**, **y1()**, and **yn()** return −**HUGE_VAL**.

ERRORS

No errors are defined.

SEE ALSO

j0(3M), math(5), values(5).

M. Abramowitz and I. Stegun, *Handbook of Mathematical Functions* (New York: Dover Publications, 1972).

STANDARDS CONFORMANCE

y0(): SVID3, XPG4.2

y1(): SVID3, XPG4.2

yn(): SVID3, XPG4.2

y

NAME

ypclnt(), yp_all(), yp_bind(), yp_first(), yp_get_default_domain(), yp_master(), yp_match(), yp_next(), yp_order(), yp_unbind(), yperr_string(), ypprot_err() - Network Information Service client interface

SYNOPSIS

cc [*flag*...] *file*... **-lnsl** [*library*...]

```
#include <rpcsvc/ypclnt.h>

#include <sys/types.h>
#include <rpc/rpc.h>
#include <rpcsvc/yp_prot.h>

int yp_all(
     char *indomain,
     char *inmap,
     struct ypall_callback *incallback
);

int yp_bind(char *indomain);

int yp_first(
     char *indomain,
     char *inmap,
     char **outkey,
     int *outkeylen,
     char **outval,
     int *outvallen
);

int yp_get_default_domain(char **outdomain);

int yp_master(
     char *indomain,
     char *inmap,
     char **outmaster
);

int yp_match(
     char *indomain,
     char *inmap,
     char *inkey,
     int inkeylen,
     char **outval,
     int *outvallen
);

int yp_next(
     char *indomain,
     char *inmap,
     char *inkey,
     int inkeylen,
     char **outkey,
     int *outkeylen,
     char **outval,
     int *outvallen
);

int yp_order(
     char *indomain,
     char *inmap,
     unsigned long *outorder
);

void yp_unbind(char *indomain);

char *yperr_string(int incode);
```

```
int ypprot_err(unsigned int incode);
```

Remarks

The Network Information Service (NIS) was formerly known as Yellow Pages (yp). Although the name has changed, the functionality of the service remains the same.

DESCRIPTION

These functions provide an interface to the Network Information Service (NIS) network-lookup service. Refer to *ypfiles*(4) and *ypserv*(1M) for an overview of the NIS, including the definitions of *map* and NIS *domain*, and a description of the various servers, databases, and commands comprising the NIS.

Input parameter names begin with **in**; output parameter names begin with **out**. Output parameters of type **char**** should be the addresses of uninitialized character pointers. Memory is allocated by the NIS client package using **malloc()** and can be freed if the user code has no continuing need for it (see *malloc*(3C)). For each *outkey* and *outval*, two extra bytes of memory are allocated at the end that contain new-line and null (in that order), but these two bytes are not reflected in *outkeylen* and *outvallen*. The *indomain* and *inmap* strings must be non-null and null-terminated. String parameters that are accompanied by a length parameter cannot be null, but can point to null strings with a length parameter of zero. Counted strings need not be null-terminated.

The NIS lookup calls require a map (database) name and a NIS domain name. The client process should know the name of the map of interest. Client processes should obtain the host's NIS domain by calling **yp_get_default_domain()** and use the returned *outdomain* as the *indomain* parameter to subsequent NIS calls.

To use the NIS services, the client process must be "bound" to an NIS server that serves the appropriate NIS domain using **yp_bind()**. Binding does not have to occur explicitly by user code. Rather, it occurs automatically whenever a NIS lookup function is called. **yp_bind()** can be called directly for processes that use a backup strategy (such as a local file) when NIS services are not available.

Each binding allocates (uses up) one client process socket descriptor. Each bound NIS domain costs one socket descriptor. However, multiple requests to the same NIS domain use that same descriptor. **yp_unbind()** is available at the client interface for processes that explicitly manage their socket descriptors while accessing multiple NIS domains. The call to **yp_unbind** makes the NIS domain **unbound** and frees all per-process and per-node resources used to bind it.

If an RPC failure results when using a binding, that NIS domain is unbound automatically. The ypclnt layer then continues retrying until the operation succeeds, provided **ypbind** is running (see *ypserv*(1M)) and either:

1. the client process cannot bind a server for the proper NIS domain, or
2. RPC requests to the server fail.

If an error is not RPC-related, if **ypbind** is not running, or if a bound **ypserv** process returns any answer (success or failure), the ypclnt layer returns control to the user code with either an error code or with a success code and any results (see **ypbind** in *ypserv*(1M)).

Operational Behavior

yp_match() Returns the value associated with a passed key. This key must be exact; no pattern matching is available.

yp_first() Returns the first key-value pair from the named map in the named NIS domain.

yp_next() Returns the next key-value pair in a named map. To obtain the second key-value pair, the *inkey* parameter should be the *outkey* returned from an initial call to **yp_first()**. To obtain the (*n* + *1*)th key-value pair, the *inkey* value should be the *outkey* value from the *n*th call to **yp_next()**.

The concepts of first and next are particular to the structure of the NIS map being processed. No relation in retrieval order exists to either the lexical order within any original ASCII file or to any obvious numerical sorting order on the keys, values, or key-value pairs. The only ordering guarantee is that if the **yp_first()** function is called on a particular map and the **yp_next()** function is called repeatedly on the same map at the same server until the call fails with an error of YPERR_NOMORE, every entry in the database is retrieved exactly once. If the same sequence of operations is performed on the same map at the same server, the entries are retrieved in the same order.

Under conditions of heavy server load or server failure, the NIS domain may become unbound and bind again (perhaps to a different server) while a client is running. This process can cause a break in one of the enumeration (retrieval) rules: specific entries may be seen twice by the client or not at all. This approach protects the client from error messages that would otherwise be returned in the midst of the enumeration.

yp_all() describes a better solution to enumerating all entries in a map.

yp_all() Provides a way to transfer an entire map from server to client in a single request using TCP (rather than UDP as with other functions in this package). The entire transaction occurs as a single RPC request and response. You can use **yp_all()** like any other NIS procedure by identifying the map in the normal manner and supplying the name of a function called to process each key-value pair within the map. A return from the call to **yp_all()** occurs only when the transaction is completed (either successfully or unsuccessfully) or the *foreach* function decides it does not want any more key-value pairs.

The third parameter to **yp_all()** is:

```
struct ypall_callback *incallback {
    int (*foreach)();
    char *data;
};
```

The function **foreach()** is called as follows:

```
foreach(
    int instatus;
    char *inkey;
    int inkeylen;
    char *inval;
    int invallen;
    char *indata;
);
```

Where:

instatus Holds one of the return status values defined in <**rpcsvc/yp_prot.h**>: either **YP_TRUE** or an error code (see **ypprot_err()** below, for a function that converts a NIS protocol error code to a ypclnt layer error code, as defined in <**rpcsvc/ypclnt.h**>).

inkey *inval* The key and value parameters are somewhat different than defined in the SYNOPSIS section above. First, the memory pointed to by *inkey* and *inval* is private to **yp_all()**, and is overwritten with the arrival of each new key-value pair. Therefore, **foreach()** should do something useful with the contents of that memory, but it does not own the memory. Key and value objects presented to the **foreach()** look exactly as they do in the server's map. Therefore, if they were not newline-terminated or null-terminated in the map, they will not be terminated with newline or null characters here, either.

indata Is the contents of the *incallback–> data* element passed to **yp_all()** The *data* element of the callback structure can share state information between **foreach()** and the mainline code. Its use is optional, and no part of the NIS client package inspects its contents. Cast it to something useful or ignore it as appropriate.

The **foreach()** function is Boolean. It should return zero to indicate it needs to be called again for further received key-value pairs, or non-zero to stop the flow of key-value pairs. If **foreach()** returns a non-zero value, it is not called again and the functional value of **yp_all()** is then 0.

yp_order() Returns the order number for a map.

yp_master() Returns the host name of the master NIS server for a map.

yperr_string()
Returns a pointer to an error message string that is null-terminated, but contains no period or newline.

ypprot_err() Takes an NIS protocol error code as input and returns a ypclnt layer error code that can be used as input to **yperr_string()**

MULTITHREAD USAGE

Thread Safe: **Yes**
Cancel Safe: **Yes**
Fork Safe: **No**
Async-cancel Safe: **No**
Async-signal Safe: **No**

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

RETURN VALUE

All functions in this package of type **int** return 0 if the requested operation is successful or one of the following errors if the operation fails.

[YPERR_ACCESS]	database access violation
[YPERR_BADARGS]	args to function are bad
[YPERR_BADDB]	NIS map is defective
[YPERR_BUSY]	database busy
[YPERR_DOMAIN]	cannot bind to server on this NIS domain
[YPERR_KEY]	no such key in map
[YPERR_MAP]	no such map in server's NIS domain
[YPERR_NODOM]	local NIS domain name not set
[YPERR_NOMORE]	no more records in map
[YPERR_PMAP]	cannot communicate with portmap
[YPERR_RESRC]	resource allocation failure
[YPERR_RPC]	RPC failure – NIS domain has been unbound
[YPERR_VERS]	NIS client/server version mismatch: the NIS server bound to uses Version 1 protocol, so it does not provide **yp_all()** functionality
[YPERR_YPBIND]	cannot communicate with ypbind
[YPERR_YPERR]	internal NIS server or client error
[YPERR_YPSERV]	cannot communicate with ypserv

AUTHOR

ypclnt() was developed by Sun Microsystems, Inc.

SEE ALSO

domainname(1), rpcinfo(1M), ypserv(1M), ypfiles(4).

y

NAME

yppasswd() - update user password in Network Information Service

SYNOPSIS

cc [*flag*...] *file*... **-lnsl** [*library*...]

```
#include <pwd.h>
#include <rpcsvc/yppasswd.h>
int yppasswd(char *oldpass, struct passwd *newpw);
```

DESCRIPTION

If *oldpass* is the old, unencrypted user password, this routine replaces the password entry with the encrypted *newpw*.

RPC Info

Program number:

YPPASSWDPROG

XDR routines:

```
xdr_yppasswd(xdrs, yp)
   XDR *xdrs;
   struct yppasswd *yp;

xdr_passwd(xdrs, pw)
   XDR *xdrs;
   struct passwd *pw;
```

Procs:

YPPASSWDPROC_UPDATE

Takes *struct yppasswd* as an argument; returns an integer. Behaves the same as the *yppasswd()* function. Uses UNIX authentication.

Versions:

YPPASSWDVERS

Structures:

```
struct yppasswd {
   char *oldpass;        /* old (unencrypted) password */
   struct passwd newpw;  /* new pw structure */
};
```

MULTITHREAD USAGE

Thread Safe:	**Yes**
Cancel Safe:	**Yes**
Fork Safe:	**No**
Async-cancel Safe:	**No**
Async-signal Safe:	**No**

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

RETURN VALUE

yppasswd() returns 0 if successful and –1 if an error occurs.

AUTHOR

yppasswd() was developed by Sun Microsystems, Inc.

SEE ALSO
yppasswd(1), yppasswdd(1M).

NAME

ypupdate - changes NIS information

SYNOPSIS

cc [*flag*...] *file*... **-lnsl** [*library*...]

```
#include <rpcsvc/ypclnt.h>
yp_update (domain, map, ypop, key, keylen, data, datalen)
char *domain;
char *map;
unsigned ypop;
char *key;
int keylen;
char *data;
int datalen;
```

Remarks

The Network Information Service (NIS) was formerly known as Yellow Pages (yp). Although the name has changed, the functionality of the service remains the same.

DESCRIPTION

The routine **yp_update()** is used to make changes to the Network Information Service (NIS) database. The syntax is the same as that of **yp_match()** (see ypclnt(3C)) except for the extra parameter ypop which may take on one of four values.

YPOP_CHANGE The data associated with the key will be changed to the new value. If the key is not found in the database, then **yp_update()** returns **YPERR_KEY.**

YPOP_INSERT The key-value pair will be inserted into the database. The error **YPERR_KEY** is returned if the key already exists in the database.

YPOP_STORE The key-value pair will be stored into the database without concern for whether it already exists or not.

YPOP_DELETE The key-value pair will be deleted from the database.

This routine depends upon secure RPC and will not work unless the network is running secure RPC.

MULTITHREAD USAGE

Thread Safe: **Yes**
Cancel Safe: **Yes**
Fork Safe: **No**
Async-cancel Safe: **No**
Async-signal Safe: **No**

These functions can be called safely in a multithreaded environment. They may be cancellation points in that they call functions that are cancel points.

In a multithreaded environment, these functions are not safe to be called by a child process after **fork()** and before **exec()**. These functions should not be called by a multithreaded application that support asynchronous cancellation or asynchronous signals.

AUTHOR

The **yp_update** routine was developed by Sun Microsystems, Inc.

SEE ALSO

ypclnt(3C).

y