ULTRIX

digital

Reference Pages
Section 3: Library Routines

Order Number: AA–LY16B–TE

ULTRIX

Reference Pages Section 3: Library Routines

Order Number: AA-LY16B-TE
June 1990

Product Version: ULTRIX Version 4.0 or higher

This manual describes the routines available in the ULTRIX libraries for programmers on both RISC and VAX platforms.

**digital equipment corporation
maynard, massachusetts**

About Reference Pages

The *ULTRIX Reference Pages* describe commands, system calls, routines, file formats, and special files for RISC and VAX platforms.

Sections

The reference pages are divided into eight sections according to topic. Within each section, the reference pages are organized alphabetically by title, except Section 3, which is divided into subsections. Each section and most subsections have an introductory reference page called `intro` that describes the organization and anything unique to that section.

Some reference pages carry a one- to three-letter suffix after the section number, for example, `scan`(1mh). The suffix indicates that there is a ''family'' of reference pages for that utility or feature. The Section 3 subsections all use suffixes and other sections may also have suffixes.

Following are the sections that make up the *ULTRIX Reference Pages*.

Section 1: Commands

This section describes commands that are available to all ULTRIX users. Section 1 is split between two binders. The first binder contains reference pages for titles that fall between A and L. The second binder contains reference pages for titles that fall between M and Z.

Section 2: System Calls

This section defines system calls (entries into the ULTRIX kernel) that are used by all programmers. The introduction to Section 2, `intro`(2), lists error numbers with brief descriptions of their meanings. The introduction also defines many of the terms used in this section.

Section 3: Routines

This section describes the routines available in ULTRIX libraries. Routines are sometimes referred to as subroutines or functions.

Section 4: Special Files

This section describes special files, related device driver functions, databases, and network support.

Section 5: File Formats

This section describes the format of system files and how the files are used. The files described include assembler and link editor output, system accounting, and file system formats.

Section 6: Games

The reference pages in this section describe the games that are available in the unsupported software subset. The reference pages for games are in the document *Reference Pages for Unsupported Software.*

Section 7: Macro Packages and Conventions

This section contains miscellaneous information, including ASCII character codes, mail addressing formats, text formatting macros, and a description of the root file system.

Section 8: Maintenance

This section describes commands for system operation and maintenance.

Platform Labels

The *ULTRIX Reference Pages* contain entries for both RISC and VAX platforms. Pages that have no platform label beside the title apply to both platforms. Reference pages that apply only to RISC platforms have a ''RISC'' label beside the title and the VAX-only reference pages that apply only to VAX platforms are likewise labeled with ''VAX.'' If each platform has the same command, system call, routine, file format, or special file, but functions differently on the different platforms, both reference pages are included, with the RISC page first.

Reference Page Format

Each reference page follows the same general format. Common to all reference pages is a title consisting of the name of a command or a descriptive title, followed by a section number; for example, `date`(1). This title is used throughout the documentation set.

The headings in each reference page provide specific information. The standard headings are:

Name	Provides the name of the entry and gives a short description.
Syntax	Describes the command syntax or the routine definition. Section 5 reference pages do not use the Syntax heading.
Description	Provides a detailed description of the entry's features, usage, and syntax variations.
Options	Describes the command-line options.
Restrictions	Describes limitations or restrictions on the use of a command or routine.
Examples	Provides examples of how a command or routine is used.

Return Values	Describes the values returned by a system call or routine. Used in Sections 2 and 3 only.
Diagnostics	Describes diagnostic and error messages that can appear.
Files	Lists related files that are either a part of the command or used during execution.
Environment	Describes the operation of the system call or routine when compiled in the POSIX and SYSTEM V environments. If the environment has no effect on the operation, this heading is not used. Used in Sections 2 and 3 only.
See Also	Lists related reference pages and documents in the ULTRIX documentation set.

Conventions

The following documentation conventions are used in the reference pages.

%	The default user prompt is your system name followed by a right angle bracket. In this manual, a percent sign (%) is used to represent this prompt.
#	A number sign is the default superuser prompt.
`user input`	This bold typeface is used in interactive examples to indicate typed user input.
`system output`	This typeface is used in text to indicate the exact name of a command, routine, partition, pathname, directory, or file. This typeface is also used in interactive examples to indicate system output and in code examples and other screen displays.
UPPERCASE lowercase	The ULTRIX system differentiates between lowercase and uppercase characters. Literal strings that appear in text, examples, syntax descriptions, and function definitions must be typed exactly as shown.
rlogin	This typeface is used for command names in the Syntax portion of the reference page to indicate that the command is entered exactly as shown. Options for commands are shown in bold wherever they appear.
filename	In examples, syntax descriptions, and routine definitions, italics are used to indicate variable values. In text, italics are used to give references to other documents.
[ ]	In syntax descriptions and routine definitions, brackets indicate items that are optional.
{ | }	In syntax descriptions and routine definitions, braces enclose lists from which one item must be chosen. Vertical bars are used to separate items.

. . .	In syntax descriptions and routine definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.
. . .	A vertical ellipsis indicates that a portion of an example that would normally be present is not shown.
`cat(1)`	Cross-references to the *ULTRIX Reference Pages* include the appropriate section number in parentheses. For example, a reference to `cat(1)` indicates that you can find the material on the `cat` command in Section 1 of the reference pages.

Online Reference Pages

The ULTRIX reference pages are available online if installed by your system administrator. The `man` command is used to display the reference pages as follows:

To display the `ls(1)` reference page:

```
% man ls
```

To display the `passwd(1)` reference page:

```
% man passwd
```

To display the `passwd(5)` reference page:

```
% man 5 passwd
```

To display the Name lines of all reference pages that contain the word ''passwd'':

```
% man -k passwd
```

To display the introductory reference page for the family of 3xti reference pages:

```
% man 3xti intro
```

Users on ULTRIX workstations can display the reference pages using the unsupported `xman` utility if installed. See the `xman(1X)` reference page for details.

Reference Pages for Unsupported Software

The reference pages for the optionally installed, unsupported ULTRIX software are in the document *Reference Pages for Unsupported Software.*

Insert tabbed divider here.
Then discard this sheet.

Name

intro – introduction to library functions

Description

This section describes functions that may be found in various libraries. The library functions are those other than the functions that directly invoke ULTRIX system primitives, described in section 2. Section 3 has the libraries physically grouped together. The functions described in this section are grouped into various libraries:

Sections 3 and 3s

The (3) functions are the standard C library functions. The C library also includes all the functions described in Section 2. These routines are included for compatibility with other systems. In particular, a number of system call interfaces provided in 4.2BSD have been included for source code compatibility. The (3s) functions comprise the standard I/O library. Together with the (3n), (3xti), (3yp) and (3) routines, these functions constitute library *libc*, which is automatically loaded by the C compiler (cc), the Pascal compiler (pc), and the FORTRAN compiler (f77). (FORTRAN and Pascal are optional and may not be installed on your system.) Declarations for these functions may be obtained from the include file, <stdio.h>. The link editor `ld`(1) searches this library under the –lc option. Declarations for some of these functions may be obtained from include files indicated on the appropriate pages.

VAX Only

On VAX machines, the GFLOAT version of *libc* is used when you use the `cc`(1) command with the **–Mg** option, or you use the `ld`(1) command with the **–lcg** option. The GFLOAT version of *libc* must be used with modules compiled with `cc`(1) using the **–Mg** option.

Note that neither the compiler nor the linker `ld`(1) can detect when mixed double floating point types are used, and your program may produce erroneous results if this occurs on a VAX machine.

Section 3cur

The (3cur) library routines make up the X/Open curses library. These routines are different from the 4.2BSD curses routines contained in Section 3x.

Section 3f

The (3f) functions are all functions callable from FORTRAN. These functions perform the same jobs as do the (3) functions. An unsupported FORTRAN compiler, f77, is included in the VAX distribution. FORTRAN is available as a layered product on both VAX and RISC machines.

Section 3int

The (3int) functions assist programs in supporting native language interfaces. They are found in the internationalization library *libi*.

Section 3krb

The library of routines for the Kerberos authentication service. These routines support the authentication of commonly networked applications across machine boundaries in a distributed network.

Section 3m

The (3m) functions constitute the math library, *libm*. They are automatically loaded as needed by the Pascal compiler (pc) and the FORTRAN compiler (f77). The link editor searches this library under the **–lm** option. Declarations for these functions may be obtained from the include file, < math.h >.

VAX Only

On VAX machines, the GFLOAT version of *libm* is used when you use the `ld`(1) command with the **–lcg** option. Note that you must use the GFLOAT version of *libm* with modules compiled using the `cc`(1) command with the **–Mg** option.

Note that neither the compiler nor the linker `ld`(1) can detect when mixed double floating point types are used, and the program may produce erroneous results if this occurs on a VAX machine.

Section 3ncs

This section describes the NCS (Network Computing System) library routines. The Title, Name, and See Also sections of the NCS reference pages do not contain the dollar ($) sign in the command names and library routines. The actual NCS commands and library routines do contain the dollar ($) sign.

Section 3n

These functions constitute the internet network library,

Section 3x

Various specialized libraries have not been given distinctive captions. Files in which such libraries are found are named on appropriate pages.

Section 3xti

The X/Open Transport Interface defines a transport service interface that is independent of any specific transport provider. The interface is provided by way of a set of library functions for the C programming language.

Section 3yp

These functions are specific to the Yellow Pages (YP) service.

Environmental Compatibility

The libraries in Sections 3, 3m, and 3s contain System V and POSIX compatibility features that are available to general ULTRIX programs. This compatibility sometimes conflicts with features already present in ULTRIX. That is, the function performed may be slightly different in the System V or POSIX environment. These features are provided for applications that are being ported from System V or written for a POSIX environment.

The descriptions in these sections include an ENVIRONMENT section to describe any differences in function between System V or POSIX and the standard C runtime library.

The System V compatibility features are not contained in the standard C runtime library. To get System V-specific behavior, you must specify that the System V environment is to be used in compiling and linking programs. You can do this in one of two ways:

1. Using the **–YSYSTEM_FIVE** option for the `cc` command.

2. Globally setting the environment variable PROG_ENV to SYSTEM_FIVE. If you are using the C shell, you would execute the following line, or include it in your .login file:

```
setenv PROG_ENV SYSTEM_FIVE
```

If you are using the Bourne shell, you would execute the following line, or include it in your .profile file:

```
PROG_ENV=SYSTEM_FIVE ; export PROG_ENV
```

In both cases, the `cc`(1) command defines the preprocessor symbol SYSTEM_FIVE, so that the C preprocessor, `/lib/cpp`, will select the System V version of various data structures and symbol definitions.

In addition, if `cc`(1) invokes `ld`(1), the library libcV.a (the System V version of the Standard C library) is searched before libc.a to resolve references to the System-V-specific routines. Also, if **–lm** is specified on either the `cc`(1) or the `ld`(1) command line, then the System V version of the math library will be used instead of the regular ULTRIX math library.

The POSIX compatibility features are included in the library libcP.a, so the only special action needed is to specify **-YPOSIX** on the `cc`(1) command line or set the environment variable PROG_ENV to POSIX. Either action will cause the `cc`(1) command to define the preprocessor symbol POSIX and search the POSIX library.

Files

/usr/lib/libc.a	
/usr/lib/lib_cg.a	(VAX only)
/usr/lib/libm.a	
/usr/lib/libc_p.a	(VAX only)
/usr/lib/m_g.a	(VAX only)
/usr/lib/libm_p.a	(VAX only)

Diagnostics

Functions in the math library (3m) may return conventional values when the function is undefined for the given arguments or when the value is not representable. In these cases the external variable *errno* is set to the value EDOM (domain error) or ERANGE (range error). For further information, see `intro`(2). The values of EDOM and ERANGE are defined in the include file <math.h>.

See Also

cc(1), ld(1), nm(1), intro(2) intro(3), intro(3s), intro(3f), intro(3m), intro(3n)

Name

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

Syntax

long a64l (s)
char *s;

char *l64a (l)
long l;

Description

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

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

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

The `l64a` subroutine 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.

Restrictions

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

Name

abort – generate an illegal instruction fault

Syntax

#include <stdlib.h>

void abort()

Description

The `abort` subroutine executes an instruction which is illegal in user mode. This causes a signal that normally terminates the process with a core dump, which may be used for debugging.

Diagnostics

Illegal instruction – core dumped
– Bourne shell.

Illegal instruction (core dumped)
– C shell.

Environment

When your program is compiled using the System V or POSIX environment, `abort` closes open files before aborting the process with an IOT fault.

Restrictions

The `abort` function does not flush standard I/O buffers. Use `fflush`(3s). For further information, see `fclose`(3s).

See Also

adb(1), exit(2), sigvec(2), fclose(3s)

Name

abs, labs – integer absolute value

Syntax

#include <stdlib.h>
#include <stdlib.h>

long labs(*i*)
long *i*;

int abs(*i*)
int *i*;

long labs(*i*)
long *i*;

Description

The `abs` and `labs` functions return the absolute value of their integer operand. The labs function does the same for a long int.

Restrictions

Applying the `abs` or `labs` function to the most negative integer generates a result which is the most negative integer. That is,

```
abs(0x80000000)
```

returns 0x80000000 as a result.

See Also

floor(3m)

alarm (3)

Name

alarm – schedule signal after specified time

Syntax

#include <unistd.h>

unsigned alarm(*seconds***)**
unsigned *seconds***;**

Description

The `alarm` subroutine causes signal SIGALRM, see `signal`(3), to be sent to the invoking process in a number of seconds given by the argument. Unless caught or ignored, the signal terminates the process.

The `alarm` requests are not stacked. Successive calls reset the alarm clock. If the argument is 0, any `alarm` request is canceled. Because of scheduling delays, resumption of execution of when the signal is caught may be delayed an arbitrary amount. The longest specifiable delay time is 100000000 seconds. Values larger than 100000000 will be silently rounded down to 100000000.

The return value is the amount of time previously remaining in the alarm clock.

Environment

When your program is compiled using the System V environment, `alarm` rounds up any positive fraction of a second to the next second.

When your program is compiled using the POSIX environment, `alarm` takes a parameter of type unsigned, and returns a value of type unsigned.

See Also

getitimer(2), sigpause(2), sigvec(2), signal(3), sleep(3)

Name

assert – program verification

Syntax

#include <assert.h>

assert(*expression*)

Description

The `assert` macro indicates *expression* is expected to be true at this point in the program. It causes an `abort`(3) with a diagnostic comment on the standard error when *expression* is false (0). Compiling with the `cc`(1) option –DNDEBUG effectively deletes `assert` from the program.

Diagnostics

'Assertion failed: a, file *f* *n*'. The *a* is the assertion that failed; *f* is the source file and *n* the source line number of the `assert` statement.

Name

atof, atoi, atol, strtol, strtoul, strtod – convert ASCII to numbers

Syntax

#include <math.h>

double atof(*nptr***)**
char ****nptr*;**

atoi(*nptr***)**
char ****nptr*;**

long atol(*nptr***)**
char ****nptr*;**

long strtol(*nptr***,** *eptr***,** *base***)**
char ****nptr*, *****eptr***;**
int *base***;**

unsigned long strtoul(*nptr***,** *eptr***,** *base***)**
char ****nptr*, *****eptr***;**
int *base***;**

double strtod (*nptr***,** *eptr***)**
char ****nptr*, *****eptr***;**

unsigned long strtoul(*nptr, eptr, base***)**
char ****nptr, **eptr*;**
int *base***;**

Description

These functions convert a string pointed to by *nptr* to floating, integer, and long integer representation respectively. The first unrecognized character ends the string.

The `atof` function recognizes (in order), an optional string of spaces, an optional sign, a string of digits optionally containing a radix character, an optional 'e' or 'E', and then an optionally signed integer.

The `atoi` and `atol` functions recognize (in order), an optional string of spaces, an optional sign, then a string of digits.

The `strtol` function returns as a long integer, the value represented by the character string *nstr*. The string is scanned up to the first character inconsistent with the *base*. Leading white-space characters are ignored.

If the value of *eptr* is not (char **) NULL, a pointer to the character terminating the scan is returned in ***eptr*. If no integer can be formed, ***eptr* is set to *nstr* , and zero is returned.

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

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

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

The `strtoul` function is the same as `strtol` except that `strtoul` returns, as an unsigned long integer, the value represented by the character string *nstr*.

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

The `strtod` function recognizes an optional string of white-space characters, as defined by *isspace* in `ctype`, 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.

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

The radix character for `atof` and `strtod` is that defined by the last successful call to `setlocale` category `LC_NUMERIC`. If `setlocale` category `LC_NUMERIC` has not been called successfully, or if the radix character is not defined for a supported language, the radix character is defined as a period (.).

International Environment

LC_CTYPE If this environment variable is set and valid, `strtod` uses the international language database named in the definition to determine character classification rules.

LC_NUMERIC If this environment is set and valid, `atof` and `strtod` use the international language database named in the definition to determine radix character rules.

LANG If this environment variable is set and valid `atof` and `strtod` use the international language database named in the definition to determine collation and character classification rules. If `LC_CTYPE` or `LC_NUMERIC` is defined, their definition supercedes the definition of LANG.

Diagnostics

The `atof` function returns HUGE if an overflow occurs, and a 0 value if an underflow occurs, and sets *errno* to ERANGE. HUGE is defined in `<math.h>`.

The `atoi` function returns INT_MAX or INT_MIN (according to the sign of the value) and sets *errno* to ERANGE, if the correct value is outside the range of values that can be represented.

The `atol` function returns LONG_MAX or LONG_MIN (according to the sign of the value) and sets *errno* to ERANGE, if the correct value is outside the range of values that can be represented.

The `strtol` function returns LONG_MAX or LONG_MIN (according to the sign of the value) and sets *errno* to ERANGE, if the correct value is outside the range of values that can be represented.

The `strtoul` function returns ULONG_MAX and sets *errno* to ERANGE, if the correct value is outside the range of values that can be represented.

The `strtod` function returns HUGE (according to the sign of the value), and sets *errno* to ERANGE if the correct value would cause overflow. A 0 is returned and *errno* is set to ERANGE if the correct value would cause underflow.

See Also

ctype(3), setlocale(3), scanf(3s), environ(5int)

Name

bsearch – binary search a sorted table

Syntax

#include <stdlib.h>

void *bsearch (*key*, *base*, *nel*, **sizeof** (**key*), *compar*)
void **key, *base*;
size_t *nel*;
int (**compar*)( );

Description

The `bsearch` subroutine 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. The *key* points to the datum to be sought in the table. The *base* points to the element at the base of the table. The *nel* is the number of elements in the table. The *compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero according to whether the first argument is to be considered less than, equal to, or greater than the second.

Diagnostics

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

Notes

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

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

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

See Also

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

Name

bcopy, bcmp, bzero, ffs – bit and byte string operations

Syntax

bcopy(b1, b2, length)
char *b1, *b2;
int length;

bcmp(b1, b2, length)
char *b1, *b2;
int length;

bzero(b1, length)
char *b1;
int length;

ffs(i)
int i;

Description

The functions `bcopy`, `bcmp`, and `bzero` operate on variable length strings of bytes. They do not check for null bytes as the routines in `string`(3) do.

The `bcopy` function copies *length* bytes from string *b1* to the string *b2*.

The `bcmp` function compares byte string *b1* against byte string *b2*, returning zero if they are identical, non-zero otherwise. Both strings are assumed to be *length* bytes long.

The `bzero` function places *length* 0 bytes in the string *b1*.

The *ffs* finds the first bit set in the argument passed it and returns the index of that bit. Bits are numbered starting at 1. A return value of 0 indicates the value passed is zero.

Restrictions

The `bcmp` and `bcopy` routines take parameters backwards from `strcmp` and `strcpy`.

Name

clock – report CPU time used

Syntax

#include <time.h>

clock_t clock ( )

CLOCKS_PER_SEC

Description

The `clock` routine returns the amount of CPU time (in microseconds) used since the first call to `clock`. The time reported is the sum of the user and system times of the calling process and its terminated child processes for which it has executed `wait`(2) or `system`(3). 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 is 16.667 milliseconds.

Restrictions

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

See Also

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

Name

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

Syntax

#include <ctype.h>

int toupper(*c*)
int c;

int tolower(*c*)
int c;

int _toupper(*c*)
int c;

int _tolower(*c*)
int c;

int toascii(*c*)
int c;

Description

The functions `toupper` and `tolower` have as their domain the range of the `getc` function. If the argument to `toupper` represents a lowercase letter, the output from the fuction is the corresponding uppercase letter. If the argument to `tolower` represents an uppercase letter, the result is the corresponding lowercase letter.

The case of *c* depends on the definition of the character in the language database. Because the case of a character can vary between language databases, the case of *c* depends on what language database is in use. Specifically, the case of arguments depends on what property tables are associated the LC_CTYPE category. Property tables are associated with the LC_CTYPE category by a successful call to the `setlocale` function that includes the LC_CTYPE category. If no successful call to define LC_CTYPE has occurred or if the character case information is unavailable for the language in use, the rules of the ASCII coded character set determine the case of arguments.

If the argument to the `toupper` function does not have the uppercase attribute, `toupper` returns the argument unchanged. Likewise, if the argument to the `tolower` function does not have the lowercase attribute, `tolower` returns it unchanged.

The macros `_toupper` and `_tolower` have the same affect as `toupper` and `tolower`. The difference is that the argument to the macros must be an ASCII character (that is, a character in the domain –1 to 127) and the argument must have the appropriate case. Arguments to `_toupper` must have the uppercase attribute and arguments to `_tolower` must the lowercase attribute. The result of supplying arguments to these macros that are outside the domain or do not have the appropriate case is undefined. These macros operate faster than the `toupper` and `tolower` functions.

The macro `toascii` converts its argument to the ASCII character set. The macro converts its argument by truncating the numerical representation of the argument so that it is between –1 and 127. You can use this macro when you move an application to a system other than an ULTRIX system.

International Environment

LC_CTYPE If this environment variable is set and valid, `conv` uses the international language database named in the definition to determine character classification rules.

See Also

ctype(3int), setlocale(3), getc(3)

Name

crypt, crypt16, setkey, encrypt – DES encryption

Syntax

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

char *crypt16(*key, salt***)**
char ****key*, ****salt***;**

setkey(*key***)**
char ****key*;**

Description

The `crypt` subroutine is the password encryption routine. It is based on the NBS Data Encryption Standard, with variations intended to frustrate use of hardware implementations of the DES for key search.

The first argument to `crypt` is normally a user's typed password. The second is a 2-character string chosen from the set [a-zA-Z0-9./]. The *salt* string is used to perturb the DES algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password, in the same alphabet as the salt. The first two characters are the salt itself.

The `crypt16` subroutine is identical to the `crypt` function except that it will accept a password up to sixteen characters in length. It generates a longer encrypted password for use with enhanced security features.

The other entries provide primitive access to the actual DES algorithm. The argument of `setkey` is a character array of length 64 containing only the characters with numerical value 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored, leading to a 56-bit key which is set into the machine.

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

Restrictions

The return values from `crypt` and `crypt16` point to static data areas whose content is overwritten by each call.

Environment

Default Environment

In the default environment on systems that do not have the optional encryption software installed the `encrypt` function expects exactly one argument, the data to be encrypted. The *edflag* argument is not supplied and there is no way to decrypt data. If the optional encryption software is installed the `encrypt` function behaves as it does in the POSIX environment. The syntax for the default environment follows:

encrypt(*block*)
char **block*;

POSIX Environment

In the POSIX environment the encrypt function always expects two arguments. The `encrypt` function will set *errno* to ENOSYS and return if *edflag* is non-zero and the optional encryption software is not present. The syntax for the POSIX environment follows:

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

In all cases the `setkey` function will set *errno* to ENOSYS and return if the optional encryption software is not present.

See Also

login(1), passwd(1), yppasswd(1yp), getpass(3), auth(5), passwd(5), passwd(5yp)
ULTRIX Security Guide for Users and Programmers

Name

ctime, localtime, gmtime, asctime, difftime, mktime, timezone, tzset – date and time functions

Syntax

As shown, the `ctime`, `localtime`, `gmtime`, `asctime`, `difftime`, `mktime`, and `tzset` calls are common to both the non-System V environment and the System V environment.

Common to Both Environments

#include <time.h>

void tzset()

char *ctime(*clock*)
time_t **clock*;

char *asctime(*tm*)
struct *tm* **tm*;

struct *tm* *localtime(*clock*)
time_t **clock*;

struct *tm* *gmtime(*clock*)
time_t **clock*;

double difftime(*time1*, *time0*)
time_t *time1*, *time0*;

time_t mktime(*timeptr*)
struct *tm* **timeptr*;

extern char *tzname[2];

BSD Environment Only

char *timezone(*zone*, *dst*)

System V and POSIX Environments Only

extern long timezone;

extern int daylight;

Description

The `tzset` call uses the value of the environment variable TZ to set up the time conversion information used by `localtime`.

If TZ does not appear in the environment, the file `/etc/zoneinfo/localtime` is used by `localtime`. If this file fails for any reason, the Greenwich Mean Time (GMT) offset as provided by the kernel is used. In this case, Daylight Savings Time (DST) is ignored, resulting in the time being incorrect by some amount if DST is currently in effect. If this fails for any reason, GMT is used.

If TZ appears in the environment but its value is a null string, GMT is used; if TZ appears and its value is not a null string, its value is interpreted using rules specific to the System V and non-System V environments.

Programs that always wish to use local wall clock time should explicitly remove the environmental variable TZ with `unsetenv(3)`.

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

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

The `localtime` and `gmtime` calls return pointers to *tm* structures, described below. The `localtime` call corrects for the time zone and possible DST; `gmtime` converts directly to GMT, which is the time the ULTRIX system uses.

The `asctime` call converts a *tm* structure to a 26-character string, as shown in the previous example, and returns a pointer to the string.

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

```
struct tm {
        int tm_sec;      /* seconds (0 - 59) */
        int tm_min;      /* minutes (0 - 59) */
        int tm_hour;     /* hours (0 - 23) */
        int tm_mday;     /* day of month (1 - 31) */
        int tm_mon;      /* month of year (0 - 11) */
        int tm_year;     /* year - 1900 */
        int tm_wday;     /* day of week (Sunday = 0) */
        int tm_yday;     /* day of year (0 - 365) */
        int tm_isdst;    /* flag: daylight savings time in
                            effect */
        long tm_gmtoff; /* offset from GMT in seconds */
        char *tm_zone;   /* abbreviation of timezone name */

};
```

tm_isdst is nonzero if DST is in effect.

tm_gmtoff is the offset (in seconds) of the time represented from GMT, with positive values indicating East of Greenwich.

The `difftime` call computes the difference between two calendar times: *time1* - *time0* and returns the difference expressed in seconds.

The `mktime` call converts the broken-down local time in the *tm* structure pointed to by *timeptr* into a calendar time value with the same encoding as that of the values returned by `time`. The values of **tm_wday** and **tm_yday** in the structure are ignored, and the other values are not restricted to the ranges indicated above for the *tm* structure. A positive or zero value for **tm_isdst** causes `mktime` to presume that DST, respectively, is or is not in effect for the specified time. A negative value causes `mktime` to attempt to determine whether DST is in effect for the specified time. On successful completion, the values of **tm_wday** and **tm_yday** are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to the ranges indicated above. If the calendar time cannot be represented, the function returns the value **(time_t)–1.**

The external variable *tzname*, contains the current time zone names. The function `tzset` sets this variable.

BSD and POSIX Environment Only

If TZ appears in the environment and its value is not a null string, its value has one of three formats:

:

or

:pathname

or

stdoffset[dst[offset][,start[/time],end[/time]]]

If TZ is the single colon format (first format), GMT is used.

If TZ is the colon followed by a pathname format (second), the characters following the colon specify a pathname of a `tzfile`(5) format file from which to read the time conversion information. If the pathname begins with a slash, it represents an absolute pathname; otherwise the pathname is relative to the system time conversion information directory `/etc/zoneinfo`. If this file fails for any reason, the GMT offset as provided by the kernel is used.

If the first character in TZ is not a colon (third format), the components of the string have the following meaning:

std and *dst*
: Three or more characters that are the designation for the standard (*std*) or summer (*dst*) time zone. Only *std* is required; if *dst* is missing, then summer time does not apply in this locale. Upper- and lowercase letters are explicitly allowed. Any characters except a leading colon (:), digits, comma (,), minus (–), plus (+), and ASCII NUL are allowed.

offset
: Indicates the value to be added to the local time to arrive at Coordinated Universal Time. The *offset* has the form:

  hh[:mm[:ss]]

  The minutes (*mm*) and seconds (*ss*) are optional. The hour (*hh*) is required and may be a single digit. The *offset* following *std* is required. If no *offset* follows *dst*, summer time is assumed to be one hour ahead of standard time. One or more digits may be used; the value is always interpreted as a decimal number. The hour must be between zero and 24, and the minutes (and seconds) – if present – between zero and 59. If preceded by a "–", the time zone is east of the Prime Meridian; otherwise it is west (which may be indicated by an optional preceding "+").

start and *end*
: Indicates when to change to and back from summer time. *Start* describes the date when the change from standard to summer time occurs and *end* describes the date when the change back happens. The format of *start* and *end* must be one of the following:

  J*n* The Julian day *n* ($1 \leq n \leq 365$). Leap days are not counted. That is, in all years, including leap years, February 28 is day 59 and March 1 is day 60. It is impossible to explicitly refer to the occasional February 29.

n The zero-based Julian day ($0 \leq n \leq 365$). Leap days are counted, and it is possible to refer to February 29.

M*m.n.d* The *n*th *d* day of month *m* ($1 \leq n \leq 5$, $0 \leq d \leq 6$, $1 \leq m \leq 12$). When *n* is 5 it refers to the last *d* day of month *m*. Day 0 is Sunday.

time The *time* field describes the time when, in current time, the change to or from summer time occurs. *Time* has the same format as *offset* except that no leading sign (a minus sign (–) or a plus sign (+)) is allowed. The default, if *time* is not given, is 02:00:00.

As an example of the previous format, if the TZ environment variable had the value EST5EDT4,M4.1.0,M10.5.0 it would describe the rule, which went into effect in 1987, for the Eastern time zone in the USA. Specifically, EST would be the designation for standard time, which is 5 hours behind GMT. EDT would be the designation for DST, which is 4 hours behind GMT. DST starts on the first Sunday in April and ends on the last Sunday in October. In both cases, since the time was not specified, the change to and from DST would occur at the default time of 2:00 AM.

The `timezone` call remains for compatibility reasons only; it is impossible to reliably map timezone's arguments (*zone*, a 'minutes west of GMT' value and *dst*, a 'daylight saving time in effect' flag) to a time zone abbreviation.

If the environmental string TZNAME exists, `timezone` returns its value, unless it consists of two comma separated strings, in which case the second string is returned if *dst* is non-zero, else the first string. If TZNAME does not exist, *zone* is checked for equality with a built-in table of values, in which case `timezone` returns the time zone or daylight time zone abbreviation associated with that value. If the requested *zone* does not appear in the table, the difference from GMT is returned; that is, in Afghanistan, `timezone(-(60*4+30),0)` is appropriate because it is 4:30 ahead of GMT, and the string 'GMT+4:30' is returned. Programs that in the past used the `timezone` function should return the *zone* name as set by `localtime` to assure correctness.

System V Environment Only

If TZ appears in the environment its value specifies a pathname of a `tzfile`(5) format file from which to read the time conversion information. If the pathname begins with a slash, it represents an absolute pathname; otherwise the pathname is relative to the system time conversion information directory `/etc/zoneinfo`.

If TZ appears in the environment and using the value as a pathname of a `tzfile`(5) format file fails for any reason, the value is assumed to be a three-letter time zone name followed by a number representing the difference between local time and GMT in hours, followed by an optional three-letter name for a time zone on DST. For example, the setting for New Jersey would be EST5EDT.

System V and POSIX Environment Only

The external *long* variable `timezone` contains the difference, in seconds, between GMT and local standard time (in EST, `timezone` is 5*60*60), The external variable *daylight* is nonzero if and only if the standard USA DST conversion should be applied. These variables are set whenever `tzset`, `ctime`, `localtime`, `mktime`, or `strftime` are called.

Restrictions

The return values point to static data whose content is overwritten by each call. The **tm_zone** field of a returned **struct tm** points to a static array of characters, which will also be overwritten at the next call (and by calls to `tzset`).

Files

/etc/zoneinfo	time zone information directory
/etc/zoneinfo/localtime	local time zone file

See Also

gettimeofday(2), getenv(3), strftime(3), time(3), tzfile(5), environ(7)

Name

isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii – character classification macros

Syntax

#include <ctype.h>

int isalpha (*c*)

int c;

Description

These macros classify character-coded integer values according to the rules of the coded character set (codeset) identified by the last successful call to `setlocale` category `LC_CTYPE`. All macros return non-zero for true and zero for false.

If `setlocale` category `LC_CTYPE` has not been called successfully, or if character classification information is not available for a supported language, then characters are classified according to the rules of the ASCII 7-bit coded character set, returning 0 for values above octal 0177.

The macro `isascii` provides a result for all integer values. The rest provide a result for EOF and values in the character range of the codeset identified by the last successful call to `setlocale` category `LC_CTYPE`.

`isalpha`	*c* is a letter
`isupper`	*c* is an uppercase letter
`islower`	*c* is a lowercase letter
`isdigit`	*c* is a digit
`isxdigit`	*c* is a hexadecimal digit, by default [0-9], [A-F], or [a-f]
`isalnum`	*c* is an alphanumeric character
`isspace`	*c* is a space, tab, carriage return, new line, or form feed
`ispunct`	*c* is a punctuation character (neither control, alphanumeric, nor space)
`isprint`	*c* is a printing character, by default code 040(8) (space) through 0176 (tilde)
`isgraph`	*c* is a printing character, like `isprint` except false for space
`iscntrl`	*c* is a delete character (0177) or ordinary control character (less than 040) except for space characters
`isascii`	*c* is an ASCII character, code less than 0200

International Environment

LC_CTYPE If this environment variable is set and valid, `ctype` uses the international language database named in the definition to determine character classification rules.

LANG If this environment variable is set and valid, `ctype` uses the international language database named in the definition to determine the character classification rules. If `LC_CTYPE` is defined, that definition supercedes the definition of `LANG`.

See Also

conv(3), setlocale(3), stdio(3s), environ(5int), ascii(7)
Guide to Developing International Software

Name

opendir, readdir, telldir, seekdir, rewinddir, closedir – directory operations

Syntax

#include <sys/types.h>
#include <sys/dir.h>

DIR *opendir(*dirname***)**
char ****dirname*;**

struct direct *readdir(*dirp***)**
DIR ****dirp*;**

long telldir(*dirp***)**
DIR ****dirp*;**

seekdir(*dirp, loc***)**
DIR ****dirp*;**
long loc;

rewinddir(*dirp***)**
DIR ****dirp*;**

int closedir(*dirp***)**
DIR ****dirp*;**

Description

The `opendir` library routine opens the directory named by *filename* and associates a directory stream with it. A pointer is returned to identify the directory stream in subsequent operations. The pointer NULL is returned if the specified *filename* can not be accessed, or if insufficient memory is available to open the directory file.

The `readdir` routine returns a pointer to the next directory entry. It returns NULL upon reaching the end of the directory or on detecting an invalid `seekdir` operation. The `readdir` routine uses the `getdirentries` system call to read directories. Since the `readdir` routine returns NULL upon reaching the end of the directory or on detecting an error, an application which wishes to detect the difference must set errno to 0 prior to calling `readdir`.

The `telldir` routine returns the current location associated with the named directory stream. Values returned by `telldir` are good only for the lifetime of the DIR pointer from which they are derived. If the directory is closed and then reopened, the `telldir` value may be invalidated due to undetected directory compaction.

The `seekdir` routine sets the position of the next `readdir` operation on the directory stream. Only values returned by `telldir` should be used with `seekdir`.

The `rewinddir` routine resets the position of the named directory stream to the beginning of the directory.

The `closedir` routine closes the named directory stream and returns a value of 0 if successful. Otherwise, a value of –1 is returned and errno is set to indicate the error. All resources associated with this directory stream are released.

Examples

The following sample code searches a directory for the entry *name*.

```
len = strlen(name);

dirp = opendir(".");

for (dp = readdir(dirp); dp != NULL; dp = readdir(dirp))

if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {

             closedir(dirp);

             return FOUND;

       }

closedir(dirp);

return NOT_FOUND;
```

Environment

In the POSIX environment, the file descriptor returned in the DIR structure after an `opendir()` call will have the FD_CLOEXEC flag set. See **<fcntl.h>** for more detail.

Return Value

Upon successful completion, `opendir()` returns a pointer to an object of type DIR. Otherwise, a value of NULL is returned and errno is set to indicate the error.

The `readdir()` routine returns a pointer to an object of type struct dirent upon successful completion. Otherwise, a value of NULL is returned and errno is set to indicate the error. When the end of the directory is encountered, a value of NULL is returned and errno is not changed.

The `telldir()` routine returns the current location. No errors are defined for `telldir()`, `seekdir()`, and `rewinddir()`.

The `closedir()` routine returns zero upon successful completion. Otherwise, a value of -1 is returned and errno is set to indicate the error.

Diagnostics

The `closedir()` routine will fail if:

[EBADF] The *dirp* argument does not refer to an open directory stream.

[EINTR] The routine was interrupted by a signal.

The `opendir()` routine will fail if:

[EACCES] Search permission is denied for any component of *dirname* or read permission is denied for *dirname*.

[ENAMETOOLONG]
The length of the *dirname* string exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX}.

[ENOENT]	The *dirname* argument points to the name of a file which does not exist, or to an empty string and the environment defined is POSIX or SYSTEM_FIVE.
[ENOTDIR]	A component of *dirname* is not a directory.
[EMFILE]	Too many file descriptors are currently open for the process.
[ENFILE]	Too many files are currently open in the system.

The `readdir()` routine will fail if:

[EBADF]	The *dirp* argument does not refer to an open directory stream.

See Also

close(2), getdirentries(2), lseek(2), open(2), read(2), dir(5)

Name

div, ldiv – integer division

Syntax

#include <stdlib.h>

div_t div(*numer, denom***)**
int *numer***;**
int *denom***;**

ldiv_t ldiv(*numer, denom***)**
long *numer***;**
long *denom***;**

Description

The `div` and `ldiv` functions return the quotient and remainder of the division of the numerator *numer* by the denominator *denom.*

The return types div_t and ldiv_t are defined, in stdlib.h, as follows:

```
typedef struct {
                int     quot;  /* quotient */
                int     rem;   /* remainder */
        }       div_t;         /* result of div() */

typedef struct {
                long    quot;  /* quotient */
                long    rem;   /* remainder */
        }       ldiv_t;              /* result of ldiv() */
```

Restrictions

If division by zero is attempted, the behavior of `div` and `ldiv` is undefined.

Name

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

Syntax

double drand48 ( )

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

long lrand48 ( )

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

long mrand48 ( )

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

void srand48 (seedval)
long seedval;

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

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

Description

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Notes

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

long irand48 (m)
unsigned short m;

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

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

See Also

rand(3)
ULTRIX Programmer's Manual, Unsupported

Name

ecvt, fcvt, gcvt – output conversion

Syntax

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

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

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

Description

The `ecvt` routine converts the *value* to a null-terminated string of *ndigit* ASCII digits and returns a pointer thereto. 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). If the sign of the result is negative, the word pointed to by *sign* is non-zero, otherwise it is zero. The low-order digit is rounded.

The `fcvt` routine is identical to `ecvt`, except that the correct digit has been rounded for FORTRAN F-format output of the number of digits specified by *ndigits*.

The `gcvt` routine converts the *value* to a null-terminated ASCII string in *buf* and returns a pointer to *buf*. It attempts to produce *ndigit* significant digits in FORTRAN F format if possible, otherwise E format is used, ready for printing. Trailing zeros may be suppressed.

The symbol used to represent a radix character is obtained from the last successful call to `setlocale` category `LC_NUMERIC`. The symbol can be determined by calling:

```
nl_langinfo (RADIXCHAR);
```

If `setlocale` category `LC_NUMERIC` has not been called successfully, or if the radix character is not defined for a supported language, the radix character defaults to a period (.).

International Environment

LC_NUMERIC If this environment is set and valid, `ecvt` uses the international language database named in the definition to determine radix character rules.

LANG If this environment is set and valid, `ecvt` uses the international language database named in the definition to determine radix character rules. If `LC_NUMERIC` is defined, its definition supercedes the definition of `LANG`.

Restrictions

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

See Also

setlocale(3), nl_langinfo(3int), printf(3int), printf(3s)
Guide to Developing International Software

SC

emulate_branch (3)

Name

emulate_branch, execute_branch – branch emulation

Syntax

#include <signal.h>

emulate_branch(*scp, branch_instruction***)**
struct sigcontext ****scp*;**
unsigned long *branch_instruction***;**

execute_branch(*branch_instruction***)**
unsigned long *branch_instruction***;**

Description

The `emulate_branch` function is passed a signal context structure and a branch instruction. It emulates the branch based on the register values in the signal context structure. It modifies the value of the program counter in the signal context structure (*sc_pc*) to the target of the *branch_instruction*. The program counter must initially be pointing at the branch and the register values must be those at the time of the branch. If the branch is not taken the program counter is advanced to point to the instruction after the delay slot (*sc_pc* += 8).

If the branch instruction is a 'branch on coprocessor 2' or 'branch on coprocessor 3' instruction, `emulate_branch` calls `execute_branch` to execute the branch in data space to determine if it is taken or not.

Return Value

The `emulate_branch` function returns a 0 if the branch was emulated successfully. A non-zero value indicates the value passed as a branch instruction was not a branch instruction.

The `execute_branch` function returns non-zero on taken branches and zero on non-taken branches.

Restrictions

Since `execute_branch` is only intended to be used by `emulate_branch` it does not check its parameter to see if in fact it is a branch instruction. It is really a stop gap in case a coprocessor is added without the kernel fully supporting it (which is unlikely).

See Also

cacheflush(2), sigvec(2), signal(3)

Name

end, etext, edata – last locations in program

Syntax

extern end;
extern etext;
extern edata;
extern eprol;

Description

These names refer neither to routines nor to locations with interesting contents. The address of `etext` is the first address above the program text, `edata` above the initialized data region, and `eprol` is the first instruction of the user's program that follows the runtime startup routine.

When execution begins, the program break coincides with `end,` but it is reset by the routines `brk`(2), `malloc`(3), standard input/output `stdio`(3s), the profile (**–p**) option of `cc`(1), and so forth. The current value of the program break is reliably returned by sbrk(0). For further information, see `brk`(2).

See Also

cc(1), brk(2), malloc(3), stdio(3s)

Name

end, etext, edata – last locations in program

Syntax

extern end;
extern etext;
extern edata;

Description

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

When execution begins, the program break coincides with `end,` but it is reset by the routines `brk`(2), `malloc`(3), standard input/output `stdio`(3s), the profile (**–p**) option of `cc`(1), and so forth. The current value of the program break is reliably returned by 'sbrk(0)'. For further information, see `brk`(2).

See Also

brk(2), malloc(3), stdio(3s)

Name

execl, execv, execle, execlp, execvp, exect, environ – execute a file

Syntax

execl(name, arg0, arg1, ..., argn, (char *)0)
char *name, *arg0, *arg1, ..., *argn;

execv(name, argv)
char *name, *argv[];

execle(name, arg0, arg1, ..., argn, (char *)0, envp)
char *name, *arg0, *arg1, ..., *argn, *envp[];

execlp(file, arg0, arg1, ..., argn, (char *)0)
char *file, *arg0, *arg1, ..., *argn;

execvp(file,argv)
char *file, *argv[];

exect(name, argv, envp)
char *name, *argv[], *envp[];

extern char **environ;

Description

These routines provide various interfaces to the `execve` system call. Refer to `execve`(2) for a description of their properties; only brief descriptions are provided here.

In all their forms, these calls overlay the calling process with the named file, then transfer to the entry point of the core image of the file. There can be no return from a successful exec. The calling core image is lost.

The *name* argument is a pointer to the name of the file to be executed. The pointers *arg*[*0*], *arg*[*1*] ... address null-terminated strings. Conventionally *arg*[*0*] is the name of the file.

Two interfaces are available. `execl` is useful when a known file with known arguments is being called; the arguments to `execl` are the character strings constituting the file and the arguments; the first argument is conventionally the same as the file name (or its last component). A 0 argument must end the argument list.

The `execv` version is useful when the number of arguments is unknown in advance. The arguments to `execv` are the name of the file to be executed and a vector of strings containing the arguments. The last argument string must be followed by a 0 pointer.

The `exect` version is used when the executed file is to be manipulated with `ptrace`(2). The program is forced to single step a single instruction giving the parent an opportunity to manipulate its state.

VAX-11

On VAX-11 machines, this is done by setting the trace bit in the process status longword.

When a C program is executed, it is called as follows:

```
main(argc, argv, envp)
int argc;
char **argv, **envp;
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. As indicated, *argc* is conventionally at least one and the first member of the array points to a string containing the name of the file.

The *argv* is directly usable in another `execv` because *argv*[*argc*] is 0.

The *envp* is a pointer to an array of strings that constitute the *environment* of the process. Each string consists of a name, an ``='', and a null-terminated value. The array of pointers is terminated by a null pointer. The shell `sh`(1) passes an environment entry for each global shell variable defined when the program is called. See `environ`(7) for some conventionally used names. The C run-time start-off routine places a copy of *envp* in the global cell `environ`, which is used by `execv` and `execl` to pass the environment to any subprograms executed by the current program.

The `execlp` and `execvp` routines are called with the same arguments as `execl` and `execv`, but duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the environment.

Restrictions

If `execvp` is called to execute a file that turns out to be a shell command file, and if it is impossible to execute the shell, the values of *argv[0]* and *argv[–1]* will be modified before return.

Diagnostics

If the file cannot be found, if it is not executable, if it does not start with a valid magic number if maximum memory is exceeded, or if the arguments require too much space, a return constitutes the diagnostic; the return value is –1. For further information, see `a.out`(5). Even for the super-user, at least one of the execute-permission bits must be set for a file to be executed.

Files

/bin/sh Shell, invoked if command file found by execlp or execvp

See Also

csh(1), execve(2), fork(2), environ(7)

Name

execl, execv, execle, execlp, execvp, exect, environ – execute a file

Syntax

execl(*name, arg0, arg1, ..., argn, (char *)0***)**
char ****name, *arg0, *arg1, ..., *argn*;**

execv(*name, argv***)**
char ****name, *argv[]*;**

execle(*name, arg0, arg1, ..., argn, (char *)0, envp***)**
char ****name, *arg0, *arg1, ..., *argn, *envp[]*;**

execlp(*file, arg0, arg1, ..., argn, (char *)0***)**
char ****file, *arg0, *arg1, ..., *argn*;**

execvp(*file,argv***)**
char ****file, *argv[]*;**

exect(*name, argv, envp***)**
char ****name, *argv[], *envp[]*;**

extern *char **environ***;**

Description

These routines provide various interfaces to the `execve` system call. Refer to `execve`(2) for a description of their properties; only brief descriptions are provided here.

In all their forms, these calls overlay the calling process with the named file, then transfer to the entry point of the core image of the file. There can be no return from a successful exec. The calling core image is lost.

The *name* argument is a pointer to the name of the file to be executed. The pointers *arg*[*0*], *arg*[*1*] ... address null-terminated strings. Conventionally *arg*[*0*] is the name of the file.

Two interfaces are available. `execl` is useful when a known file with known arguments is being called; the arguments to `execl` are the character strings constituting the file and the arguments; the first argument is conventionally the same as the file name (or its last component). A 0 argument must end the argument list.

The `execv` version is useful when the number of arguments is unknown in advance. The arguments to `execv` are the name of the file to be executed and a vector of strings containing the arguments. The last argument string must be followed by a 0 pointer.

The `exect` version is used when the executed file is to be manipulated with `ptrace`(2). The program is forced to single step a single instruction giving the parent an opportunity to manipulate its state. On the VAX-11 this is done by setting the trace bit in the process status longword.

When a C program is executed, it is called as follows:

```
main(argc, argv, envp)
int argc;
char **argv, **envp;
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. As indicated, *argc* is conventionally at least one and the first member of the array points to a string containing the name of the file.

The *argv* is directly usable in another `execv` because *argv*[*argc*] is 0.

The *envp* is a pointer to an array of strings that constitute the *environment* of the process. Each string consists of a name, an "=", and a null-terminated value. The array of pointers is terminated by a null pointer. The shell `sh`(1) passes an environment entry for each global shell variable defined when the program is called. See `environ`(7) for some conventionally used names. The C run-time start-off routine places a copy of *envp* in the global cell `environ`, which is used by `execv` and `execl` to pass the environment to any subprograms executed by the current program.

The `execlp` and `execvp` routines are called with the same arguments as `execl` and `execv`, but duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the environment.

Restrictions

If `execvp` is called to execute a file that turns out to be a shell command file, and if it is impossible to execute the shell, the values of *argv[0]* and *argv[-1]* will be modified before return.

Diagnostics

If the file cannot be found, if it is not executable, if it does not start with a valid magic number, if maximum memory is exceeded, or if the arguments require too much space, a return constitutes the diagnostic; the return value is -1. For further information, see `a.out`(5). Even for the super-user, at least one of the execute-permission bits must be set for a file to be executed.

Files

/bin/sh	Shell, invoked if command file found by execlp or execvp

See Also

csh(1), execve(2), fork(2), environ(7)

Name

exit – terminate a process after flushing any pending output

Syntax

exit(*status*)
int *status*;
int atexit(*func*)
void (**func*)();

Description

The `exit` function terminates a process after calling the Standard I/O library function, *_cleanup,* to flush any buffered output. The `exit` function never returns.

The `atexit` function registers a function to be called (without arguments) at normal program termination; functions are called in the reverse order of their registration (that is, most recent first). If a function is registered more than once, it will be called more than once.

Return Value

The `atexit` function returns zero if the registration succeeds, or -1 if the function pointer is null or if too many functions are registered.

See Also

exit(2), intro(3s)

Name

fpc, get_fpc_csr, set_fpc_csr, swapRM, swapINX – floating-point control registers

Syntax

#include <mips/fpu.h>

int get_fpc_csr()

int set_fpc_csr(*csr***)**
int *csr***;**

int get_fpc_irr()

int swapRM(*x***)**
int *x***;**

int swapINX(*x***)**
int *x***;**

Description

These functions are to get and set the floating-point control registers of RISC floating-point units. All of these functions take and return their values as 32 bit integers.

The file **<mips/fpu.h>** contains unions for each of the control registers. Each union contains a structure that breaks out the bit fields into the logical parts for each control register. This file also contains constants for fields of the control registers.

RISC floating-point implementations have a control and status register and an implementation revision register. The control and status register is returned by `get_fpc_csr`. The routine `set_fpc_csr` sets the control and status register and returns the old value. The implementation revision register is read-only and is returned by the routine `get_fpc_irr`.

The function `swapRM` sets only the rounding mode and returns the old rounding mode. The function `swapINX` sets only the sticky inexact bit and returns the old one. The bits in the arguments and return values to `swapRM` and `swapINX` are right justified.

Name

fp_class – classes of IEEE floating-point values

Syntax

#include <fp_class.h>

int fp_class_d(double *x***);**

int fp_class_f(float *x***);**

Description

These routines are used to determine the class of IEEE floating-point values. They return one of the constants in the file `<fp_class.h>` and never cause an exception, even for signaling NaNs. These routines are to implement the recommended function class(x) in the appendix of the IEEE 754-1985 standard for binary floating-point arithmetic. The constants in `<fp_class.h>` refer to the following classes of values:

Constant	Class
FP_SNAN	Signaling NaN (Not-a-Number)
FP_QNAN	Quiet NaN (Not-a-Number)
FP_POS_INF	$+\infty$ (positive infinity)
FP_NEG_INF	$-\infty$ (negative infinity)
FP_POS_NORM	positive normalized nonzero
FP_NEG_NORM	negative normalized nonzero
FP_POS_DENORM	positive denormalized
FP_NEG_DENORM	negative denormalized
FP_POS_ZERO	+0.0 (positive zero)
FP_NEG_ZERO	-0.0 (negative zero)

Also See

ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic

Name

frexp, ldexp, modf – split into mantissa and exponent

Syntax

#include <math.h>

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

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

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

Description

The `frexp` subroutine returns the mantissa of a double *value* as a double quantity, *x,* of magnitude less than 1.0 and greater than or equal to 0.5 (0.5 <= |x| < 1) and stores an integer *n* such that *value* = *x**2***n* indirectly through *eptr*.

The `ldexp` returns the quantity *value**2***exp*.

The `modf` returns the positive fractional part of *value* and stores the integer part indirectly through *iptr*.

Return Value

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

Name

ftoi, itof, dtoi, itod, gtoi, itog – convert floating values between VAX and IEEE format

Syntax

int ftoi(*value***)**
 float ****value*;**

int itof(*value***)**
 float ****value*;**

int dtoi(*value***)**
 double ****value*;**

int itod(*value***)**
 double ****value*;**

int gtoi(*value***)**
 double ****value*;**

int itog(*value***)**
 double ****value*;**

Description

The following C library functions convert floating values between VAX and IEEE formats.

The `ftoi` function converts the specified VAX ffloat number to IEEE single-precision format. It returns zero if successful and nonzero without performing the conversion if not successful (for example, underflow).

The `itof` function converts the specified IEEE single-precision number to VAX ffloat format. It returns zero if successful and nonzero without performing the conversion if not successful (for example, overflow).

The `dtoi` function converts the specified VAX dfloat number to IEEE double-precision format. It returns zero if successful and nonzero without performing the conversion if not successful (for example, underflow).

The `itod` function converts the specified IEEE double-precision number to VAX dfloat format. It returns zero if successful and nonzero without performing the conversion if not successful (for example, underflow or overflow).

The `gtoi` function converts the specified VAX gfloat number to IEEE double-precision format. It returns zero if successful and nonzero without performing the conversion if not successful (for example, underflow).

The `itog` function converts the specified IEEE double-precision number to VAX gfloat format. It returns zero if successful and nonzero without performing the conversion if not successful (for example, underflow).

Name

ftok – standard interprocess communication package

Syntax

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

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

Description

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

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

Return Value

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

Warning

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

See Also

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

Name

ftw – walk a file tree

Syntax

#include <ftw.h>

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

Description

The `ftw` subroutine 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, and an integer. For further information, see `stat`(2). Possible values of the integer, defined in the <ftw.h> header file, are FTW_F for a file, FTW_D for a directory, FTW_DNR for a directory that cannot be read, and FTW_NS for an object for which **stat** could not successfully be executed. If the integer is FTW_DNR, descendants of that directory will not be processed. If the integer is FTW_NS, the the contents of the **stat** structure will be undefined. An example of an object that would cause FTW_NS to be passed to *fn* would be a file in a directory with read but without execute (search) permission.

The `ftw` subroutine visits a directory before visiting any of its descendants.

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

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

Restrictions

Because `ftw` is recursive, it is possible for it to terminate with a memory fault when applied to very deep file structures.
It could be made to run faster and use less storage on deep structures at the cost of considerable complexity.
The `ftw` subroutine uses `malloc`(3) to allocate dynamic storage during its operation. If `ftw` is forcibly terminated, such as by *longjmp* being executed by *fn* or an interrupt routine, `ftw` will not have a chance to free that storage, so it will remain permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have *fn* return a nonzero value at its next invocation.

Diagnostics

[EACCES] Search permission is denied on a component of *path* or read permission is denied for path.

[ENAMETOOLONG] The length of the path string exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX}.

[ENOENT] The path argument points to the name of a file which does not exist, or to an empty string and the environment defined is POSIX or SYSTEM_FIVE.

[ENOTDIR] A component of *path* is not a directory.

[ENOMEM] Not enough memory was available to complete the file tree walk.

See Also

stat(2), malloc(3)

Name

getauthuid, storeauthent, setauthfile, endauthent – get/set auth entry

Syntax

#include <sys/types.h>
#include <auth.h>

AUTHORIZATION *getauthuid(*uid***)**
uid_t *uid***;**

int storeauthent(*auth***)**
AUTHORIZATION **auth***;**

void setauthfile(*pathname***)**
char **pathname***;**

int endauthent()

Description

The `getauthuid` function looks up the auth entry for the specified user ID and returns a pointer to a static area containing it.

The `storeauthent` function will store the specified auth entry into the local auth database, overwriting any existing entry with the same *a_uid* field.

The `setauthfile` function will set the pathname of the file to be used for the local auth database in all subsequent operations.

The `endauthent` functions closes the auth database. Subsequent calls to getauthuid and storeauthent will reopen it.

The auth database may be distributed via the BIND/Hesiod naming service.

Restrictions

Only the super-user and members of the group *authread* may read information from the auth database.

Only the super-user may modify the auth database.

The auth databse may not be distributed via the Yellow Pages service.

Return Value

Functions which return a pointer value will return the null pointer (0) on EOF or error. Other functions will return zero (0) on success and a negative value on failure.

Files

/etc/auth.[pag,dir]

See Also

getpwent(3), auth(5), edauth(8)
Security Guide for Users and Programmers
Security Guide for Administrators
Guide to the BIND/Hesiod Service

Name

getcwd – get pathname of working directory

Syntax

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

Description

The `getcwd` subroutine returns a pointer to the current directory pathname. The value of *size* must be at least two greater than the length of the pathname to be returned.

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

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

Examples

```
char *cwd, *getcwd();
.
.
.
if ((cwd = getcwd((char *)NULL, 64)) == NULL) {

     perror("pwd");
     exit(1);
}
printf("%s\n", cwd);
```

Return Value

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

Diagnostics

[EINVAL]	The size argument is zero or negative.
[ERANGE]	The size argument is greater than zero, but is smaller than the length of the pathname+1;
[EACCES]	Read or search permission is denied for a component of the pathname.
[ENOMEM]	Insufficient storage space is available.

See Also

pwd(1), malloc(3), popen(3)

Name

getenv, setenv, unsetenv – manipulate environment variables

Syntax

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

setenv(*name***,** *value***,** *overwrite***)**
char ****name*,** *value***;**
int *overwrite***;**

void unsetenv(*name***)**
char ****name*;**

Description

The `getenv` subroutine searches the environment list for a string of the form *name* = *value* and returns a pointer to the string *value* if such a string is present, otherwise `getenv` returns the value 0 (NULL). For further information, see `environ`(7).

The `setenv` subroutine searches the environment list in the same manner as `getenv`. If the string *name* is not found, a string of the form *name=value* is added to the environment. If it is found, and *overwrite* is non-zero, its value is changed to *value*. The `setenv` subroutine returns 0 on success and -1 on failure, where failure is caused by an inability to allocate space for the environment.

The `unsetenv` subroutine removes all occurrences of the string *name* from the environment. There is no library provision for completely removing the current environment. It is suggested that the following code be used to do so.

```
static char     *envinit[1];
extern char     **environ;
environ = envinit;
```

All of these routines permit, but do not require, a trailing equals sign (=) on *name* or a leading equals sign on *value*.

See Also

csh(1), sh(1), execve(2), putenv(3), environ(7)

Name

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

Syntax

#include <grp.h>

struct group *getgrent()

struct group *getgrgid(*gid***)**
gid_t *gid***;**

struct group *getgrnam(*name***)**
char ****name*;**

setgrent()

endgrent()

Description

The `getgrent`, `getgrgid` and `getgrnam` subroutines each return pointers to an object with the following structure containing the broken-out fields of a line in the group database:

```
struct  group { /* see getgrent(3) */
        char    *gr_name;
        char    *gr_passwd;
        int     gr_gid;
        char    **gr_mem;
};

struct group *getgrent(), *getgrgid(), *getgrnam();
```

The members of this structure are:

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

A call to `setgrent` has the effect of rewinding the group file to allow repeated searches. The `endgrent` may be called to close the group database when processing is complete.

The `getgrent` subroutine simply reads the next line while `getgrgid` and `getgrnam` search until a matching *gid* or *name* is found (or until EOF is encountered). The `getgrent` subroutine keeps a pointer in the database, allowing successive calls to be used to search the entire file.

A call to `setgrent` must be made before a `while` loop using `getgrent` in order to perform initialization and an `endgrent` must be used after the loop. Both `getgrgid` and `getgrnam` make calls to `setgrent` and `endgrent`.

Restrictions

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

If YP is running, `getgrent` does not return the entries in any particular order. See the *Guide to the Yellow Pages Service* for setup information.

The group database may also be distributed via the BIND/Hesiod naming service. See the *Guide to the BIND/Hesiod Service* for more information.

Return Value

A null pointer (0) is returned on EOF or error.

Files

`/etc/group`

See Also

group(5), svc.conf(5)
Guide to the BIND/Hesiod Service
Guide to the Yellow Pages Service

gethostsex (3)

Name

gethostsex – get the byte sex of the host machine

Syntax

#include <sex.h>
int gethostsex()

Description

The `gethostsex` routine returns one of two constants, BIGENDIAN or LITTLEENDIAN, for the sex of the host machine. These constants are in **sex.h.**

See Also

swapsex(3)

Name

getlogin – get login name

Syntax

char *getlogin()

Description

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

If `getlogin` is called within a process that is not attached to a typewriter, it returns NULL. The correct procedure for determining the login name is to first call `getlogin` and if it fails, to call `getpw` (`getuid`).

Restrictions

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

Return Value

Returns NULL (0) if name not found.

Files

/etc/utmp

See Also

getgrent(3), getpw(3), getpwent(3), utmp(5)

Name

getmountent – get information about mounted file systems without blocking

Syntax

#include <sys/types.h>
#include <sys/param.h>
#include <sys/mount.h>

```
getmountent(start, buffer, nentries)
int             *start;
struct fs_data  *buffer;
int              nentries;
```

Description

The `getmountent` library routine retrieves mounted file system information from memory without blocking. The file system information retrieved (the number of free inodes and blocks) might not be up to date. If the accuracy of the file system information retrieved is critical, you should use `statfs` or `getmnt` instead of `getmountent`.

The *start* argument is the current logical location within the internal system mount table and must be initially set to 0. The *buffer* argument is the holding area for the returned information; that is, the `fs_data` structures. The size of *buffer* should be at least the number of entries times the size of the `fs_data` structure, in bytes.

The *nentries* argument defines the number of mount table entries that are to be retrieved.

The number of file systems described by the information placed in *buffer* is returned. The *start* argument is updated so that successive calls can be used to retrieve the entire mount table.

Return Value

Upon successful completion, a value indicating the number of `struct fs_data` structures stored in *buffer* is returned. If there are no more file systems in the mount table, `0` is returned. Otherwise, `-1` is returned and the global variable *errno* is set to indicate the error.

Diagnostics

EINVAL	Invalid argument.
EFAULT	Either *buffer* or *start* causes an illegal address to be referenced.
EIO	An I/O error occurred while reading from the file system.

See Also

getmnt(2), statfs(3)

Name

getopt – get option letter from argument vector

Syntax

#include <stdio.h>
int getopt (argc, argv, optstring)
int argc;
char **argv;
char *optstring;

extern char *optarg;
extern int optind, opterr;

Description

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

The function `getopt` places in *optind* the *argv* index of the next argument to be processed. The external variable optind is automatically initialized to 1 before the first call to `getopt`.

When all options have been processed (that is, up to the first non-option argument), `getopt` returns EOF. The special option — may be used to delimit the end of the options; EOF will be returned, and — will be skipped.

Diagnostics

The function `getopt` prints an error message on *stderr* and returns a question mark (?) when it encounters an option letter that is not included in *optstring*. Setting opterr to 0 disables this error message.

Examples

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options **a** and **b**, and the options **f** and **o**, both of which require arguments:

```
#include <stdio.h>
main (argc, argv)
int argc;
char **argv;
{
        int c;
        extern int optind, opterr;
        extern char *optarg;
        .
        .
        .
        .
        while ((c = getopt (argc, argv, "abf:o:")) != EOF)
                switch (c) {
                case 'a':
```

```
                        if (bflg)
                                errflg++;
                        else
                                aflg++;
                        break;
                case 'b':
                        if (aflg)
                                errflg++;
                        else
                                bproc( );
                        break;
                case 'f':
                        ifile = optarg;
                        break;
                case 'o':
                        ofile = optarg;
                        bufsiza = 512;
                        break;
                case '?':
                        errflg++;
                }
        if (errflg) {
                fprintf (stderr, "usage: . . . ");
                exit (2);
        }
        for ( ; optind < argc; optind++) {
               if (access (argv[optind], 4)) {
        .
        .
        .
}
```

See Also

getopt(1)

Name

getpass – read a password

Syntax

char *getpass(prompt)
char *prompt;

Description

The `getpass` subroutine reads a password from the file `/dev/tty`, or if that cannot be opened, from the standard input, after prompting with the null-terminated string *prompt* and disabling echoing. The `getpass` subroutine can return up to `PASS_MAX` characters. `PASS_MAX` is defined in `/usr/include/sys/limits.h`. A pointer is returned to a null-terminated string of at most 16 characters.

Environment

When your program is compiled using the System V environment, if the file `/dev/tty` cannot be opened, a NULL pointer is returned. An interrupt will terminate input and send an interrupt signal to the calling process before returning.

Restrictions

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

Files

`/dev/tty`

See Also

`crypt(3)`

Name

getpw – get name from uid

Syntax

getpw(uid, buf)
char *buf;

Description

The `getpw` routine has been superseded by `getpwuid`, see `getpwent`(3).

The `getpw` routine searches the password file for the (numerical) *uid*, and fills in *buf* with the corresponding line; it returns nonzero if *uid* could not be found. The line is null terminated.

Diagnostics

Nonzero return on error.

Files

/etc/passwd

See Also

getpwent(3), passwd(5yp)

Name

getpwent, getpwuid, getpwnam, setpwent, endpwent, setpwfile – get password entry

Syntax

#include <pwd.h>

struct passwd *getpwent()

struct passwd *getpwuid(*uid*)
uid_t *uid*;

struct passwd *getpwnam(*name*)
char **name*;

void setpwent()

void endpwent()

void setpwfile(*pathname*)
char **pathname*

Description

The routines, `getpwent`, `getpwuid` and `getpwnam`, each return a pointer to an object with the following structure containing the broken-out fields of a line in the password database:

```
struct  passwd { /* see getpwent(3) */
        char    *pw_name;
        char    *pw_passwd;
        uid_t    pw_uid;
        gid_t    pw_gid;
        int      pw_quota;
        char    *pw_comment;
        char    *pw_gecos;
        char    *pw_dir;
        char    *pw_shell;
};

struct passwd *getpwent(), *getpwuid(), *getpwnam();
```

The fields *pw_quota* and *pw_comment* are unused; the others have meanings described in `passwd`(5).

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

The `getpwent` subroutine simply retieves the next entry while `getpwuid` and `getpwnam` search until a matching *uid* or *name* is found (or until all entries are exhausted). The `getpwent` subroutine keeps a pointer in the database, allowing successive calls to be used to search the entire database.

A call to `setpwent` must be made before a `while` loop using `getpwent` in order to perform initialization and an `endpwent` must be used after the loop. Both `getpwuid` and `getpwnam` make calls to `setpwent` and `endpwent`.

The `setpwfile` subroutine sets the pathname of the ASCII passwd file and optional hashed database to be used for local passwd lookups. If a passwd file has been left open by a call to `setpwent` or `getpwent`, `setpwfile` will close it first. `Setpwfile` does not directly affect the use of distributed passwd databases.

Restrictions

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

If YP is running, `getpwent` does not return the entries in any particular order. See the *Guide to the Yellow Pages Service* for setup information.

The password database may also be distributed via the BIND/Hesiod naming service. See the *Guide to the BIND/Hesiod Service* for more information.

Return Value

Null pointer (0) returned on EOF or error.

Files

`/etc/passwd`

See Also

getlogin(3), passwd(5), svc.conf(5)
Guide to the BIND/Hesiod Service
Guide to the Yellow Pages Service

Name

getrpcent, getrpcbynumber, getrpcbyname, setrpcent, endrpcent – get rpc entry

Syntax

#include <netdb.h>

struct rpcent *getrpcent()

struct rpcent *getrpcbynumber(*number*)
int *number*;

struct rpcent *getrpcbyname(*name*)
char **name*;

setrpcent(*stayopen*)
int *stayopen*;

endrpcent( )

Description

The `getrpcent`, `getrpcbynumber` and `getrpcbyname` subroutines each return pointers to an object with the following structure containing the broken-out fields of a line in the rpc database:

```
struct  rpcent {               /* see getrpcent(3) */
        char    *r_name;
        char    **r_aliases;   /* alias list */
        char    r_number;      /* rpc program number */
};
struct group *getrpcent(), *getrpcbynumber(), *getrpcbyname();
```

The members of this structure are:

r_name The name of the rpc.
r_aliases A zero-terminated list of alternate names for the rpc.
r_number The rpc program number for the rpc.

If the *stayopen* flag on the `setrpcent` subroutine is NULL, the `rpc` database is opened. Otherwise the `setrpcent` has the effect of rewinding the `rpc` database. The `endrpcent` may be called to close the `rpc` file when processing is complete.

The `getrpcent` subroutine simply reads the next line while `getrpcbynumber` and `getrpcbyname` search until a matching *gid* or *name* is found (or until EOF is encountered). The `getrpcent` subroutine keeps a pointer in the database, allowing successive calls to be used to search the entire file.

A call to `setrpcent` must be made before a `while` loop using `getrpcent` in order to perform initialization and an `endrpcent` must be used after the loop. Both `getrpcbynumber` and `getrpcbyname` make calls to `setrpcent` and `endrpcent`.

Restrictions

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

If YP is running, `getrpcent` does not return the entries in any particular order. See the *Guide to the Yellow Pages Service* for setup information.

The `rpc` database may also be distributed by the BIND/Hesiod naming service. See the *Guide to the BIND/Hesiod Service* for more information.

Return Value

A null pointer (0) is returned on EOF or error.

Files

`/etc/rpc`

See Also

rpc(5), svc.conf(5)
Guide to the BIND/Hesiod Service
Guide to the Yellow Pages Service

Name

getsvc - get a pointer to the svcinfo structure

Syntax

#include <sys/svcinfo.h>

struct svcinfo *getsvc()

Description

The `getsvc` call retrieves information from the system about the `svcinfo` structure by returning a pointer to the structure. This structure is initialized the first time a `getsvc` call is made. The contents of the `/etc/svc.conf` file are parsed and stored in the `svcinfo` structure. If the `/etc/svc.conf` file is modified, the contents of this structure will be updated upon the next `getsvc` call.

The `/etc/svc.conf` file contains the names of the databases that can be served by YP, BIND, or local files and the name service selection for each database. It also has settings for four security parameters. The database service selection and security parameters are stored in the `svcinfo` structure.

The following structure exists in the `svcinfo.h` file:

```
#define SVC_DATABASES 20
#define SVC_PATHSIZE 8
struct svcinfo {
    int svcdate;        /* Last mod date of /etc/svc.conf */

    int svcpath[SVC_DATABASES][SVC_PATHSIZE];     /* indexed by
                          databases and choice 0=first choice
                         1=second choice, etc value stored is
                      source */

       struct {
           int passlenmin;
           int passlenmax;
           int softexp;
           int seclevel;
       } svcauth;
};
```

The `svcdate` field contains the date that the `/etc/svc.conf` file was last modified. The `svcpath` array contains the name service choices for each database. The `svcauth` structure contains the values for the four security parameters: password length minimum (*passlenmin*), password length maximum (*passlenmax*), soft expiration date of a password (*softexp*), and security mode of a system (*seclevel*).

Examples

The following programming example shows how to use the `getsvc` call to use the information in the `svcinfo` structure to process specific host information.

```
#include <sys/svcinfo.h>
struct svcinfo *svcinfo;

if ((svcinfo = getsvc()) != NULL)
  for (i=0; (j = svcinfo->svcpath[SVC_HOSTS][i]) != SVC_LAST; i++)
        switch(j) {
            case SVC_BIND:
                /* process BIND hosts */
            case SVC_YP:
                /* process YP hosts */
            case SVC_LOCAL:
               /* process LOCAL hosts */
        }
```

Files

/etc/svc.conf
/usr/include/sys/svcinfo.h

See Also

svc.conf(5), svcsetup(8)

Name

getttyent, getttynam, setttyent, endttyent – get ttys file entry

Syntax

#include <ttyent.h>
struct ttyent *getttyent ( )
struct ttyent *getttynam (*name* **) char** **name;*
int setttyent ( )
int endttyent ( )

Description

These functions allow a program to access data in the file /etc/ttys. The `getttyent` function reads the `/etc/ttys` file line by line, opening the file if necessary. `setttyent` rewinds the file, and `endttyent` closes it. `getttynam` searches from the beginning of the file until a matching name is found, or until end-of-file is encountered.

The functions `getttyent` and `getttynam` each return a pointer to an object that has the following structure. Each element of the structure contains one field of a line in the `/etc/ttys` file.

```
struct ttyent {         /* see getttyent(3) */
       char *ty_name;   /* terminal device name */
       char *ty_getty;  /* command to execute, usually getty */
       char *ty_type;   /* terminal type for termcap (3X) */
       int  ty_status;  /* status flags (see below for defines) */
       char *ty_window; /* command to start up window manager */
       char *ty_comment;/* usually the location of the terminal */
       };

#define TTY_ON      0x1  /* enable logins (startup getty) */
#define TTY_SECURE  0x2  /* allow root to login */
#define TTY_LOCAL   0x4  /* line is local direct connect and
                            should ignore modem signals */
#define TTY_SHARED  0x8  /* line is shared - i.e. can be use
                            for both incoming and outgoing
                            connections. */
#define TTY_TRACK   0x10 /* track modem status changes */
#define TTY_TERMIO  0x20 /* open line with termio defaults */

extern struct ttyent *getttyent();
extern struct ttyent *getttynam();
```

A description of the fields follows:

ty_name is the name of the terminal's special file in the directory `/dev`.

ty_getty is the command invoked by `init` to initialize terminal line characteristics. This command is usually `getty`(8), but any arbitrary command can be used. A typical use is to initiate a terminal emulator in a window system.

ty_type is the name of the default terminal type connected to this tty line. This is typically a name from the `termcap`(5) data base. The environment variable 'TERM' is initialized with this name by `login`(1).

ty_status is a mask of bit flags that indicate various actions allowed on this terminal line. The following is a description of each flag.

TTY_ON
Enables logins. For instance, `init`(8) will start the specified `getty` command on this entry.

TTY_SECURE
Allows root to login on this terminal. TTY_ON must also be included for this to work.

TTY_LOCAL
Indicates that the line is to ignore modem signals.

TTY_SHARED
Indicates that the line can be used for both incoming and outgoing connections.

TTY_TERMIO
Indicates that a line is to be opened with default terminal attributes which are compliant with System Five termio defaults. The line discipline will be set to be TERMIODISC.

ty_window
is the quoted string of a command to execute for a window system associated with the line. If none is specified, this will be a null string.

ty_comment
Currently unused.

Restrictions

The information returned is in a static area, so you must copy it to save it. (Static areas are described in "The C Programming Language," *ULTRIX Supplementary Documents*, Vol. II:Programmers.)

Return Value

A null pointer (0) is returned on an end-of-file or error.

Files

`/etc/ttys` The file examined by these routines.

See Also

ttyname(3), ttys(5), init(8)

Name

getwd – get current working directory pathname

Syntax

char *getwd(pathname)
char *pathname;

Description

The `getwd` subroutine copies the absolute pathname of the current working directory to *pathname* and returns a pointer to the result.

Restrictions

The `getwd` subroutine may fail to return to the current directory if an error occurs.

Pathnames can be no longer than MAXPATHLEN as defined in <sys/param.h>.

Return Value

The `getwd` subroutine returns zero and places a message in *pathname* if an error occurs.

Name

hes_init, hes_to_bind, hes_error, hes_resolve - routines for using Hesiod

Syntax

#include <hesiod.h>

hes_init()

char *hes_to_bind(*HesiodName*, *HesiodNameType*)
char **HesiodName*, **HesiodNameType*;

hes_error()

har **hes_resolve(*HesiodName*, *HesiodNameType*)
char **HesiodName*, **HesiodNameType*;

Description

The `hes_init()` routine opens and reads the Hesiod configuration file, `/etc/hesiod.conf` to extract the left hand side and right hand side of the Hesiod name.

The `hes_to_bind()` routine takes as arguments a `HesiodName` and `HesiodNameType` and returns a fully qualified name to be handed to BIND.

The two most useful routines to the applications programmer are `hes_error()` and `hes_resolve()`. The `hes_error()` routine has no arguments and returns an integer which corresponds to a set of errors which can be found in `hesiod.h` file.

```
#define HES_ER_UNINIT        -1

#define HES_ER_OK             0

#define HES_ER_NOTFOUND       1

#define HES_ER_CONFIG         2

#define HES_ER_NET            3
```

The `hes_resolve()` routine resolves given names via the Hesiod name server. It takes as arguments a name to be resolved, the `HesiodName`, and a type corresponding to the name, the `HesiodNameType`, and returns a pointer to an array of strings which contains all data that matched the query, one match per array slot. The array is null terminated.

If applications require the data to be maintained throughout multiple calls to `hes_resolve()`, the data should be copied since another call to `hes_resolve()` will overwrite any previously-returned data. A null is returned if the data cannot be found.

Examples

The following example shows the use of the Hesiod routines to obtain a Hesiod name from a Hesiod database:

```
#include <hesiod.h>

char *HesiodName, *HesiodNameType;
char **hp;

hp = hes_resolve(HesiodName, HesiodNameType);
if (hp == NULL) {
       error = hes_error();
       switch(error) {
                    .
                    .
                    .
              }
       }
       else
              process(hp);
```

Files

/etc/hesiod.conf
/usr/include/hesiod.h

See Also

hesiod.conf(5), bindsetup(8)
Guide to the BIND/Hesiod Service

Name

hsearch, hcreate, hdestroy – manage hash search tables

Syntax

#include <search.h>

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

int hcreate (nel)
unsigned nel;

void hdestroy ( )

Description

The `hsearch` subroutine 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. The *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.) The *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.

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

The `hdestroy` subroutine destroys the search table, and may be followed by another call to `hcreate`.

Restrictions

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

Diagnostics

The `hsearch` subroutine 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.

The `hcreate` subroutine returns zero if it cannot allocate sufficient space for the table.

See Also

bsearch(3), lsearch(3), string(3), tsearch(3)

Name

insque, remque – insert/remove element from a queue

Syntax

struct qelem {
struct qelem *q_forw;
struct qelem *q_back;
char q_data[];
};

insque(*elem, pred*)
struct qelem **elem, *pred*;

remque(*elem*)
struct qelem **elem*;

Description

The `insque` and `remque` subroutines manipulate queues built from doubly linked lists. Each element in the queue must in the form of ''struct qelem.'' The `insque` subroutine inserts *elem* in a queue immediately after *pred*. The `remque` subroutine removes an entry *elem* from a queue.

Name

isnan – test for NaN

Syntax

#include <math.h>

int isnan (*x*)
double *x*;

Description

The `isnan` function returns 1 if *x* is NaN (the IEEE floating point reserved not-a-number value) and zero otherwise. On VAX, the return value is always zero.

Name

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

Syntax

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

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

Description

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

The `ltol3` performs the reverse conversion from long integers (*lp*) to three-byte integers (*cp*).

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

Restrictions

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

See Also

fs(5)

Name

lockf – record locking on files

Syntax

#include <unistd.h>

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

Description

The `lockf` subroutine allows sections of a file to be locked. These are advisory mode locks. Locking calls from other processes which attempt to lock the locked file section return either an error value or are put to sleep until the resource becomes unlocked. All the locks for a process are removed when the process terminates. For more information about record locking, see `fcntl`(2).

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

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

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

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

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

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

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

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

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

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

Sleeping on a resource is interrupted with any signal. You can use the `alarm`(3) command to provide a timeout facility in applications which require this facility.

File region locking is supported over NFS, if the NFS locking service has been enabled.

Restrictions

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

Return Value

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

Diagnostics

The `lockf` subroutine fails if:

[EBADF]	The *fildes* is not a valid open descriptor.
[EACCESS]	The *cmd* is F_TLOCK or F_TEST and the section is already locked by another process. Or, the file is remotely mounted, and the NFS locking service has not been enabled.
[EDEADLK]	The *cmd* is F_LOCK or F_TLOCK and a deadlock would occur. Also the *cmd* is either of the above or F_ULOCK and the number of entries in the lock table would exceed the number allocated on the system.
[EINVAL]	The value given for the *request* argument is invalid.

See Also

close(2), creat(2), fcntl(2), intro(2), open(2), read(2), write(2), lockd(8c)

Name

lsearch, lfind – linear search and update

Syntax

#include <search.h>
#include <sys/types.h>

void *lsearch (*key, base, nelp, width, compar*)
void **key*;
void **base*;
size_t **nelp*;
size_t *width*;
int (**compar*)( );

void *lfind (*key, base, nelp, width, compar*)
void **key*;
void **base*;
size_t **nelp*;
size_t *width*;
int (**compar*)( );

Description

The `lsearch` subroutine 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. The *key* points to the datum to be sought in the table. The *base* points to the first element in the table. The *nelp* points to an integer containing the current number of elements in the table. The *width* is the size of an element in bytes. The integer is incremented if the datum is added to the table. The *compar* is the name of the comparison function which the user must supply (strcmp, for example). It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal and non-zero otherwise.

The `lfind` subroutine is the same as lsearch except that if the datum is not found, it is not added to the table. Instead, a NULL pointer is returned.

NOTE

> 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.

Restrictions

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

Return Value

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

See Also

bsearch(3), hsearch(3), tsearch(3)

Name

malloc, free, realloc, calloc, alloca – memory allocator

Syntax

char *malloc(size)
unsigned size;

free(ptr)
void *ptr;

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

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

char *alloca(size)
int size;

Description

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

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

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

The `malloc` subroutine maintains multiple lists of free blocks according to size, allocating space from the appropriate list. It calls `sbrk` to get more memory from the system when there is no suitable space already free. For further information, see `brk`(2).

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

In order to be compatible with older versions, `realloc` also works if *ptr* points to a block freed since the last call of `malloc, realloc,` or `calloc.` Sequences of `free, malloc,` and `realloc` were previously used to attempt storage compaction. This procedure is no longer recommended.

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

The `alloca` subroutine allocates *size* bytes of space associated with the stack frame of the caller. This temporary space is available for reuse when the caller returns. On MIPS machines, calling `alloca`(0) reclaims all available storage. On VAX machines, the space is automatically freed on return.

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

Restrictions

When `realloc` returns 0, the block pointed to by *ptr* may be destroyed.

Currently, the allocator is unsuitable for direct use in a large virtual environment where many small blocks are kept, since it keeps all allocated and freed blocks on a circular list. Just before more memory is allocated, all allocated and freed blocks are referenced.

Because the `alloca` subroutine is machine dependent, its use should be avoided.

Diagnostics

The `malloc`, `realloc`, and `calloc` subroutines return a null pointer (0) if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block.

Name

malloc, free, realloc, calloc, alloca – memory allocator

Syntax

#include <stdlib.h>

void *malloc(*size***)**
size_t *size***;**

free(*ptr***)**
void **ptr***;**

void *realloc(*ptr, size***)**
void **ptr***;**
size_t *size***;**

void *calloc(*nelem, elsize***)**
size_t *nelem, elsize***;**

void *alloca(*size***)**
size_t *size***;**

Description

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

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

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

The `malloc` subroutine maintains multiple lists of free blocks according to size, allocating space from the appropriate list. It calls `sbrk` to get more memory from the system when there is no suitable space already free. For further information, see `brk`(2).

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

If *ptr* is a null pointer, then `realloc` behaves like `malloc` for the specified *size*. If *size* is zero, then `realloc` frees the space pointed to by *ptr*.

In order to be compatible with older versions, `realloc` also works if *ptr* points to a block freed since the last call of `malloc`, `realloc`, or `calloc`. Sequences of `free`, `malloc`, and `realloc` were previously used to attempt storage compaction. This procedure is no longer recommended.

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

The `alloca` subroutine allocates *size* bytes of space in the stack frame of the caller. This temporary space is automatically freed on return.

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

Restrictions

When `realloc` returns 0, the block pointed to by *ptr* may be destroyed.

Currently, the allocator is unsuitable for direct use in a large virtual environment where many small blocks are kept, since it keeps all allocated and freed blocks on a circular list. Just before more memory is allocated, all allocated and freed blocks are referenced.

The `alloca` subroutine is machine dependent.

Diagnostics

The `malloc`, `realloc`, and `calloc` subroutines return a null pointer (0) if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block.

The `malloc`, `realloc`, `calloc`, and `alloca` subroutines will fail and no additional memory will be allocated if one of the following is true:

[ENOMEM] The limit, as set by `setrlimit`(2), is exceeded.

[ENOMEM] The maximum possible size of a data segment (compiled into the system) is exceeded.

[ENOMEM] Insufficient space exists in the swap area to support the expansion.

Name

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

Syntax

#include <string.h>

void *memccpy (*s1, s2, c, n*)
void **s1*, **s2*;
int *c*;
size_t *n*;

void *memchr (*s, c, n*)
void **s*;
int *c*;
size_t *n*;

int memcmp (*s1, s2, n*)
void **s1*, **s2*;
size_t *n*;

void *memcpy (*s1, s2, n*)
void **s1*, **s2*;
size_t *n*;

void *memset (*s, c, n*)
void **s*;
int *c*;
size_t *n*;

void *memmove (*s1, s2, n*)
void **s1*, **s2*;
size_t *n*;

Description

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

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

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

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

The `memcpy` subroutine copies *n* characters from memory area *s2* to *s1*. It returns *s1*.

The `memmove` subroutine is like `memcpy` , except that if s1 and s2 specify overlapping areas, `memmove` works as if an intermediate buffer is used.

The `memset` subroutine sets the first *n* characters in memory area *s* to the value of character *c*. It returns *s*.

Restrictions

The `memcmp` subroutine uses native character comparison, which is signed on PDP–11s, unsigned on other machines.

Character movement is performed differently in different implementations of `memccpy` and `memcpy`. Thus overlapping moves, using these subroutines, may yield unpredictable results.

mkfifo (3)

Name

mkfifo – make a FIFO special file

Syntax

#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(path, mode)
char *path;
mode_t mode;

Description

The `mkfifo` function creates a new FIFO special file whose name is *path*. The file permission bits of the new FIFO are initialized from *mode,* where the value of *mode,* is one (or more) of the file permission bits defined in <sys/stat.h>. The *mode* argument is modified by the process's file creation mask (see `umask`(1)).

The FIFO's owner ID is set to the process's effective user ID. The FIFO's group ID is set to the process's effective group ID.

Return Value

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

Diagnostics

The `mkfifo` function will fail and the FIFO will not be created if:

[EACCES] A component of the path prefix denies search permission.

[EEXIST] The named file exists.

[ENAMETOOLONG]
A component of a pathname exceeded 255 characters, or an entire pathname exceeded 1023 characters.

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] A component of the path prefix does not exist or the *path* argument points to an empty string.

[EROFS] The named file resides on a read-only file system.

[EFAULT] *Path* points outside the process's allocated address space.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EIO] An I/O error occurred while making the directory entry.

[ENOSPC] The directory in which the entry for the new FIFO is being placed cannot be extended because there is no space left on the file system.

[ENOSPC] There are no free inodes on the file system on which the node is being created.

[EDQUOT]	The directory in which the entry for the new FIFO is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.
[EDQUOT]	The user's quota of inodes on the file system on which the FIFO is being created has been exhausted.
[ESTALE]	The file handle given in the argument is invalid. The file referred to by that file handle no longer exists or has been revoked.
[ETIMEDOUT]	A connect request or remote file operation failed because the connected party did not properly respond after a period of time which is dependent on the communications protocol.

See Also

mknod(1), umask(1)

mktemp (3)

Name

mktemp – make a unique file name

Syntax

char *mktemp(template)
char *template;

Description

The `mktemp` subroutine replaces *template* by a unique file name, and returns the address of the template. The template should look like a file name with six trailing X's, which will be replaced with the current process ID and a unique letter.

Note: The use of `mktemp` is not recommended for new applications. See `tmpnam`(3) for less error-prone alternatives.

See Also

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

Name

monitor, monstartup, moncontrol – prepare execution profile

Synopsis

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

monstartup(lowpc, highpc)
int (*lowpc)(), (*highpc)();

moncontrol(mode)

Description

These functions use the system call `profil`(2) to control program-counter sampling. Using the option **–p** when compiling or linking a program automatically generates calls to these functions. You do need not to call these functions explicitly unless you want more control.

Typically, you would call either `monitor` or `monstartup` to initialize pc-sampling and enable it; call `moncontrol` to disable or reenable it; and call `monitor` at the end of execution to disable sampling and record the samples in a file.

Your initial call to `monitor` enables pc-sampling. The parameters *lowpc* and *highpc* specify the range of addresses to be sampled. The lowest address is that of *lowpc* and the highest is just below *highpc*. The *buffer* parameter is the address of a (user allocated) array of *bufsize* short integers, which holds a record of the samples; for best results, the buffer should not be less than a few times smaller than the range of addresses sampled. The *nfunc* parameter is ignored.

The environment variable PROFDIR determines the name of the output file and whether pc-sampling takes place: if it is not set, the file is named mon.out; if set to the empty string, no pc-sampling occurs; if set to a non-empty string, the file is named string/pid.progname, where pid is the process id of the executing program and progname is the program's name as it appears in argv[0]. The subdirectory string must already exist.

To profile the entire program, use the following:

```
extern eprol(), etext();
. . .
monitor(eprol, etext, buf, bufsize, 0);
```

The routine `eprol` lies just below the user program text, and `etext` lies just above it, as described in `end`(3). (Because the user program does not necessarily start at a low memory address, using a small number in place of `eprol` is dangerous).

The `monstartup` routine is an alternate form of `monitor` that calls `sbrk` (see `brk`(2)) for you to allocate the buffer.

The function `moncontrol` selectively disables and re-enables pc-sampling within a program, allowing you to measure the cost of particular operations. The function `moncontrol(0)` disables pc-sampling, and `moncontrol(1)` reenables it.

To stop execution monitoring and write the results in the output file, use the following:

```
monitor(0);
```

Files

mon.out default name for output file
libprofl.a routines for pc-sampling

See Also

cc(1), ld(1), profil(2), brk(2)

Name

monitor, monstartup, moncontrol – prepare execution profile

Syntax

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

monstartup(lowpc, highpc)
int (*lowpc)(), (*highpc)();

moncontrol(mode)

Description

There are two different forms of monitoring available: An executable program created by:

```
cc -p . . .
```

automatically includes calls for the `prof`(1) monitor and includes an initial call to its start-up routine `monstartup` with default parameters; `monitor` need not be called explicitly except to gain fine control over profil buffer allocation. An executable program created by:

```
cc -pg . . .
```

automatically includes calls for the `gprof`(1) monitor.

The `monstartup` is a high level interface to `profil`(2). The *lowpc* and *highpc* specify the address range that is to be sampled; the lowest address sampled is that of *lowpc* and the highest is just below *highpc*. The `monstartup` subroutine allocates space using `sbrk`(2) and passes it to `monitor` (see below) to record a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions, in the buffer. Only calls of functions compiled with the profiling option **–p** of `cc`(1) are recorded.

To profile the entire program, it is sufficient to use

```
extern etext();
monstartup((int) 2, etext);
```

The *etext* lies just above all the program text, see end(3).

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

```
monitor(0);
```

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

The `moncontrol` subroutine is used to selectively control profiling within a program. This works with either `prof`(1) or `gprof`(1) type profiling. When the program starts, profiling begins. To stop the collection of histogram ticks and call counts use moncontrol(0); to resume the collection of histogram ticks and call counts use moncontrol(1). This allows the cost of particular operations to be measured. Note that an output file will be produced upon program exit regardless of the state of `moncontrol.`

The `monitor` subroutine is a low level interface to `profil`(2). The *lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a (user supplied) array of *bufsize* short integers. At most *nfunc* call counts can be kept. For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled. The `monitor` subroutine divides the buffer into space to record the histogram of program counter samples over the range *lowpc* to *highpc*, and space to record call counts of functions compiled with the **–p** option to `cc`(1).

To profile the entire program, it is sufficient to use

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

Files

mon.out

See Also

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

Name

dbm_open, dbm_close, dbm_fetch, dbm_store, dbm_delete, dbm_firstkey, dbm_nextkey, dbm_error, dbm_clearerr – data base subroutines

Syntax

```
#include <ndbm.h>

typedef struct {
   char *dptr;
   int dsize;
} datum;

DBM *dbm_open(file, flags, mode)
   char *file;
   int flags, mode;

void dbm_close(db)
   DBM *db;

datum dbm_fetch(db, key)
   DBM *db;
   datum key;

int dbm_store(db, key, content, flags)
   DBM *db;
   datum key, content;
   int flags;

int dbm_delete(db, key)
   DBM *db;
   datum key;

datum dbm_firstkey(db)
   DBM *db;

datum dbm_nextkey(db)
   DBM *db;

int dbm_error(db)
   DBM *db;

int dbm_clearerr(db)
   DBM *db;
```

Description

These functions maintain key/content pairs in a data base. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two file system accesses. This package replaces the earlier dbm(3x) library, which managed only a single database.

The *keys* and *contents* are described by the **datum** typedef. A **datum** specifies a string of **dsize** bytes pointed to by **dptr.** Arbitrary binary data, as well as normal ASCII strings, are allowed. The data base is stored in two files. One file is a directory containing a bit map 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 will only insert new entries into the database and will not change an existing entry with the same key. DBM_REPLACE will replace 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 may be made, in an (apparently) random order, by use of **dbm_firstkey** and **dbm_nextkey**. **dbm_firstkey** will return the first key in the database. **dbm_nextkey** will return the next key in the database. This code will traverse the data base:

> **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. A zero return indicates ok. Routines that return a **datum** indicate errors with a null (0) **dptr.** If **dbm_store** called with a *flags* value of DBM_INSERT finds an existing entry with the same key it returns 1.

Restrictions

The '.pag' file will contain holes so that its apparent size is about four times its actual content. Older systems may create real file blocks for these holes when touched. These files cannot be copied by normal means (`cp`, `cat`, `tp`, `tar`, `ar`) without filling in the holes.

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 4096 bytes). Moreover all key/content pairs that hash together must fit on a single block. **dbm_store** will return 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.

See Also

`dbm`(3X)

Name

nice – set program priority

Syntax

nice(incr)

Description

The scheduling priority of the process is augmented by *incr*. Positive priorities get less service than normal. Priority 10 is recommended to users who wish to execute long-running programs without flack from the administration.

Negative increments are ignored except on behalf of the super-user. The priority is limited to the range –20 (most urgent) to 20 (least).

The priority of a process is passed to a child process by `fork`(2). For a privileged process to return to normal priority from an unknown state, `nice` should be called successively with arguments –40 (goes to priority –20 because of truncation), 20 (to get to 0), then 0 (to maintain compatibility with previous versions of this call).

Environment

When your program is compiled using the System V environment, upon success, `nice` returns –20.

See Also

nice(1), fork(2), setpriority(2), renice(8)

Name

nlist – get entries from name list

Syntax

#include <nlist.h>

nlist(filename, nl)
char *filename;
struct nlist nl[];

Description

The `nlist` subroutine examines the name list in the given executable output file and selectively extracts a list of values. The name list consists of an array of structures containing names, types and values. The list is terminated with a null name. Each name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the next two fields. If the name is not found, both entries are set to 0. See `a.out`(5) for the structure declaration.

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

Diagnostics

If the file cannot be found or if it is not a valid namelist –1 is returned; otherwise, the number of unfound namelist entries is returned.

The type entry is set to 0 if the symbol is not found.

See Also

a.out(5)

Name

nlist – get entries from name list

Syntax

#include <nlist.h>

nlist(filename, nl)
char *filename;
struct nlist nl[];

Description

The `nlist` subroutine examines the name list in the given executable output file and selectively extracts a list of values. The name list consists of an array of structures containing names, types and values. The list is terminated with a null name. Each name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the next two fields. If the name is not found, both entries are set to 0. See `a.out`(5) for the structure declaration.

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

Diagnostics

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

See Also

a.out(5)

Name

pathconf, fpathconf – get configurable pathname variables (POSIX)

Syntax

#include <unistd.h>

long pathconf(*path, name***)**
char **path***;**
int *name***;**

long fpathconf(*fildes, name***)**
int *fildes, name***;**

Description

The `pathconf(3)` and `fpathconf(3)` functions provide a method for the application to determine the current value of a configurable limit or option that is associated with a file or directory.

For `pathconf(3)`, the *path* argument points to the pathname of a file or directory. For `fpathconf(3)`, the *fildes* argument is an open file descriptor.

The *name* argument represents the variable to be queried relative to that file or directory. The following table lists the variables which may be queried and the corresponding value for the *name* argument. The values for the *name* argument are defined in the <unistd.h> header file.

Variable	name Value
LINK_MAX	_PC_LINK_MAX
MAX_CANON	_PC_MAX_CANON
MAX_INPUT	_PC_MAX_INPUT
NAME_MAX	_PC_NAME_MAX
PATH_MAX	_PC_PATH_MAX
PIPE_BUF	_PC_PIPE_BUF
_POSIX_CHOWN_RESTRICTED	_PC_CHOWN_RESTRICTED
_POSIX_NO_TRUNC	_PC_NO_TRUNC
_POSIX_VDISABLE	_PC_VDISABLE

Return Value

Upon successful completion, the `pathconf(3)` and `fpathconf(3)` functions return the current variable value for the file or directory.

If *name* is an invalid value, `pathconf(3)` and `fpathconf(3)` return –1 and *errno* is set to indicate the reason. If the variable corresponding to *name* is not defined on the system, `pathconf(3)` and `fpathconf(3)` return –1 without changing the value of *errno*.

Diagnostics

The `pathconf`(3) and `fpathconf`(3) functions fail if the following occurs:

[EINVAL] The value of the *name* argument is invalid.

See Also

<unistd.h>

pause (3)

Name

pause – stop until signal

Syntax

pause()

Description

The `pause` subroutine never returns normally. It is used to give up control while waiting for a signal from `kill`(2) or an interval timer, see `setitimer`(2). Upon termination of a signal handler started during a `pause,` the `pause` call will return.

Diagnostics

The `pause` subroutine always returns:

[EINTR] The call was interrupted, that is, always returns –1.

See Also

kill(2), select(2), sigpause(2)

Name

perror, strerror, sys_errlist, sys_nerr – system error messages

Syntax

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

int sys_nerr;
char *sys_errlist[];

#include <string.h>

char *strerror(*err*)
int *err;*

Description

The `perror` subroutine produces a short error message on the standard error file describing the last error encountered during a call to the system from a C program. First the argument string *s* , if it is not a null pointer, is printed followed by a colon and a space; then the message and a new line are printed. Most usefully, the argument string is the name of the program which incurred the error. The error number is taken from the external variable *errno* which is set when errors occur but not cleared when nonerroneous calls are made. For further information, see `intro`(2).

To simplify variant formatting of messages, the vector of message strings *sys_errlist* is provided; *errno* can be used as an index in this table to get the message string without the new line. The *sys_nerr* is the number of messages provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table. The `strerror` function will also return a pointer to the message text for a given error number.

See Also

intro(2), `errno`(2), psignal(3)

Name

pfopen – open a packet filter file

Syntax

pfopen(ifname, flags)
char *ifname;
int flags;

Description

The packet filter (see `packetfilter(4)`) provides raw access to Ethernets and similar network data link layers. The routine `pfopen` is used to open a packet filter file descriptor. The routine hides various details about the way packet filter files are opened and named.

The *ifname* argument is a pointer to a null-terminated string containing the name of the interface for which the application is opening the packet filter. This name may be the name of an actual interface on the system (for example, ''de0'', ''qe2'') or it may be a pseudo-interface name of the form ''pf*n*'', used to specify the *n*th interface attached to the system. For example, ''pf0'' specifies the first such interface. If *ifname* is NULL, the default interface (''pf0'') is used.

The *flags* argument has the same meaning as the corresponding argument to the `open(2)` system call.

The file descriptor returned by `pfopen` is otherwise identical to one returned by `open(2)`.

Diagnostics

The `pfopen` routine returns a negative integer if the file could not be opened. This may be because of resource limitations, or because the specified interface does not exist.

If there are a lot of packet filter applications in use, the `pfopen` routine might take a while.

See Also

open(2), packetfilter(4)
The Packet Filter: An Efficient Mechanism for User Level Network Code

Name

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

Syntax

#include <stdio.h>

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

pclose(stream)
FILE *stream;

Description

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

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

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

Environment

Differs from the System V definition in that ENFILE is not a possible error condition.

Diagnostics

The `popen` routine returns a null pointer if files or processes cannot be created, or the shell cannot be accessed.

The `pclose` routine returns –1 if *stream* is not associated with a 'popened' command.

Restrictions

Buffered reading before opening an input filter may leave the standard input of that filter mispositioned. Similar problems with an output filter may be forestalled by careful buffer flushing, for instance, with `fflush`. For further information, see `fclose`(3).

The `popen` routine always calls `sh,` and never calls `csh`.

See Also

sh(1), pipe(2), wait(2), system(3), fclose(3s), fopen(3s)

Name

psignal, sys_siglist – system signal messages

Syntax

psignal(sig, s)
unsigned sig;
char *s;

char *sys_siglist[];

Description

The `psignal` subroutine produces a short message on the standard error file describing the indicated signal. First the argument string *s* is printed, then a colon, then the name of the signal and a new-line. Most usefully, the argument string is the name of the program which incurred the signal. The signal number should be from among those found in <signal.h>.

To simplify variant formatting of signal names, the vector of message strings `sys_siglist` is provided. The signal number can be used as an index in this table to get the signal name without the newline. The define NSIG defined in <signal.h> is the number of messages.

See Also

sigvec(2), perror(3)

Name

putenv – change or add value to environment

Syntax

int putenv (string)
char *string;

Description

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

Diagnostics

The `putenv` subroutine returns nonzero if it was unable to obtain enough space via `malloc` for an expanded environment, otherwise zero.

Warnings

The `putenv` subroutine 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`(3) to enlarge the environment.
After `putenv` is called, environmental variables are not in alphabetical order.
A potential error is to call `putenv` with an automatic variable as the argument, then exit the calling function while *string* is still part of the environment.

See Also

execve(2), getenv(3), malloc(3), environ(7)

Name

putpwent – write password file entry

Syntax

#include <pwd.h>

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

Description

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

Diagnostics

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

Caution

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

Name

qsort – quicker sort

Syntax

#include <stdlib.h>

void qsort(*base, nel, width, compar***)**
void ****base*;**
size_t *nel, width***;**
int (**compar***)();**

Description

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

See Also

sort(1)

rand(3)

Name

rand, srand – random number generator

Syntax

#include <stdlib.h>

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

rand()

Description

The newer `random(3)` should be used in new applications. The `rand` subroutine remains for compatibility.

The `rand` subroutine uses a multiplicative congruential random number generator with period 2^{32} to return successive pseudo-random numbers in the range from 0 to $2^{31}-1$.

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

Environment

For the System V environment, the `rand` subroutine returns numbers in the range from 0 to $2^{15}-1$.

See Also

random(3)

Name

random, srandom, initstate, setstate – better random number generator; routines for changing generators

Syntax

long random()

srandom(seed)
int seed;

char *initstate(seed, state, n)
unsigned seed;
char *state;
int n;

char *setstate(state)
char *state;

Description

The `random` subroutine uses a non-linear additive feedback random number generator employing a default table of size 31 long integers to return successive pseudo-random numbers in the range from 0 to (2**31)-1. The period of this random number generator is very large, approximately 16*((2**31)-1).

The `random/srandom` subroutines have (almost) the same calling sequence and initialization properties as `rand/srand`. The difference is that rand(3) produces a much less random sequence – in fact, the low dozen bits generated by rand go through a cyclic pattern. All the bits generated by `random` are usable. For example, ''random()&01'' will produce a random binary value.

Unlike `srand`, `srandom` does not return the old seed; the reason for this is that the amount of state information used is much more than a single word. (Two other routines are provided to deal with restarting/changing random number generators.) Like `rand`(3), however, `random` will by default produce a sequence of numbers that can be duplicated by calling `srandom` with *1* as the seed.

The `initstate` routine allows a state array, passed in as an argument, to be initialized for future use. The size of the state array (in bytes) is used by `initstate` to decide how sophisticated a random number generator it should use – the more state, the better the random numbers will be. (Current "optimal" values for the amount of state information are 8, 32, 64, 128, and 256 bytes; other amounts will be rounded down to the nearest known amount. Using less than 8 bytes will cause an error). The seed for the initialization (which specifies a starting point for the random number sequence, and provides for restarting at the same point) is also an argument. `Initstate` returns a pointer to the previous state information array.

Once a state has been initialized, the `setstate` routine provides for rapid switching between states. The `setstate` subroutine returns a pointer to the previous state array; its argument state array is used for further random number generation until the next call to `initstate` or `setstate`.

Once a state array has been initialized, it may be restarted at a different point either by calling `initstate` (with the desired seed, the state array, and its size) or by calling both `setstate` (with the state array) and `srandom` (with the desired seed).

The advantage of calling both `setstate` and `srandom` is that the size of the state array does not have to be remembered after it is initialized.

With 256 bytes of state information, the period of the random number generator is greater than 2^{69}, which should be sufficient for most purposes.

Diagnostics

If `initstate` is called with less than 8 bytes of state information, or if `setstate` detects that the state information has been garbled, error messages are printed on the standard error output.

See Also

rand(3)

Name

re_comp, re_exec – regular expression handler

Syntax

char *re_comp(s)
char *s;

re_exec(s)
char *s;

Description

The `re_comp` subroutine compiles a string into an internal form suitable for pattern matching. The `re_exec` subroutine checks the argument string against the last string passed to `re_comp`.

The `re_comp` subroutine returns 0 if the string *s* was compiled successfully; otherwise a string containing an error message is returned. If `re_comp` is passed 0 or a null string, it returns without changing the currently compiled regular expression.

The `re_exec` subroutine returns 1 if the string *s* matches the last compiled regular expression, 0 if the string *s* failed to match the last compiled regular expression, and –1 if the compiled regular expression was invalid (indicating an internal error).

The strings passed to both `re_comp` and `re_exec` may have trailing or embedded newline characters; they are terminated by nulls. The regular expressions recognized are described in the manual entry for `ed`(1), given the above difference.

Diagnostics

The `re_exec` subroutine returns –1 for an internal error.

The `re_comp` subroutine returns one of the following strings if an error occurs:

No previous regular expression
Regular expression too long
unmatched \(
missing]
too many \(\) pairs
unmatched \)

See Also

ed(1), ex(1), egrep(1), fgrep(1), grep(1)

remove(3)

Name

remove – removes files

Syntax

remove (*path*)
char *path;

Arguments

path	Provides the specification for a file or directory.

Description

The `remove` library function removes a file. If the *path* does not name a directory then *remove(path)* is equivalent to *unlink(path)*. If the *path* does name a directory then *remove(path)* is equivalent to *rmdir(path)*.

Return Value

A 0 is returned if the remove succeeds; otherwise a –1 is returned and an error code is stored in the global location *errno*.

See Also

errno(2), rmdir(2), unlink(2)

Name

res_mkquery, res_send, res_init, dn_comp, dn_expand – resolver routines

Syntax

#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

res_mkquery(*op, dname, class, type, data, datalen, newrr, buf, buflen***)**
int *op***;**
char ****dname*;**
int *class, type***;**
char ****data*;**
int *datalen***;**
struct rrec ****newrr*;**
char ****buf*;**
int *buflen***;**

res_send(*msg, msglen, answer, anslen***)**
char ****msg*;**
int *msglen***;**
char ****answer*;**
int *anslen***;**

res_init()

dn_comp(*exp_dn, comp_dn, length, dnptrs, lastdnptr***)**
char ****exp_dn, *comp_dn*;**
int *length***;**
char *****dnptrs, **lastdnptr*;**

dn_expand(*msg, eomorig, comp_dn, exp_dn, length***)**
char ****msg, *eomorig, *comp_dn, exp_dn*;**
int *length***;**

Description

The resolver routines are used for making, sending, and interpreting packets to BIND servers. Global information that is used by the resolver routines is kept in the variable `_res`. Most of the values have reasonable defaults and you need not be concerned with them. The options are a simple bit mask and are or'ed in to enable. The options stored in `_res.options` are defined in `/usr/include/resolv.h` and are as follows:

RES_INIT	True if the initial name server address and default domain name are initialized, for example if `res_init` has been called.
RES_DEBUG	Print debugging messages.
RES_AAONLY	Accept authoritative answers only.
RES_USEVC	Use TCP connections for queries instead of UDP.
RES_STAYOPEN	This is used with `RES_USEVC` to keep the TCP connection

open between queries. This is useful only in programs that regularly do many queries. You should normally use UDP.

RES_RECURSE Set the recursion desired bit in queries. This is the default. The `res_send` routine does not do iterative queries and expects the BIND server to handle recursion.

RES_DEFNAMES Append the default domain name to single label queries. This is the default.

The following lists the routines found in `/usr/lib/libc.a`

res_init This routine reads the initialization file to get the default domain name and the Internet address of the initial hosts running the name server. If this line does not exist, the host running the resolver is tried.

res_mkquery This routine makes a standard query message and places it in *buf*. The *res_mkquery* routine returns the size of the query or –1 if the query is larger than *buflen*.

op The opcode is usually `QUERY`, but can be any of the query types defined in *nameser.h*.

Dname
This variable is the domain name. If *dname* consists of a single label and the `RES_DEFNAMES` flag is enabled, which is the default, *dname* is appended with the current domain name. The current domain name is defined in a system file, but you can override it by using the environment variable `LOCALDOMAIN`.

res_send This routine sends a query to the BIND servers and returns an answer. It calls the `res_init` routine. If `RES_INIT` is not set, `res_send` sends the query to the local name server, and handle timeouts and retries. The length of the message is returned or –1 if there were errors.

dn_comp This routine 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. The *length* is the size of the array pointed to by *comp_dn*.

dnptrs
This variable is a list of pointers to previously compressed names in the current message. The first pointer points to the beginning of the message and the list ends with `NULL`.

lastdnptr
This is a pointer to the end of the array pointed to by *dnptrs*. A side effect is to update the list of pointers for labels inserted into the message by *dn_comp* as the name is compressed. If *dnptr* is NULL, the names are not compressed. If *lastdnptr* is NULL, the list is not updated.

dn_expand This routine expands the compressed domain name `comp_dn` to a full BIND domain name. Expanded names are converted to upper case.

msg This variable is a pointer to the beginning of the message.

exp_dn
This variable is a pointer to a buffer of size *length* for the result. The size of the compressed name is returned or -1 if there was an error.

Files

```
/etc/resolv.conf
/usr/include/resolv.h
/usr/include/arpa/nameser.h
```

See Also

named(8), resolv.conf(5)
Guide to the BIND/Hesiod Service

Name

scandir – scan a directory

Syntax

#include <sys/types.h>
#include <sys/dir.h>

scandir(dirname, namelist, select, compar)
char *dirname;
struct direct *(*namelist[]);
int (*select)();
int (*compar)();

alphasort(d1, d2)
struct direct **d1, **d2;

Description

The `scandir` subroutine reads the directory *dirname* and builds an array of pointers to directory entries using `malloc`(3). 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`(3) to sort the completed array. If this pointer is null, the array is not sorted. The `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* by freeing each pointer in the array and the array itself. For further information, see `malloc`(3).

Diagnostics

Returns –1 if the directory cannot be opened for reading or if `malloc`(3) cannot allocate enough memory to hold all the data structures.

See Also

directory(3), malloc(3), qsort(3), dir(5)

Name

setjmp, longjmp – non-local goto

Syntax

#include <setjmp.h>

int setjmp (env)
jmp_buf env;

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

Description

The `setjmp` and `longjmp` functions help deal with errors and interrupts encountered in a low-level subroutine of a program.

The `setjmp` function 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.

The `longjmp` function restores the environment saved by the last call of `setjmp` with the corresponding *env* argument. After `longjmp` finishes, 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*. The `longjmp` function cannot cause `setjmp` to return the value 0. If `longjmp` is invoked with a second argument of 0, `setjmp` returns 1. At the time of the second return from `setjmp`, all accessible data have values as of the time `longjmp` is called. However, global variables have the expected values. For example, those as of the time of the `longjmp`(see

Examples

```
#include <setjmp.h>

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

        if(setjmp(env) != 0) {
                (void) printf("value of i on 2nd return from setjmp: %d0, i);
                exit(0);
        }
        (void) printf("value of i on 1st return from setjmp: %d0, i);
        i = 1;
        g();
        /*NOTREACHED*/
}
```

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

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

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

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

NOTE

Unexpected behavior occurs if `longjmp` is called without a previous call to `setjmp`, or when the last such call was in a function which has since returned.

Restrictions

The values of the registers on the second return from `setjmp` are register values at the time of the first call to `setjmp`, not those of the `longjmp`. Thus, variables in a given function can produce unexpected results in the presence of `setjmp`, depending on whether they are register or stack variables.

See Also

signal(2).

Name

setjmp, longjmp – nonlocal goto

Syntax

#include <setjmp.h>

setjmp(env)
jmp_buf env;

longjmp(env, val)
jmp_buf env;

_setjmp(env)
jmp_buf env;

_longjmp(env, val)
jmp_buf env;

Description

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

The `setjmp` subroutine saves its stack environment in *env* for later use by `longjmp`. It returns value 0.

The `longjmp` subroutine restores the environment saved by the last call of `setjmp`. It then returns in such a way that execution continues as if the call of `setjmp` had just returned the value *val* to the function that invoked `setjmp`, which must not itself have returned in the interim. However, `longjmp` cannot cause `setjmp` to return the value 0. If `longjmp` is invoked with a *val* of 0, `setjmp` will return 1. All accessible data have values as of the time `longjmp` was called.

The `setjmp` and `longjmp` subroutines save and restore the signal mask sigsetmask(2), while `_setjmp` and `_longjmp` manipulate only the C stack and registers.

Restrictions

The `setjmp` subroutine does not save current notion of whether the process is executing on the signal stack. The result is that a longjmp to some place on the signal stack leaves the signal stack state incorrect.

See Also

sigstack(2), sigvec(2), signal(3)

Name

setlocale – set localization for internationalized program

Syntax

#include <locale.h>

char *setlocale (*category*, *locale*)
int category;
char *locale;

Description

The `setlocale` function changes or queries the run-time environment of the program. The function can affect the settings of language, territory, and codeset in the program's environment.

In the *category* argument, you specify what part of the run-time environment you want to affect. Possible values for *category* are shown in the following table:

category	**Effect of Specifying the Value**	**Environment Variable Affected**
LC_ALL	Sets or queries entire environment	LANG
LC_COLLATE	Changes or queries collation sequences	LC_COLLATE
LC_CTYPE	Changes or queries character classification	LC_CTYPE
LC_NUMERIC	Changes or queries number format information	LC_NUMERIC
LC_TIME	Changes or queries time conversion parameters	LC_TIME
LC_MONETARY	Changes or queries monetary information	LC_MONETARY

You change only one part of the program's locale in a single call to `setlocale`, unless you use the category `LC_ALL`.

The *locale* argument is a pointer to a character string containing the required setting of *category* in the following format:

```
language[_territory[.codeset]][@modifier]
```

You use *language* to specify the native language you want in the program environment. You can specify what dialect of the native language you want in *_territory*, and the codeset to be used in *codeset*. For example, the following string specifies the French native language, as spoken in France (as opposed to Switzerland), and the Digital Multinational Character Set:

```
LANG = FRE_FR.MCS
```

You use *@modifier* to select a specific instance of an environment setting within a single category. For example, you could use *@modifier* to select dictionary sorting of data, as opposed to telephone directory sorting. You can use *@modifier* for all categories, except `LC_ALL`.

The following preset values of *locale* are defined for all the settings of *category*:

"C" Specifies setting the locale to the minimum C language environment, as specified by the ANSI standard for the C language. (Draft ANSI X3.159)

"" Specifies using the environment variable corresponding to *category* to set the locale. If the appropriate environment variable is not set, the `LANG` environment variable is used. If `LANG` is not set, `setlocale` returns an error.

NULL Queries the current international environment and returns current locale setting. You can use the string `setlocale` returns only as input to a subsequent `setlocale` call; in particular, the string cannot be printed for category `LC_ALL`. The string `setlocale` returns is a pointer to static data area that might be written over.

International Environment

INTLINFO The INTLINFO environment variable specifies the directory to search for language databases. The default is to search the `/usr/lib/intln` directory.

Examples

The following calls to the `setlocale` function set the environment to the French language and then modify the collating sequence to German dictionary collation:

```
setlocale (LC_ALL, "FRE_FR.MCS");
setlocale (LC_COLLATE, "GER_DE.MCS@dict");
```

You can use the `setlocale` function to bind the specific language requirements of a user to the program as follows:

```
status = setlocale (LC_ALL, "");
```

For this example to work properly, the user of the international program sets the LANG variable before running the program. Once LANG is set and the program runs, this call causes `setlocale` to use the definition of LANG to set the current locale. You should test the value of `status` after the call completes to be sure no errors occur.

Return Values

If you pass valid setting for *category* and *locale*, other than NULL, `setlocale` changes the current locale and returns the string associated with that locale.

If *locale* is NULL, `setlocale` returns the string associated with *category* for the current *locale*. The current *locale* is unchanged. The string `setlocale` returns may not be in a printable format.

If either the *category* or *locale* argument is invalid, `setlocale` returns `NULL`. The `setlocale` function does not modify the locale if any part of the call is invalid.

The `setlocale` function stores its return values in a data area that may be written over. You should move the return value to another location if you want to use it in your program.

See Also

ic(1int), nl_langinfo(3int), printf(3int), environ(5int), lang(5int)
Guide to Developing International Software

Name

setpgid – set process group (POSIX)

Syntax

#include <sys/types.h>
int
setpgid(pid, pgrp)
pid_t pid, pgrp;

Description

The `setpgid` function is used to either join an existing process group or create a new process group within the session of the calling process (see `setsid`(2)). Upon successful completion, the process group ID of the process that has a process ID which matches *pid* is set to *pgrp*. If *pid* is zero, then the call applies to the current process. In addition, if *pgrp* is zero, the process ID of the indicated process is used.

This function is available only in the POSIX environment.

Return Value

The `setpgid` function returns 0 when the operation is successful. If the request fails, –1 is returned and the global variable `errno` indicates the reason.

Diagnostics

The `setpgid` function fails and the process group is not altered if one of the following occurs:

[EACCES] The value of the *pid* argument matches the process ID of a child process of the calling process and the child process has successfully executed an `exec` function.

[EINVAL] The value of the *pgrp* argument is less than zero or is not a supported value.

[EPERM] The process indicated by the *pid* argument is a session leader.

The value of the *pid* argument matches the process ID of a child process of the calling process and the child process is not in the same session as the calling process.

The value of the *pgrp* argument does not match the process ID of the process indicated by the *pid* argument and there is no process with a process group ID that matches the value of the *pgrp* argument in the same session as the calling process.

[ESRCH] The value of the *pid* argument does not match the process ID of the calling process of a child process of the calling process.

See Also

getpgrp(2), setsid(2)

Name

setuid, seteuid, setruid, setgid, setegid, setrgid – set user and group ID

Syntax

#include <*sys/types.h*>
#include <*unistd.h*>

setuid(*uid*)
uid_t *uid*;
seteuid(*euid*)
uid_t *euid*;
setruid(*ruid*)
uid_t *ruid*;

setgid(*gid*)
gid_t *gid*;
setegid(*egid*)
gid_t *egid*;
setrgid(*rgid*)
gid_t *rgid*;

Description

The `setuid` subroutine sets both the real and effective user ID of the current process to the ID specified. Likewise, the `setgid` subroutine sets the real and effective group ID of the current process to the ID specified.

The `seteuid` subroutine sets the effective user ID of the current process, while the `setegid` subroutine sets the effective group ID of the current process.

The `setruid` subroutine sets the real user ID of the current process, while the `setrgid` subroutine sets the real group ID of the current process.

These calls are only permitted to the super-user or if the argument is the real or effective ID.

Environment

POSIX
SYSTEM_FIVE
When your program is compiled in POSIX or System V mode the following semantics apply when using the `setuid` or `setgid` functions:

If the process is the super-user the real, effective, and saved set (as described in `execve`(2)) user/group ID are set to *uid*.

If the process is not the super-user, but *uid* is equal to the real or the saved set user/group ID, the effective user/group ID is set to *uid*. The real and saved set user/group ID remain unchanged.

POSIX
In POSIX mode, the `setuid` function returns a value of type uid_t. The `setgid` function returns a value of type gid_t.

Return Values

Zero is returned if the user ID or group ID is set; –1 is returned otherwise.

See Also

setreuid(2), setregid(2), getuid(2), getgid(2)

Name

sigaction – software signal facilities (POSIX)

Syntax

```
#include <signal.h>

struct sigaction {
        void      (*sa_handler)();
        sigset_t sa_mask;
        int       sa_flags;
};

int sigaction(sig, vec, ovec)
int sig;
struct sigaction *vec, *ovec;
```

Description

The sigaction call is the POSIX equivalent to the `sigvec(2)` system call. This call behaves as described on the `sigvec(2)` reference page with the following modifications:

- The signal mask is manipulated using the `sigsetops(3)` functions.
- A process can suppress the generation of the SIGCHLD when a child stops by setting the SA_NOCLDSTOP bit in *sa_flags*.
- The SV_INTERRUPT flag is always set by the system when using `sigaction(3)` in POSIX mode. The flag is set so that interrupted system calls will fail with the EINTR error instead of getting restarted.

Return Value

A 0 return value indicated that the call succeeded. A –1 return value indicates an error occurred and *errno* is set to indicated the reason.

Diagnostics

The `sigaction` system call fails and a new signal handler is not installed if one of the following occurs:

[EFAULT] Either *vec* or *ovec* points to memory which is not a valid part of the process address space.

[EINVAL] *Sig* is not a valid signal number.

[EINVAL] An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

See Also

sigvec(2), sigsetops(3), sigprocmask(3), sigsuspend(3), sigpending(2), setjmp(3), siginterrupt(3), tty(4)

siginterrupt(3)

Name

siginterrupt – allow signals to interrupt system calls

Syntax

siginterrupt(*sig, flag*)
int *sig, flag*;

Description

The `siginterrupt` system call is used to change the system call restart behavior when a system call is interrupted by the specified signal. If the flag is false (0), then system calls will be restarted if they are interrupted by the specified signal and no data has been transferred yet. System call restart is the default behavior on 4.2 BSD.

If the flag is true (1), then restarting of system calls is disabled. If a system call is interrupted by the specified signal and no data has been transferred, the system call will return -1 with errno set to EINTR. Interrupted system calls that have started transferring data will return the amount of data actually transferred. System call interrupt is the signal behavior found on 4.1 BSD and AT&T System V systems.

Note that the new signal handling semantics are not altered in any other way. Most notably, signal handlers always remain installed until explicitly changed by a subsequent `sigvec`(2) call, and the signal mask operates as documented in `sigvec`(2.) Programs may switch between restartable and interruptible system call operation as often as desired in the execution of a program.

Issuing a `siginterrupt` call during the execution of a signal handler will cause the new action to take place on the next signal to be caught.

Environment

This library routine uses an extension of the `sigvec`(2) system call that is not available in ULTRIX 2.0 or earlier versions. Hence it should not be used if backward compatibility is needed.

Return Value

A 0 value indicates that the call succeeded. A -1 value indicates that an invalid signal number has been supplied.

See Also

sigvec(2), sigblock(2), sigpause(2), sigsetmask(2)

Name

signal – simplified software signal facilities

Syntax

#include <signal.h>

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

Description

The `signal` subroutine is a simplified interface to the more general sigvec(2) facility.

A signal is generated by some abnormal event, initiated by a user at a terminal (quit, interrupt, stop), by a program error (bus error, etc.), by request of another program (kill), or when a process is stopped because it wishes to access its control terminal while in the background. For further information, see `tty`(4). Signals are optionally generated when a process resumes after being stopped, when the status of child process changes, or when input is ready at the control terminal. Most signals cause termination of the receiving process if no action is taken; some signals instead cause the process receiving them to be stopped, or are simply discarded if the process has not requested otherwise. Except for the SIGKILL and SIGSTOP signals, the `signal` call allows signals either to be ignored or to cause an interrupt to a specified location. The following is a list of all signals with names as in the include file < signal.h >:

SIGHUP	1	Hangup
SIGINT	2	Interrupt
SIGQUIT	3*	Quit
SIGILL	4*	Illegal instruction
SIGTRAP	5*	Trace trap
SIGIOT	6*	IOT instruction
SIGEMT	7*	EMT instruction
SIGFPE	8*	Floating point exception
SIGKILL	9	Kill (cannot be caught or ignored)
SIGBUS	10*	Bus error
SIGSEGV	11*	Segmentation violation
SIGSYS	12*	Bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	Alarm clock
SIGTERM	15	Software termination signal
SIGURG	16•	Urgent condition present on socket
SIGSTOP	17+	Stop (cannot be caught or ignored)
SIGTSTP	18+	Stop signal generated from keyboard
SIGCONT	19•	Continue after stop
SIGCHLD	20•	Child status has changed
SIGTTIN	21+	Background read attempted from control terminal
SIGTTOU	22+	Background write attempted to control terminal
SIGIO	23•	I/O is possible on a descriptor (see fcntl(2))
SIGXCPU	24	Cpu time limit exceeded (see setrlimit(2))
SIGXFSZ	25	File size limit exceeded (see setrlimit(2))

SIGVTALRM	26	Virtual time alarm (see setitimer(2))
SIGPROF	27	Profiling timer alarm (see setitimer(2))
SIGWINCH	28•	Window size change
SIGUSR1	30	User defined signal
SIGUSR2	31	User defined signal
SIGCLD		System V name for SIGCHLD
SIGABRT		X/OPEN name for SIGIOT

The starred signals in the list above cause a core image if not caught or ignored.

If *func* is SIG_DFL, the default action for signal *sig* is reinstated; this default is termination (with a core image for starred signals) except for signals marked with • or +. Signals marked with • are discarded if the action is SIG_DFL; signals marked with + cause the process to stop. If *func* is SIG_IGN the signal is subsequently ignored and pending instances of the signal are discarded. Otherwise, when the signal occurs further occurrences of the signal are automatically blocked and *func* is called.

A return from the function unblocks the handled signal and continues the process at the point it was interrupted. Unlike previous signal facilities, the handler *func* remains installed after a signal has been delivered.

If a caught signal occurs during certain system calls, causing the call to terminate prematurely, the call is automatically restarted. In particular this can occur during a *read* or `write`(2) on a slow device (such as a terminal; but not a file) and during a `wait`(2).

The value of `signal` is the previous (or initial) value of *func* for the particular signal.

After a `fork`(2) or `vfork`(2) the child inherits all signals. The `execve`(2) system call resets all caught signals to the default action; ignored signals remain ignored.

Environment

When your program is compiled using the System V environment the handler function does NOT remain installed after the signal has been delivered.

Also, when a signal which is to be caught occurs during a read, write, or ioctl to a slow device (like a terminal, but not a file); or during a pause; or wait that does not return immediately, the signal handler function is executed, and then the interrupted system call may return a -1 to the calling process with errno set to EINTR.

Notes

The handler routine can be declared as follows:

```
handler(sig, code, scp)
int sig, code;
struct sigcontext *scp;
```

Here *sig* is the signal number. The MIPS hardware exceptions are mapped to specific signals as defined by the table below. The parameter *code* is either a constant as given below or zero. The parameter *scp* is a pointer to the *sigcontext* structure (defined in <*signal.h*>), that is the context at the time of the signal and is used to restore the context if the signal handler returns.

The following defines the mapping of MIPS hardware exceptions to signals and codes. All of these symbols are defined in either *<signal.h>* or *<mips/cpu.h>*:

Hardware exception	Signal	Code
Integer overflow	SIGFPE	EXC_OV
Segmentation violation	SIGSEGV	SEXC_SEGV
Illegal Instruction	SIGILL	EXC_II
Coprocessor Unusable	SIGILL	SEXC_CPU
Data Bus Error	SIGBUS	EXC_DBE
Instruction Bus Error	SIGBUS	EXC_IBE
Read Address Error	SIGBUS	EXC_RADE
Write Address Error	SIGBUS	EXC_WADE
User Breakpoint (used by debuggers)	SIGTRAP	BRK_USERBP
Kernel Breakpoint (used by prom)	SIGTRAP	BRK_KERNELBP
Taken Branch Delay Emulation	SIGTRAP	BRK_BD_TAKEN
Not Taken Branch Delay Emulation	SIGTRAP	BRK_BD_NOTTAKEN
User Single Step (used by debuggers)	SIGTRAP	BRK_SSTEPBP
Overflow Check	SIGTRAP	BRK_OVERFLOW
Divide by Zero Check	SIGTRAP	BRK_DIVZERO
Range Error Check	SIGTRAP	BRK_RANGE

When a signal handler is reached, the program counter in the signal context structure (*sc_pc*) points at the instruction that caused the exception as modified by the *branch delay* bit in the *cause* register. The *cause* register at the time of the exception is also saved in the sigcontext structure (*sc_cause*). If the instruction that caused the exception is at a valid user address it can be retrieved with the following code sequence:

```
    if(scp->sc_cause & CAUSE_BD){
        branch_instruction = *(unsigned long *)(scp-
>sc_pc);
        exception_instruction = *(unsigned long *)(scp-
>sc_pc + 4);
    }
    else
        exception_instruction = *(unsigned long *)(scp-
>sc_pc);
```

Where CAUSE_BD is defined in *<mips/cpu.h>*.

The signal handler may fix the cause of the exception and re-execute the instruction, emulate the instruction and then step over it or perform some non-local goto such as a *longjump()* or an *exit()*.

If corrective action is performed in the signal handler and the instruction that caused the exception would then execute without a further exception, the signal handler simply returns and re-executes the instruction (even when the *branch delay* bit is set).

If execution is to continue after stepping over the instruction that caused the exception the program counter must be advanced. If the *branch delay* bit is set the program counter is set to the target of the branch else it is incremented by 4.

This can be done with the following code sequence:

```
if(scp->sc_cause & CAUSE_BD)
    emulate_branch(scp, branch_instruction);
else
    scp->sc_pc += 4;
```

Emulate_branch() modifies the program counter value in the sigcontext structure to the target of the branch instruction. See *emulate_branch(3)* for more details.

For SIGFPE's generated by floating-point instructions (*code* == 0) the *floating-point control and status* register at the time of the exception is also saved in the sigcontext structure (*sc_fpc_csr*). This register has the information on which exceptions have occurred. When a signal handler is entered the register contains the value at the time of the exception but with the *exceptions bits* cleared. On a return from the signal handler the exception bits in the floating-point control and status register are also cleared so that another SIGFPE does not occur (all other bits are restored from *sc_fpc_csr*).

For SIGSEGV and SIGBUS errors the faulting virtual address is saved in *sc_badvaddr* in the signal context structure.

The SIGTRAP's caused by **break** instructions noted in the above table and all other yet to be defined **break** instructions fill the *code* parameter with the first argument to the **break** instruction (bits 25-16 of the instruction).

Return Value

The previous action is returned on a successful call. Otherwise, –1 is returned and *errno* is set to indicate the error.

Diagnostics

The `signal` subroutine fails and action is not taken if one of the following occurs:

[EINVAL]	The *sig* is not a valid signal number.
[EINVAL]	An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

See Also

kill(1), kill(2), ptrace(2), sigblock(2), sigpause(2), sigsetmask(2), sigstack(2), sigvec(2), setjmp(3), tty(4)

Name

signal – simplified software signal facilities

Syntax

#include <signal.h>

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

Description

The `signal` subroutine is a simplified interface to the more general sigvec(2) facility.

A signal is generated by some abnormal event, initiated by a user at a terminal (quit, interrupt, stop), by a program error (bus error, etc.), by request of another program (kill), or when a process is stopped because it wishes to access its control terminal while in the background. For further information, see `tty`(4). Signals are optionally generated when a process resumes after being stopped, when the status of child process changes, or when input is ready at the control terminal. Most signals cause termination of the receiving process if no action is taken; some signals instead cause the process receiving them to be stopped, or are simply discarded if the process has not requested otherwise. Except for the SIGKILL and SIGSTOP signals, the `signal` call allows signals either to be ignored or to cause an interrupt to a specified location. The following is a list of all signals with names as in the include file < signal.h >:

SIGHUP	1	Hangup
SIGINT	2	Interrupt
SIGQUIT	3*	Quit
SIGILL	4*	Illegal instruction
SIGTRAP	5*	Trace trap
SIGIOT	6*	IOT instruction
SIGEMT	7*	EMT instruction
SIGFPE	8*	Floating point exception
SIGKILL	9	Kill (cannot be caught or ignored)
SIGBUS	10*	Bus error
SIGSEGV	11*	Segmentation violation
SIGSYS	12*	Bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	Alarm clock
SIGTERM	15	Software termination signal
SIGURG	16•	Urgent condition present on socket
SIGSTOP	17+	Stop (cannot be caught or ignored)
SIGTSTP	18+	Stop signal generated from keyboard
SIGCONT	19•	Continue after stop
SIGCHLD	20•	Child status has changed
SIGTTIN	21+	Background read attempted from control terminal
SIGTTOU	22+	Background write attempted to control terminal
SIGIO	23•	I/O is possible on a descriptor (see fcntl(2))
SIGXCPU	24	Cpu time limit exceeded (see setrlimit(2))
SIGXFSZ	25	File size limit exceeded (see setrlimit(2))

SIGVTALRM	26	Virtual time alarm (see setitimer(2))
SIGPROF	27	Profiling timer alarm (see setitimer(2))
SIGWINCH	28•	Window size change
SIGSHORT	29	System V record locking
SIGUSR1	30	User defined signal
SIGUSR2	31	User defined signal
SIGCLD		System V name for SIGCHLD
SIGABRT		X/OPEN name for SIGIOT

The starred signals in the list above cause a core image if not caught or ignored.

If *func* is SIG_DFL, the default action for signal *sig* is reinstated; this default is termination (with a core image for starred signals) except for signals marked with • or +. Signals marked with • are discarded if the action is SIG_DFL; signals marked with + cause the process to stop. If *func* is SIG_IGN the signal is subsequently ignored and pending instances of the signal are discarded. Otherwise, when the signal occurs further occurrences of the signal are automatically blocked and *func* is called.

A return from the function unblocks the handled signal and continues the process at the point it was interrupted. Unlike previous signal facilities, the handler *func* remains installed after a signal has been delivered.

If a caught signal occurs during certain system calls, causing the call to terminate prematurely, the call is automatically restarted. In particular this can occur during a *read* or `write`(2) on a slow device (such as a terminal; but not a file) and during a `wait`(2).

The value of `signal` is the previous (or initial) value of *func* for the particular signal.

After a `fork`(2) or `vfork`(2) the child inherits all signals. The `execve`(2) system call resets all caught signals to the default action; ignored signals remain ignored.

Return Value

The previous action is returned on a successful call. Otherwise, –1 is returned and *errno* is set to indicate the error.

Diagnostics

The `signal` subroutine will fail and no action will take place if one of the following occur:

[EINVAL] The *sig* is not a valid signal number.

[EINVAL] An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

Notes (VAX-11)

The handler routine can be declared:

```
handler(sig, code, scp)
```

Here *sig* is the signal number, into which the hardware faults and traps are mapped as defined below. Code is a parameter which is either a constant as given below or, for compatibility mode faults, the code provided by the hardware. The *scp* is a pointer to the *struct sigcontext* used by the system to restore the process context from before the signal. Compatibility mode faults are distinguished from the other SIGILL traps by having PSL_CM set in the psl.

The following defines the mapping of hardware traps to signals and codes. All of these symbols are defined in < signal.h >:

Hardware condition	Signal	Code
Arithmetic traps:		
Integer overflow	SIGFPE	FPE_INTOVF_TRAP
Integer division by zero	SIGFPE	FPE_INTDIV_TRAP
Floating overflow trap	SIGFPE	FPE_FLTOVF_TRAP
Floating/decimal division by zero	SIGFPE	FPE_FLTDIV_TRAP
Floating underflow trap	SIGFPE	FPE_FLTUND_TRAP
Decimal overflow trap	SIGFPE	FPE_DECOVF_TRAP
Subscript-range	SIGFPE	FPE_SUBRNG_TRAP
Floating overflow fault	SIGFPE	FPE_FLTOVF_FAULT
Floating divide by zero fault	SIGFPE	FPE_FLTDIV_FAULT
Floating underflow fault	SIGFPE	FPE_FLTUND_FAULT
Length access control	SIGSEGV	faulting virtual addr
Protection violation	SIGBUS	faulting virtual addr
Reserved instruction	SIGILL	ILL_PRIVIN_FAULT
Customer-reserved instr.	SIGEMT	
Reserved operand	SIGILL	ILL_RESOP_FAULT
Reserved addressing	SIGILL	ILL_RESAD_FAULT
Trace pending	SIGTRAP	
Bpt instruction	SIGTRAP	
Compatibility-mode	SIGILL	hardware supplied code
Chme	SIGSEGV	
Chms	SIGSEGV	
Chmu	SIGSEGV	

Environment

When your program is compiled using the System V environment the handler function does NOT remain installed after the signal has been delivered.

Also, when a signal which is to be caught occurs during a read(), write(), or ioctl() to a slow device (like a terminal, but not a file); or during a pause(); or wait() that does not return immediately, the signal handler function will be executed, and then the interrupted system call may return a -1 to the calling process with errno set to EINTR.

See Also

kill(1), kill(2), ptrace(2), sigblock(2), sigpause(2), sigsetmask(2), sigstack(2), sigvec(2), setjmp(3), tty(4)

Name

sigprocmask – examine and change blocked signals (POSIX)

Syntax

#include <signal.h>

int sigprocmask(*how, set, oset***)**
int *how***;**
sigset_t **set, *oset***;**

Description

The `sigprocmask` system call is used to examine and/or change the calling process's signal mask. If the value of the argument *set* is not NULL, it points to a set of signals that will be used to change the currently blocked set.

The value of the argument *how* indicates the manner in which the set is changed as defined by the following values, defined in <signal.h>:

SIG_BLOCK
: The resulting signal set is the union of the current set and the signal set pointed to by the argument *set.*

SIG_UNBLOCK
: The resulting signal set is the intersection of the current set and the complement of the signal set pointed to by the argument *set.*

SIG_SETMASK
: The resulting signal set is the signal set pointed to by the argument *set.*

If the argument *oset* is not NULL, the previous mask is stored in the space pointed to by *oset.* If the value of the argument *set* is NULL, the process's signal mask is unchanged; thus, the `sigprocmask`(3) function can be used to enquire about currently blocked signals.

The signal masks used as arguments to this function are manipulated using the `sigsetops`(3) functions.

As a system restriction, SIGKILL and SIGSTOP cannot be blocked.

Return Value

A 0 return value indicates a successful call. A –1 return value indicates an error and *errno* is set to indicated the reason.

Diagnostics

The `sigprocmask` function fails and the signal mask remains unchanged if the follow occurs:

[EINVAL] The value of the *how* argument is not equal to one of the defined values.

See Also

kill(2), sigsetmask(2), sigvec(2), sigblock(2), sigsetops(3)

Name

sigsetjmp, siglongjmp – nonlocal goto

Syntax

#include <setjmp.h>

sigsetjmp(*env, savemask***)**
sigjmp_buf *env***;**

siglongjmp(*env, val***)**
sigjmp_buf *env***;**

Description

These routines deal with errors and interrupts encountered in a low-level subroutine of a program.

The `sigsetjmp` subroutine saves its stack environment in *env* for later use by `siglongjmp`. It returns a value of 0. If the value of the *savemask* argument is not zero, the `sigsetjmp` subroutine also saves the process′ current signal mask as part of the calling environment.

The `siglongjmp` subroutine restores the environment saved by the last call of `sigsetjmp` with the supplied *env* buffer. If the *env* argument was initialized by a call to the `sigsetjmp` subroutine with a nonzero *savemask* argument, the `siglongjmp` subroutine restores the saved signal mask. It then returns in such a way that execution continues as if the call of `sigsetjmp` had just returned the value *val* to the subroutine that invoked `sigsetjmp`, which must not itself have returned in the interim. However, `siglongjmp` cannot cause `sigsetjmp` to return the value 0. If `siglongjmp` is invoked with a *val* of 0, `sigsetjmp` returns a value of 1. All accessible data have values as of the time `siglongjmp` was called.

Restrictions

The `sigsetjmp` subroutine does not save the current notion of whether the process is executing on the signal stack. When you invoke the `siglongjmp` subroutine, the signal stack is left in an incorrect state.

See Also

sigstack(2), sigvec(2), signal(3), sigprocmask(3)

Name

sigemptyset, sigfillset, sigaddset, sigdelset, sigismember – manipulate signal sets (POSIX)

Syntax

#include <signal.h>

int sigemptyset(*set***)**
sigset_t ***set*;**

int sigfillset (*set***)**
sigset_t ***set*;**

int sigaddset(*set,sig***)**
sigset_t ***set*;**
int *sig***;**

int sigdelset(*set,sig***)**
sigset_t ***set*;**
int *sig***;**

int sigismember(*set,sig***)**
sigset_t ***set*;**
int *sig***;**

Description

The `sigsetops`(3) functions manipulate signal sets used by the other POSIX signal functions `sigaction`(3,) `sigprocmask`(3,) `sigsuspend`(3).

The `sigemptyset`(3) function initializes the signal set pointed to by the argument *set* so that all signals are excluded.

The `sigfillset`(3) function initializes the signal set pointed to by the argument *set* so that all signals are included.

The `sigaddset`(3) and `sigdelset`(3) functions respectively add and delete the individual signal specified by the value of the argument *sig* from the signal set pointed to by the argument *set*.

The `sigismember`(3) function tests whether the signal specified by the value of the argument *sig* is a member of the set pointed to by the argument *set*.

Return Value

Upon successful completion, the `sigismember`(3) function returns a value of 1 if the specified signal is a member of the set. If it is not a member of the set, a value of 0 is returned.

If the `sigaddset`(3,) `sigdelset`(3,) or `sigismember`(3) functions fail a –1 value is returned and *errno* is set to indicate the reason.

Diagnostics

The `sigsetops`(3) function will fail and the signal mask will remain unchanged if one of the following occur:

[EINVAL] The value of the *sig* argument is not a valid signal number

See Also

sigprocmask(3), sigaction(3), sigsuspend(3), sigpending(2)

Name

sigsuspend – wait for signal (POSIX)

Syntax

sigsuspend(sigmask)
sigset_t *sigmask;

Description

The `sigsuspend` system call is the POSIX equivalent of the `sigpause`(2) system call. The behavior of this call is as described on the `sigpause`(2) reference page except, the signal mask is manipulated using the `sigsetops`(3) functions.

See Also

sigpause(2), sigaction(3), sigvec(2)

Name

sleep – suspend execution for interval

Syntax

unsigned
sleep(seconds)
unsigned seconds;

Description

The current process is suspended from execution for the number of seconds specified by the argument. The actual suspension time may be up to 1 second less than that requested, because scheduled wakeups occur at fixed 1-second intervals, and an arbitrary amount longer because of other activity in the system.

The routine is implemented by setting an interval timer and pausing until it occurs. The previous state of this timer is saved and restored. If the sleep time exceeds the time to the expiration of the previous timer, the process sleeps only until the signal would have occurred, and the signal is sent 1 second later.

Return Value

The value returned by `sleep` is the unslept amount(the requested time minus the time actually slept). This return value may be non-zero in cases where the caller had an alarm set to go off earlier than the end of the requested time, or where `sleep` was interrupted due to a caught signal(see ENVIRONMENT below).

Environment

POSIX
SYSTEM_FIVE
When your program is compiled in POSIX or System V mode, the `sleep` will be terminated by any caught signal. The `sleep` function will return following execution of the signal's catching routine.

See Also

setitimer(2), sigpause(2)

Name

statfs, – get file system statistics

Syntax

#include <sys/types.h>
#include <sys/param.h>
#include <sys/mount.h>

statfs(path, buffer)
char *path;
struct fs_data *buffer;

Description

The `statfs` library routine returns up-to-date information about a mounted file system. The *path* is the path name of any file within the mounted file system. The *buffer* is a pointer to an `fs_data` structure as defined in `getmnt`(2).

Return Value

Upon successful completion, a value of `1` is returned. If the file system is not mounted, `0` is returned. Otherwise, `-1` is returned and the global variable *errno* is set to indicate the error.

Diagnostics

The `statfs` library routine fails if one or more of the following are true:

[ENOTDIR] A component of the path prefix of *path* is not a directory.

[EINVAL] *path* contains a character with the high-order bit set.

[ENAMETOOLONG]
The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.

[ENOENT] The file referred to by *path* does not exist.

[EACCES] Search permission is denied for a component of the path prefix of *path*.

[ELOOP] Too many symbolic links were encountered in translating *path*.

[EFAULT] *buffer* or *path* points to an invalid address.

[EIO] An I/O error occurred while reading from the file system.

See Also

getmnt(2), getmountent(3)

Name

staux – routines that provide scalar interfaces to auxiliaries

Syntax

```
#include <syms.h>

long st_auxbtadd(bt)
long bt;

long st_auxbtsize(iaux,width)
long iaux;
long width;

long st_auxisymadd (isym)
long isym;

long st_auxrndxadd (rfd,index)
long rfd;
long index;

long st_auxrndxadd (idn)
long idn;

void st_addtq (iaux,tq)
long iaux;
long tq;

long st_tqhigh_aux(iaux)
long iaux;

void st_shifttq (iaux, tq)
int iaux;
int tq;

long st_iaux_copyty (ifd, psym)
long ifd;
pSYMR psym;

void st_changeaux (iaux, aux)
long iaux;
AUXU aux;

void st_changeauxrndx (iaux, rfd, index)
long iaux;
long rfd;
long index;
```

Description

Auxiliary entries are unions with a fixed length of four bytes per entry. Much information is packed within the auxiliaries. Rather than have the compiler front-ends handle each type of auxiliary entry directly, the following set of routines provide a high-level scalar interface to the auxiliaries:

st_auxbtadd	Adds a type information record (TIR) to the auxiliaries. It sets the basic type (bt) to the argument and all other fields to zero. The index to this auxiliary entry is returned.

st_auxbtsize Sets the bit in the TIR, pointed to by the *iaux* argument. This argument says the basic type is a bit field and adds an auxiliary with its width in bits.

st_auxisymadd Adds an index into the symbol table (or any other scalar) to the auxiliaries. It sets the value to the argument that will occupy all four bytes. The index to this auxiliary entry is returned.

st_auxrndxadd Adds a relative index, RNDXR, to the auxiliaries. It sets the rfd and index to their respective arguments. The index to this auxiliary entry is returned.

st_auxrndxadd_idn Works the same as *st_auxrndxadd* except that RNDXR is referenced by an index into the dense number table.

st_iaux_copyty Copies the type from the specified file (ifd) for the specified symbol into the auxiliary table for the current file. It returns the index to the new aux.

st_shifttq Shifts in the specified type qualifier, tq (see sym.h), into the auxiliary entry TIR, which is specified by the 'iaux' index into the current file. The current type qualifiers shift up one tq so that the first tq (tq0) is free for the new entry.

st_addtq Adds a type qualifier in the highest or most significant non-tqNil type qualifier.

st_tqhigh_iaux Returns the most significant type qualifier given an index into the files aux table.

st_changeaux Changes the iauxth aux in the current file's auxiliary table to aux.

st_changeauxrndx Converts the relative index (RNDXR) auxiliary, which is specified by iaux, to the specified arguments.

See Also

stfd(3)

Name

stcu – routines that provide a compilation unit symbol table interface

Syntax

#include <syms.h>

pCHDRR st_cuinit ()

void st_setchdr (pchdr)
pCHDRR pchdr;

pCHDRR st_currentpchdr()

void st_free()

long st_extadd (iss, value, st, sc, index)
long iss;
long value;
long st;
long sc;
long index;

pEXTR st_pext_iext (iext)
long iext;

pEXTR st_pext_rndx (rndx)
RNDXR rndx;

long st_iextmax()

long st_extstradd (str)
char *str;

char *st_str_extiss (iss)
long iss;

long st_idn_index_fext (index, fext)
long index;
long fext;

long st_idn_rndx (rndx)
RNDXR rndx;

pRNDXR st_pdn_idn (idn)
long idn;
RNDXR st_rndx_idn (idn)
long idn;

void st_setidn (idndest, idnsrc)
long idndest;
long idnsrc;

Description

The **stcu** routines provide an interface to objects that occur once per object, rather than once per file descriptor (for example, external symbols, strings, and dense numbers). The routines provide access to the current *chdr* (compile time hdr), which represents the symbol table in running processes with pointers to symbol table sections rather than indices and offsets used in the disk file representation.

A new symbol table can be created with *st_cuinit*. This routine creates and initializes a CHDRR (see *cmplrs/stsupport.h)*. The CHDRR is the current chdr and is used in all later calls.

NOTE

A chdr can also be created with the read routines (see `stio`(3)). The *st_cuinit* routine returns a pointer to the new CHDRR record.

st_currentchdr Returns a pointer the current chdr.

st_setchdr Sets the current chdr to the *pchdr* argument and sets the per file structures to reflect a change in symbol tables.

st_free Frees all constituent structures associated with the current chdr.

st_extadd Lets you add to the externals table. It returns the index to the new external for future reference and use. The *ifd* field for the external is filled in by the current file (see `stfd`(3)). For more details on the parameters, see *sym.h.*

st_pext_iext and st_pext_rndx
Returns pointers to the external, given a index referencing them. The latter routine requires a relative index where the *index* field should be the index in external symbols and the *rfd* field should be the constant ST_EXTIFD. **NOTE**: The externals contain the same structure as symbols (see the *SYMR* and *EXTR* definitions).

st_iextmax Returns the current number of entries in the external symbol table.

The *iss* field in external symbols (the index into string space) must point into external string space.

st_extstradd Adds a null-terminated string to the external string space and returns its index.

st_str_extiss Converts that index into a pointer to the external string.

The dense number table provides a convenience to the code optimizer, generator, and assembler. This table lets them reference symbols from different files and externals with unique densely packed numbers.

st_idn_index_fext Returns a new dense number table index, given an index into the symbol table of the current file (or if *fext* is set, the externals table).

st_idn_rndx Returns a new dense number, but expects a RNDXR (see *sym.h* to specify both the file index and the symbol index rather than implying the file index from the current file. The RNDXR contains two fields: an index into the externals table and a file index *rsyms* can point into the symbol table, as well). The file index is ST_EXTIFD (see *stsupport.h)* for externals.

st_rndx_idn Returns a RNDX, given an index into the dense number table.

st_pdn_idn Returns a pointer to the RNDXR index by the idn argument.

See Also

stfe(3), stfd(3)

Name

stfd – routines that provide access to per file descriptor section of the symbol table

Syntax

```
#include <syms.h>

long st_currentifd ()

long st_ifdmax ()

void st_setfd (ifd)
long ifd;

long st_fdadd (filename)
char *filename;

long st_symadd (iss, value, st, sc, freloc, index)
long iss;
long value;
long st;
long sc;
long freloc;
long index;

long st_auxadd (aux)
AUXU aux;

long st_stradd (cp)
char *cp;

long st_lineadd (line)
long line;

long st_pdadd (isym)
long isym;

long st_ifd_pcfd (pcfd1)
pCFDR pcfd1;

pCFDR st_pcfd_ifd (ifd)
long ifd;

pSYMR st_psym_ifd_isym (ifd, isym)
long ifd;
long isym;

pAUXU st_paux_ifd_iaux (ifd, iaux)
long ifd;
long iaux;

pAUXU st_paux_iaux (iaux)
long iaux;

char *st_str_iss (iss)
long iss;
```

char *st_str_ifd_iss (ifd, iss)
long ifd;
long iss;

pPDR st_ppd_ifd_isym (ifd, isym)
long ifd;
long isym;

char * st_malloc (ptr,psize,itemsize,baseitems)
char *ptr;
long *size;
long itemsize;
long baseitems;

Description

The **stfd** routines provide an interface to objects handled on a per file descriptor (or fd) level. For example: local symbols, auxiliaries, local strings, line numbers, optimization entries, procedure descriptor entries, and the file descriptors. These routines constitute a group because they deal with objects corresponding to fields in the *FDR* structure.

A fd can be activated by reading an existing one into memory or by creating a new one. The compilation unit routines *st_readbinary* and *st_readst* read file descriptors and their constituent parts into memory from a symbol table on disk.

The *st_fdadd* adds a file descriptor to the list of file descriptors. The *lang* field is initialized from a user specified global *st_lang* that should be set to a constant designated for the language in *symconst.h.* The *fMerge* field is initialized from the user specified global st_merge that specifies whether the file is to start with the attribute of being able to be merged with identical files at load time. The *fBigendian* field is initialized by the `gethostsex`(3) routine, which determines the permanent byte ordering for the auxiliary and line number entries for this file.

The *st_fdadd* adds the null string to the new files string table that is accessible by the constant issNull (0. It also adds the filename to the string table and sets the *rss* field. Finally, the current file is set to the newly added file so that later calls operate on that file.

All routines for fd-level objects handle only the current file unless a file index is specified. The current file can also be set with *st_setfd.*

Programs can find the current file by calling *st_currentifd,* which returns the current index. Programs can find the number of files by calling *st_ifdmax.* The fd routines only require working with indices to do most things. They allow more in-depth manipulation by allowing users to get the compile time file descriptor *(CFDR* see *stsupport.h)* that contains memory pointers to the per file tables (rather than indices or offsets used in disk files). Users can retrieve a pointer to the CFDR by calling *st_pcfd_ifd* with the index to the desired file. The inverse mapping *st_ifd_pcfd* exists, as well.

Each of fd's constituent parts has an add routine: *st_symadd, st_stradd, st_lineadd, st_pdadd,* and *st_auxadd.* The parameters of the add routines correspond to the fields of the added object (see *sym.h*). The *pdadd* routine lets users fill in the isym field only. Further information can be added by directly accessing the procedure descriptor entry.

The add routines return an index that can be used to retrieve a pointer to part of the desired object with one of the following routines: *st_psym_isym, st_str_iss,* and *st_paux_iaux.*

NOTE

These routines only return objects within the current file. The following routines allow for file specification: *st_psym_ifd_isym, st_aux_ifd_iaux,* and *st_str_ifd_iss.*

The *st_ppd_ifd_isym* allows access to procedures through the file index for the file where they occur and the isym field of the entry that points at the local symbol for that procedure.

The return index from st_symadd should be used to get a dense number (see stcu). That number should be the ucode block number for the object the symbol describes.

See Also

stcu(3), stfe(3), sym.h(5), stsupport.h(5)

Name

stfe – routines that provide a high-level interface to basic functions needed to access and add to the symbol table

Syntax

```
#include <syms.h>

long st_filebegin (filename)
char *filename;

long st_endallfiles ()

long st_fileend (idn)
long idn;

long st_blockbegin(iss, value, sc)
long iss;
long value;
long sc;

long st_textblock()

long st_blockend(size)
long size;

long st_procend(idn)
long idn

long st_procbegin (idn)
long idn;

char *st_str_idn (idn)
long idn;

char *st_sym_idn (idn, value, sc, st, index)
long idn;
long *value;
long *sc;
long *st;
long *index;

long st_abs_ifd_index (ifd, index)
long ifd;
long index;

long st_fglobal_idn (idn)
long idn;

pSYMR st_psym_idn_offset (idn, offset)
long idn;
long offset;

long st_pdadd_idn (idn)
long idn;
```

Description

The **stfe** routines provide a high-level interface to the symbol table based on common needs of the compiler front-ends.

st_filebegin — Takes a file name and calls *st_fdadd (see* `stfd`(3)). If it is a new file, a symbol is added to the symbol table that for that file or symbol, and the user supplied routine, *st_feinit,* is called. This allows special file parameters to be initialized. For example, the C front-end adds basic type auxiliaries to each file's aux table so that all variables of that type can refer to a single instance instead of making individual copies of them. The rountine *st_filebegin* returns a dense number that references the symbol added for this file. It tracks files as they appear in a CPP line directive with a stack. It detects (from the order of the CPP directives) that a file ends and calls *st_filend.* If a file is closed with a *st_fileend,* a new instance of the filename is created. For example, multiply included files.

st_fileend — Requires the dense number from the corresponding *st_filebegin call for the file* in question. It then generates an end symbol and patches the references so that the index field of the begin file points to that of one beyond the end file. The end file points to the begin file.

st_endallfiles — Is called at the end of execution to close off all files that have not been ended by previous calls to *st_filebegin.* CPP directives might not reflect the return to the original source file; therefore, this routine can possibly close many files.

st_blockbegin — Supports both language blocks (for example, C's left curly brace blocks), beginning of structures, and unions. If the storage class is scText, it is the former; if it is scInfo, it is one of the latter. The iss (index into string space) specifies the name of the structure/etc, if any.

If the storage class is scText, we must check the result of *st_blockbegin.* It returns a dense number for outer blocks and a zero for nested blocks. The non-zero block number should be used in the BGNB ucode. Users of languages without nested blocks that provide variable declarations can ignore the rest of this paragraph. Nested blocks are two-staged: one stage occurs when the language block is detected and the other stage occurs when the block has content. If the block has content (for example, local variables), the front-end must call *st_textblock* to get a non-zero dense number for the block's BGNB ucode. If the block does not have content and *st_textblock* is not called, the block's *st_blockbegin* and *st_blockend* do not produce block and end symbols.

If it is scInfo, *st_blockbegin* creates a begin block symbol in the symbol table and returns a dense number referencing it. The dense number is necessary to build the auxiliary required to reference the structure/etc. It goes in the aux after the TIR along with a file index. This dense number is also noted in a stack of blocks used by *st_blockend.*

The *st_blockbegin should not be called for* language blocks when the front-end is not producing debugging symbols.

The *st_blockend* requires that blocks occur in a nested fashion. It retrieves the dense number for the most recently started block and creates a corresponding end symbol. As in *fileend,* both the begin and end symbol index fields point at the other end's symbol. If the symbol ends a structure/etc., as determined by the storage class of the begin symbol, the size parameter is assigned to the begin symbol's value field. It is usually the size of the structure or max value of a enum. We only know it at this point. The dense number of the end symbol is returned so that the ucode ENDB can use it. If it is an ignored text block, the dense number is zero and no ENDB should be generated.

In general, defined external procedures or functions appear in the symbols table and the externals table. The external table definition must occur first through the use of a *st_extadd. After that definition, st_procbegin* can be called with a dense number referring to the external symbol for that procedure. It checks to be sure we have a defined procedure (by checking the storage class). It adds a procedure symbol to the symbol table. The external's index should point at its auxiliary data type information (or if debugging is off, indexNil). This index is copied into the regular symbol's index field or a copy of its type is generated (if the external is in a different file than the regular symbol). Next, we put the index to symbol in the external's index field. The external's dense number is used as a block number in ucodes referencing it and is used to add a procedure when in the *st_pdadd_idn.*

st_procend	Creates an end symbol and fixes the indices as in *blockend* and *fileend,* except that the end procedure reference is kept in the begin procedure's aux rather than in the index field (because the begin procedure has a type as well as an end reference). This must be called with the dense number of the procedure's external symbol as an argument and returns the dense number of the end symbol to be used in the END ucode.
st_str_idn	Returns the string associated with symbol or external referenced by the dense number argument. If the symbol was anonymous (for example, there is not a symbol), a (char *), -1 is returned.
st_sym_idn	Returns the same result as *st_str_idn, except that the rest of* by the *idn* are returned in the arguments.
st_fglobal_idn	Returns a 1 if the symbol associated with the specified idn is non-static; otherwise, a 0 is returned.
st_abs_ifd_index	Returns the absolute offset for a dense number. If the symbol is global, the global's index is returned. If the symbol occurred in a file, the sum of all symbols in files occurring before that file and the symbol's index within the file is returned.
st_pdadd_idn	Adds an entry to the procedure table for the *st_proc entry* generated by procbegin. This should be called when the front-end generates code for the procedure in question.

See Also

stcu(3), stfd(3), sym.h(5), stsupport.h(5)

Name

stime – set time

Syntax

int stime (tp)
long *tp;

Description

The `stime` system call sets the system's time and date. The *tp* argument points to the value of time as measured in seconds from 00:00:00 GMT January 1, 1970.

Return Value

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

Diagnostics

[EPERM] The effective user ID of the calling process is not the superuser.

See Also

gettimeofday(2), time(3)

Name

stio – routines that provide a binary read/write interface to the MIPS symbol table

Syntax

#include <syms.h>

long st_readbinary (filename, how)
char *filename;
char how;

long st_readst (fn, how, filebase, pchdr,flags)
long fn;
char how;
long filebase;
pCHDRR pchdr;
long flags;

void st_writebinary (filename, flags)
char *filename;
long flags;

void st_writest (fn, flags)
long fn;
long flags;

Description

The CHDRR structure (see **cmplrs/stsupport.h** and the `stcu`(3)). represents a symbol table in memory. A new CHDRR can be created by reading a symbol table in from disk. The *st_readbinary* and *st_readst* routines read a symbol table in from disk.

The routine *st_readbinary* takes the file name of the symbol table and assumes the symbol table header *(HDRR* in **sym.h** occurs at the beginning of the file. The *st_readst* assumes that its file number references a file positioned at the beginning of the symbol table header and that the *filebase* parameter specifies where the object or symbol table file is based (for example, non-zero for archives).

The second parameter to the read routines can be r for read only or a for appending to the symbol table. Existing local symbol, line, procedure, auxiliary, optimization, and local string tables cannot be appended. If they didn't exist on disk, they can be created. This restriction stems from the allocation algorithm for those symbol table sections when read in from disk and follows the standard pattern for building the symbol table.

The symbol table can be read incrementally. If *pchdr is zero, st_readst* assumes that a symbol table has not been read yet; therefore, it reads in the symbol table header and file descriptors. The *flags* argument is a bit mask that defines what other tables should be read. The *t_p** constants for each table, defined in stsupport.h, can be ORed. If *flags* equals -1, all tables are read. If *pchdr* is set, the tables specified by *flags* are added to the tables that have already been read. The *pchdr's value can be taken from st_current_pchdr. See* `stcu`(3.)

Line number entries are encoded on disk; the read routines expand them to longs.

If the version stamp is out of date, a warning message is issued to stderr. If the magic number in the HDRR is incorrect, *st_error* is called. All other errors cause the read routines to read non-zero; otherwise, a zero is returned.

The routines *st_writebinary* and *st_writest* are symmetric to the read routines, excluding the *how* and *pchdr* parameters. The *flags* parameter is a bit mask that defines what table should be written. The *st_p** constants for each table, defined in *stsupport.h,* can be ORed. If *flags* equals -1, all tables are written.

The write routines write sections of the table in the approved order, as specified in the link editor `ld(1)` specification.

Line numbers are compressed on disk.

The write routines start all sections of the symbol table on four-byte boundaries.

If the write routines encounter an error, *st_error* is called. After writing the symbol table, further access to the table by other routines is undefined.

See Also

stcu(3), stfs(3), stfw (3), sym.h(5), sterror(5) stsupport.h(5)

Name

strcoll – string collation comparison

Syntax

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

Description

The `strcoll` function returns an integer less than, equal to, or greater than zero depending on whether the string pointed to by *s1* is lexicographically less than, equal to, or greater than the string pointed to by *s2*.

The `strcoll` function performs the comparison by using the collating information defined in the program's locale, category `LC_COLLATE`.

In the C locale, characters collate as if they are unsigned. In all cases `strcoll` works as if `strxfrm` were called on *s1* and *s2*, and `strcmp` was called on the resulting strings.

International Environment

LC_COLLATE Contains the user requirements for language, territory, and codeset for the character collation format. `LC_COLLATE` affects the behavior of regular expressions and the string collation functions in `strcoll`. If `LC_COLLATE` is not defined in the current environment, `LANG` provides the necessary default.

LANG If this environment is set and valid, `strcoll` uses the international language database named in the definition to determine the character collation formatting rules. If `LC_COLLATE` is defined, its definition supercedes the definition of `LANG`.

See Also

string(3), setlocale(3), strxfrm(3), environ(5int)

Name

strftime – convert time and date to string

Syntax

#include <time.h>

int strftime (*s, maxsize, format, tm*)
char **s*;
size_t *maxsize*;
char **format*;
struct tm **tm*;

Description

The `strftime` function places characters in the array pointed to by *s*. No more than *maxsize* characters are placed into the array. The `format` string controls this process. This string consists of zero or more directives and ordinary characters. A directive consists of a `%` character followed by a character that determines the behavior of the directive. All ordinary characters are copied unchanged into the array, including the terminating null character.

Each directive is replaced by the appropriate characters as shown in the following table. The characters are determined by the program's locale category `LC_TIME` and the values contained in the structure pointed to by *tm*.

Directive	Replaced by
%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 date and time representation
%d	Day of month as a decimal number (01–31)
%D	Date (%m/%d/%y)
%h	Locale's abbreviated month name
%H	Hour as a decimal number (00–23)
%I	Hour as a decimal number (01–12)
%j	Day of year (001–366)
%m	Number of month (01–12)
%M	Minute number (00–59)
%n	Newline character
%p	Locale's equivalent to AM or PM
%r	Time in AM/PM notation
%S	Second number (00–59)
%t	Tab character
%T	Time (%H/%M/%S)
%U	Week number (00–53), Sunday as first day of week
%w	Weekday number (0[Sunday]–6)
%W	Week number (00–53), Monday as first day of week
%x	Locale's date representation
%X	Locale's time representation

%y	Year without century (00–99)
%Y	Year with century
%Z	Timezone name, no characters if no timezone
%%	%

If a directive is used that is not contained in the table, the results are undefined.

International Environment

LC_TIME Contains the user's requirements for language, territory, and codeset for the time format. `LC_TIME` affects the behavior of the time functions in `strftime`. If `LC_TIME` is not defined in the current environment, `LANG` provides the necessary default.

LANG If this environment is set and valid, `strftime` uses the international language database named in the definition to determine the time formatting rules. If `LC_TIME` is defined, its definition supercedes the definition of `LANG`.

Return Value

If the total number of resulting characters, including the terminal null character, is not more than *maxsize*, the `strftime` function returns the total of resultant characters placed into the array pointed to by *s*, not including the terminating null character. In all other cases zero is returned and the contents of the array are indeterminate.

As the `timezone` name is not contained in the *tm* structure the value returned by *%Z* is determined by the `timezone` function, see `ctime`.

See Also

ctime(3), setlocale(3)

Name

strcasecmp, strncasecmp, strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strstr, strtok, index, rindex – string operations

Syntax

#include <strings.h>

or

#include <string.h>

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

strncasecmp(*s1, s2, n***)**
char ****s1*, ****s2***;**

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

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

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

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

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

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

size_t strlen(*s***)**
char ****s*;**

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

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

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

size_t strspn(*s1, s2***)**
char ****s1*, ****s2***;**

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

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

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

char *rindex(*s, c*)
char **s, c*;
char *strstr(*s1, s2*)
char **s1*, **s2*;

Description

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

The `strcat` subroutine appends a copy of string *s2* to the end of string *s1*. The `strncat` subroutine copies at most *n* characters. Both return a pointer to the null-terminated result.

The `strcmp` subroutine compares its arguments and returns an integer greater than, equal to, or less than 0, according as *s1* is lexicographically greater than, equal to, or less than *s2*. The `strncmp` subroutine makes the same comparison but looks at at most *n* characters. The `strcasecmp` and `strncasecmp` subroutines are identical in function, but are case insensitive. The returned lexicographic difference reflects a conversion to lower-case.

The `strcpy` subroutine copies string *s2* to *s1*, stopping after the null character has been copied. The `strncpy` subroutine copies exactly *n* characters, truncating *s2* or adding null characters to *s1* if necessary. The result will not be null-terminated if the length of *s2* is *n* or more. Each function returns *s1*.

The `strlen` subroutine returns the number of characters in *s*, not including the terminating null character.

The `strstr` subroutine returns a pointer to the first occurrence of s2 (excluding the terminating null character) in s1, or a NULL pointer if s2 does not occur in s1. If `strlen`(s2) is zero, `strstr` returns s1.

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

The `strpbrk` subroutine 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*.

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

The `strtok` subroutine considers the string *s1* to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *s2*. The first call (with pointer *s1* specified) returns a pointer to the first character of the first token, and will have written a null character into *s1* immediately following the returned token. The function keeps track of its position in the string between separate calls, so that subsequent calls (which must be made with the first argument a NULL pointer) will work through the string *s1* immediately following that token. In this way, subsequent calls will work through the string *s1* until no tokens remain. The separator string *s2* may be different from call to call. When no token remains in *s1*, a NULL pointer is returned.

The `index` (`rindex`) subroutine returns a pointer to the first (last) occurrence of character *c* in string *s*, or zero if *c* does not occur in the string.

NOTE

The <string.h> header file is provided for compatibility with System V; both <string.h> and <strings.h> refer to the same file.
The `strcmp` and `strncmp` subroutines do unsigned character comparisons.

Name

strxfrm – string transformation

Syntax

size_t strxfrm (*to, from, maxsize*)
char **to*;
char **from*;
size_t *maxsize*;

Description

The `strxfrm` function transforms the string pointed to by *from* and places the resulting string into the array pointed to by *to*. The transformation is such that two transformed strings can be ordered by the `strcmp` function as appropriate to the program's locale category `LC_COLLATE`.

The length of the resulting string may be much longer than the original. No more than `maxsize` characters are placed into the resulting string including the terminator. If the transformed string does not exceed `maxsize` characters, the number of characters (less the terminator) is returned. Otherwise the number of characters (less the terminator) in the transformed string is returned and the contents of the array are undefined.

International Environment

LC_COLLATE Contains the user requirements for language, territory, and codeset for the character collation format. `LC_COLLATE` affects the behavior of regular expressions and the string collation functions in `strxfrm`. If `LC_COLLATE` is not defined in the current environment, `LANG` provides the necessary default.

LANG If this environment is set and valid, `strxfrm` uses the international language database named in the definition to determine the character collation formatting rules. If `LC_COLLATE` is defined, its definition supercedes the definition of `LANG`.

See Also

string(3), setlocale(3), strcoll(3), environ(5int)

Name

stty, gtty – set and get terminal state

Syntax

#include <sgtty.h>

stty(fd, buf)
int fd;
struct sgttyb *buf;

gtty(fd, buf)
int fd;
struct sgttyb *buf;

Description

This interface has been superseded by `ioctl`(2).

The `stty` subroutine sets the state of the terminal associated with *fd*. The `gtty` subroutine retrieves the state of the terminal associated with *fd*. To set the state of a terminal the call must have write permission.

The `stty` call is actually ''ioctl(fd, TIOCSETP, buf)'', while the `gtty` call is ''ioctl(fd, TIOCGETP, buf)''. See `ioctl`(2) and `tty`(4) for an explanation.

Return Value

If the call is successful 0 is returned, otherwise –1 is returned and the global variable *errno* contains the reason for the failure.

See Also

ioctl(2), tty(4)

Name

swab – swap bytes

Syntax

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

Description

The `swab` subroutine copies *nbytes* bytes pointed to by *from* to the position pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between machines. The *nbytes* should be even.

Name

swap_word, swap_half, swap_filehdr, swap_aouthdr, swap_scnhdr, swap_hdr, swap_fd, swap_fi, swap_sym, swap_ext, swap_pd, swap_dn, swap_opt, swap_aux, swap_reloc, swap_ranlib – swap the sex of the specified structure

Syntax

#include <sex.h>
#include <filehdr.h>
#include <aouthdr.h>
#include <scnhdr.h>
#include <sym.h>
#include <symconst.h>
#include <cmplrs/stsupport.h>
#include <reloc.h>
#include <ar.h>

long swap_word(*word* **)**
long word;

short swap_half(*half* **)**
short half;

void swap_filehdr(*pfilehdr, destsex* **)**
FILHDR *pfilehdr;
long destsex;

void swap_aouthdr(*paouthdr, destsex* **)**
AOUTHDR *paouthdr;
long destsex;

void swap_scnhdr(*pscnhdr, destsex* **)**
SCNHDR *pscnhdr;
long destsex;

void swap_hdr(*phdr, destsex* **)**
pHDRR phdr;
long destsex;

void swap_fd(*pfd, count, destsex* **)**
pFDR pfd;
long count;
long destsex;

void swap_fi(*pfi, count, destsex* **)**
pFIT pfi;
long count;
long destsex;

void swap_sym(*psym, count, destsex* **)**
pSYMR psym;
long count;
long destsex;

void swap_ext(*pext, count, destsex*)
pEXTR pext;
long count;
long destsex;

void swap_pd(*ppd, count, destsex*)
pPDR ppd;
long count;
long destsex;

void swap_dn(*pdn, count, destsex*)
pRNDXR pdn;
long count;
long destsex;

void swap_opt(*popt, count, destsex*)
pOPTR popt;
long count;
long destsex;

void swap_aux(*paux, type, destsex*)
pAUXU paux;
long type;
long destsex;

void swap_reloc(*preloc, count, destsex*)
struct reloc *preloc;
long count;
long destsex;

void swap_ranlib(*pranlib, count, destsex*)
struct ranlib *pranlib;
long count;
long destsex;

Description

All `swapsex` routines that swap headers take a pointer to a header structure to change the byte's sex. The *destsex* argument lets the swapsex routines decide whether to swap bitfields before or after swapping the words in which they occur. If *destsex* equals the hostsex of the machine you are running on, the flip happens before the swap; otherwise, the flip happens after the swap. Although not all routines swap structures containing bitfields, the destsex is required.

The `swap_aux` routine takes a pointer to an *aux* entry and a *type*, which is a ST_AUX_* constant in cmplrs/stsupport.h. The constant specifies the type of the aux entry to change the sex of. All other `swapsex` routines are passed a pointer to an array of structures and a *count* of structures to have the byte sex changed. The routines `swap_word` and `swap_half` are macros declared in *sex.h*. Only the include files that describe the structures being swapped have to be included.

See Also

gethostsex(3)

sysconf (3)

Name

sysconf – get configurable system variables (POSIX)

Syntax

#include <unistd.h>

long sysconf(*name*)
int *name*;

Description

The `sysconf` function provides a method for the application to determine the current value of a configurable system limit or option.

The *name* argument represents the system variable to be queried. The following table lists the system variables which may be queried and the corresponding value for the *name* argument. The values for the *name* argument are defined in the <unistd.h> header file.

Variable	name Value
ARG_MAX	_SC_ARG_MAX
CHILD_MAX	_SC_CHILD_MAX
CLK_TCK	_SC_CLK_TCK
NGROUPS_MAX	_SC_NGROUPS_MAX
OPEN_MAX	_SC_OPEN_MAX
PASS_MAX	_SC_PASS_MAX
_POSIX_JOB_CONTROL	_SC_JOB_CONTROL
_POSIX_SAVED_IDS	_SC_SAVED_IDS
_POSIX_VERSION	_SC_VERSION
_XOPEN_VERSION	_SC_XOPEN_VERSION

Return Value

Upon successful completion, the `sysconf` function returns the current variable value on the system.

If *name* is an invalid value, `sysconf` returns –1 and *errno* is set to indicate the reason. If the variable corresponding to *name* is not defined on the system, `sysconf` returns –1 without changing the value of *errno.*

Diagnostics

The `sysconf` function fails if the following occurs:

[EINVAL] The value of the *name* argument is invalid.

Name

syslog, openlog, closelog – control system log

Syntax

#include <syslog.h>

openlog(ident, logstat)
char *ident;

syslog(priority, message, parameters ...)
char *message;

closelog()

Description

The `syslog` subroutine arranges to write the message onto the system log maintained by `syslog`(8). The message is tagged with priority and it looks like a `printf`(3s) string except that **%m** is replaced by the current error message (collected from `errno`). A trailing new line is added if needed. This message is read by `syslog`(8) and output to the system console or files as appropriate. The maximum number of parameters is 5.

If special processing is needed, `openlog` can be called to initialize the log file. Parameters are *ident* which is prepended to every message, and *logstat* which is a bit field indicating special status; current values are:

LOG_PID
log the process id with each message; useful for identifying daemons.

The `openlog` returns zero on success. If it cannot open the file `/dev/log`, it writes on `/dev/console` instead and returns –1.

The `closelog` can be used to close the log file.

Examples

```
syslog(LOG_SALERT, "who: internal error 23");

openlog("serverftp", LOG_PID);
syslog(LOG_INFO, "Connection from host %d", CallingHost);
```

See Also

syslog(8)

system (3)

Name

system – issue a shell command

Syntax

system(*string*)
char **string*;

Description

If the *string* argument is the NULL pointer (0) the `system` function tests the accessibility of the command interpreter `sh`(1). The function will return zero for failure to find the command interpretter, and positive if successful.

If the *string* argument is non-NULL the `system` routine causes the *string* to be given to `sh`(1) as input as if the string had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status in the form that `wait`(2) returns.

Diagnostics

Exit status 127 indicates the shell couldn't be executed.

See Also

execve(2), wait(2), popen(3)

Name

time, ftime – get date and time

Syntax

#include <time.h>
time_t time((*long* *)*0*)

time_t time(*tloc***)**
time_t ****tloc*;**

#include <sys/timeb.h>

ftime(*tp***)**
struct timeb ****tp*;**

Description

The `time` subroutine returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds.

If *tloc* is nonnull, the return value is also stored in the place to which *tloc* points.

The `ftime` entry fills in a structure pointed to by its argument, as defined by <sys/timeb.h>:

```
struct timeb
{
        time_t   time;
        unsigned short millitm;
        short    timezone;
        short    dstflag;
};
```

The structure contains the time since the epoch in seconds, up to 1000 milliseconds of more-precise interval, the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

See Also

date(1), gettimeofday(2), settimeofday(2), ctime(3)

Name

times – get process times

Syntax

#include <sys/times.h>

clock_t
times(*buffer*)
struct tms *_buffer_;

Description

The `times` subroutine returns time-accounting information for the current process and for the terminated child processes of the current process. All times are in 1/HZ seconds, where HZ is equivalent to 60.

The following structure is returned by `times`:

```
struct tms {
        clock_t  tms_utime;      /* user time */
        clock_t  tms_stime;      /* system time */
        clock_t  tms_cutime;     /* user time, children */
        clock_t  tms_cstime;     /* system time, children */
};
```

The children times are the sum of the children's process times and their children's times.

Return Value

If successful, the function `times` returns the elapsed time since 00:00:00 GMT, January 1, 1970 in units of 1/60's of a second. When the function `times` fails, it returns –1

See Also

time(1), getrusage(2), wait3(2), time(3)

Name

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

Syntax

#include <search.h>

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

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

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

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

Description

The `tsearch` subroutine is a binary tree search routine generalized from Knuth (6.2.2) Algorithm T. It returns a pointer into a tree indicating where a datum may be found. If the datum does not occur, it is added at an appropriate point in the tree. The *key* points to the datum to be sought in the tree. The *rootp* points to a variable that points to the root of the tree. A NULL pointer value for the variable denotes an empty tree; in this case, the variable will be set to point to the datum at the root of the new tree. The *compar* is the name of the comparison function. It is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero according as the first argument is to be considered less than, equal to, or greater than the second.

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

The `tdelete` subroutine deletes a node from a binary search tree. It is generalized from Knuth (6.2.2) algorithm D. The arguments are the same as for `tsearch.` The variable pointed to by *rootp* will be changed if the deleted node was the root of the tree. The `tdelete` subroutine returns a pointer to the parent of the deleted node, or a NULL pointer if the node is not found.

The `twalk` subroutine traverses a binary search tree. The *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.) The *action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited. The second argument is a value from an enumeration data type *typedef enum { preorder, postorder, endorder, leaf }* VISIT; (defined in the <search.h>

header file), depending on whether this is the first, second or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level zero.

Notes

The pointers to the key and the root of the tree should be of type pointer-to-element, and cast to type pointer-to-character.

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.

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

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.

Restrictions

Results are unpredictable if the calling function alters the pointer to the root.

Diagnostics

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

See Also

bsearch(3), hsearch(3), lsearch(3)

Name

ttyname, isatty, ttyslot – find terminal name

Syntax

char *ttyname(filedes)

isatty(filedes)

ttyslot()

Description

The `ttyname` subroutine returns a pointer to the null-terminated path name of the terminal device associated with file descriptor *filedes* (this is a system file descriptor and has nothing to do with the standard I/O FILE typedef).

The `isatty` subroutine returns 1 if *filedes* is associated with a terminal device, 0 otherwise.

The `ttyslot` subroutine returns the number of the entry in the `ttys`(5) file for the control terminal of the current process.

Restrictions

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

Diagnostics

The `ttyname` subroutine returns a null pointer (0) if *filedes* does not describe a terminal device in directory `/dev`.

The `ttyslot` subroutine returns 0 if `/etc/ttys` is inaccessible or if it cannot determine the control terminal.

Files

/dev/*
/etc/ttys

See Also

ioctl(2), ttys(5)

Name

ulimit – get and set user limits

Syntax

long ulimit (*cmd, newlimit*)
int *cmd*;
long *newlimit*;

Description

This function provides control over process limits. An explanation of the *cmd* values follow.

Value	Explanation
1	Get the process's file size limit. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.
2	Set the process's file size limit to the value of *newlimit*. Any process can decrease this limit, but only a process with an effective user ID of superuser can increase the limit. The `ulimit` system call fails and the limit remains unchanged, if a process with an effective user ID other than superuser attempts to increase its file size limit.
3	Get the maximum possible break value. For further information, see `brk`(2).

Return Value

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

Diagnostics

[EINVAL] Bad value for *cmd*.

[EPERM] The effective user ID of the calling process is not superuser.

See Also

brk(2), write(2)

Name

utime – set file times

Syntax

#include <sys/types.h>
int utime (*path, times*)
char **path*;
struct utimbuf **times*;

Description

The *path* points to a pathname naming a file. The `utime` function sets the access and modification times of the named file.

If *times* is NULL, the access and modification times of the file are set to the current time. A process must be the owner of the file or have write permission to use `utime` in this manner.

If *times* is not NULL, *times* is interpreted as a pointer to a *utimbuf* structure and the access and modification times are set to the values contained in the designated structure. Only the owner of the file or the super-user can use `utime` this way.

The function `utime` causes the time of the last file status change(st_ctime) to be updated with the current time.

The times in the following structure are measured in seconds since 00:00:00 GMT, January 1, 1970.

```
struct  utimbuf  {
        time_t  actime;     /* access time */
        time_t  modtime;    /* modification time */
};
```

Return Value

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

Diagnostics

The `utime` function fails, if any of the following is true:

[EACCES]	Search permission is denied by a component of the *path* prefix.
[EACCES]	The effective user ID is not super-user, not the owner of the file, *times* is NULL, and write access is denied.
[EFAULT]	The *times* is not NULL and points outside the process's allocated address space.
[EFAULT]	The *path* points outside the process's allocated address space.
[ENOENT]	The named file does not exist or *path* points to an empty string and the environment defined is POSIX or SYSTEM_FIVE.
[ENOTDIR]	A component of the *path* prefix is not a directory.

[EPERM] The effective user ID is not a super-user, not the owner of the file, and *times* is not NULL.

[EROFS] The file system containing the file is mounted read-only.

[ETIMEDOUT] A connect request or remote file operation failed, because the connected party did not respond after a period of time determined by the communications protocol.

See Also

stat(2)

Name

valloc – aligned memory allocator

Syntax

#include <stdlib.h>

void *valloc(*size***)**
size_t *size***;**

Description

The `valloc` subroutine allocates *size* bytes aligned on a page boundary. It is implemented by calling `malloc`(3) with a slightly larger request, saving the true beginning of the block allocated, and returning a properly aligned pointer.

Diagnostics

The `valloc` subroutine returns a null pointer (0) if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. The `valloc` subroutine will fail and no additional memory will be allocated if one of the following is true:

[ENOMEM] The limit, as set by `setrlimit`(2), is exceeded.

[ENOMEM] The maximum possible size of a data segment (compiled into the system) is exceeded.

[ENOMEM] Insufficient space exists in the swap area to support the expansion.

Name

varargs – variable argument list

Syntax

#include <varargs.h>

function(**va_alist**)
va_dcl
va_list *pvar*;
va_start(*pvar*);
f = **va_arg**(*pvar*, *type*);
va_end(*pvar*);

Description

This set of macros provides a means of writing portable procedures that accept variable argument lists. Routines having variable argument lists, such as `printf`(3s), that do not use varargs are inherently nonportable, since different machines use different argument passing conventions.

va_alist is used in a function header to declare a variable argument list.

va_dcl is a declaration for **va_alist**. Note that there is no semicolon after **va_dcl.**

va_list is a type which can be used for the variable *pvar*, which is used to traverse the list. One such variable must always be declared.

va_start(pvar) is called to initialize *pvar* to the beginning of the list.

va_arg(*pvar*, *type*) will return the next argument in the list pointed to by *pvar*. The *type* is the type the argument is expected to be. Different types can be mixed, but it is up to the routine to know what type of argument is expected, since it cannot be determined at runtime.

va_end(*pvar*) is used to finish up.

Multiple traversals, each bracketed by **va_start** ... **va_end,** are possible.

Examples

```
#include <varargs.h>
execl(va_alist)
va_dcl
{
        va_list ap;
        char *file;
        char *args[100];
        int argno = 0;

        va_start(ap);
        file = va_arg(ap, char *);
        while (args[argno++] = va_arg(ap, char *))
        B;
        va_end(ap);
        return execv(file, args);
}
```

Restrictions

It is up to the calling routine to determine how many arguments there are, since it is not possible to determine this from the stack frame. For example, `execl` passes a 0 to signal the end of the list. The `printf` command can tell how many arguments are supposed to be there by the format.

Name

vlimit – control maximum system resource consumption

Syntax

#include <sys/vlimit.h>

vlimit(*resource, value*)

Description

This facility has been superseded by `getrlimit`(2).

Limits the consumption by the current process and each process it creates to not individually exceed *value* on the specified *resource*. If *value* is specified as –1, then the current limit is returned and the limit is unchanged. The resources which are currently controllable are:

LIM_NORAISE	Pseudo-limit; if set nonzero then the limits may not be raised. Only the super-user may remove the *noraise* restriction.
LIM_CPU	The maximum number of cpu-seconds to be used by each process.
LIM_FSIZE	The largest single file which can be created.
LIM_DATA	The maximum growth of the data+stack region via `sbrk`(2) beyond the end of the program text.
LIM_STACK	The maximum size of the automatically-extended stack region.
LIM_CORE	the size of the largest core dump that will be created.
LIM_MAXRSS	a soft limit for the amount of physical memory (in bytes) to be given to the program. If memory is tight, the system will prefer to take memory from processes which are exceeding their declared LIM_MAXRSS.

Because this information is stored in the per-process information this system call must be executed directly by the shell if it is to affect all future processes created by the shell; *limit* is thus a built-in command to `csh`(1).

The system refuses to extend the data or stack space when the limits would be exceeded in the normal way. A *break* call fails if the data space limit is reached, or the process is killed when the stack limit is reached. Since the stack cannot be extended, there is no way to send a signal.

A file I/O operation which would create a file which is too large will cause a signal SIGXFSZ to be generated, this normally terminates the process, but may be caught. When the cpu time limit is exceeded, a signal SIGXCPU is sent to the offending process; to allow it time to process the signal it is given 5 seconds grace by raising the cpu time limit.

Restrictions

If LIM_NORAISE is set, then no grace should be given when the CPU time limit is exceeded.

See Also

csh(1)

Name

vtimes – get information about resource utilization

Syntax

vtimes(par_vm, ch_vm)
struct vtimes *par_vm, *ch_vm;

Description

This facility has been superseded by `getrusage`(2).

The `vtimes` routine returns accounting information for the current process and for the terminated child processes of the current process. Either *par_vm* or *ch_vm* or both may be 0, in which case only the information for the pointers which are non-zero is returned.

After the call, each buffer contains information as defined by the contents of the include file /usr/include/sys/vtimes.h:

```
struct vtimes {
      int     vm_utime;               /* user time (*HZ) */
      int    vm_stime;          /* system time (*HZ) */
      /* divide next two by utime+stime to get averages */
      unsigned vm_idsrss;       /* integral of d+s rss */
      unsigned vm_ixrss;        /* integral of text rss */
      int    vm_maxrss;         /* maximum rss */
      int    vm_majflt;         /* major page faults */
      int    vm_minflt;         /* minor page faults */
      int    vm_nswap;          /* number of swaps */
      int    vm_inblk;          /* block reads */
      int    vm_oublk;          /* block writes */
};
```

The *vm_utime* and *vm_stime* fields give the user and system time respectively in 60ths of a second (or 50ths if that is the frequency of wall current in your locality.) The *vm_idrss* and *vm_ixrss* measure memory usage. They are computed by integrating the number of memory pages in use each over cpu time. They are reported as though computed discretely, adding the current memory usage (in 512 byte pages) each time the clock ticks. If a process used 5 core pages over 1 cpu-second for its data and stack, then *vm_idsrss* would have the value 5*60, where *vm_utime+vm_stime* would be the 60. The *vm_idsrss* integrates data and stack segment usage, while *vm_ixrss* integrates text segment usage. The *vm_maxrss* reports the maximum instantaneous sum of the text+data+stack core-resident page count.

The *vm_majflt* field gives the number of page faults which resulted in disk activity; the *vm_minflt* field gives the number of page faults incurred in simulation of reference bits; *vm_nswap* is the number of swaps which occurred. The number of file system input/output events are reported in *vm_inblk* and *vm_oublk* These numbers account only for real I/O. Data supplied by the caching mechanism is charged only to the first process to read or write the data.

See Also

wait3(2), time(3)

X/Open curses Routines (3cur)

Insert tabbed divider here.
Then discard this sheet.

Name

intro – introduction to the X/Open Curses Package, which optimizes terminal screen handling and updating

Syntax

#include <cursesX.h>
cc [*options*] *files* **–lcursesX** [*libraries*]

Description

The `curses` (cursor optimization) package is the X/Open set of library routines used for writing screen-management programs. Cursor optimization minimizes the amount the cursor has to be moved around the screen in order to update it. Screen-management programs are used for tasks such as moving the cursor, printing a menu, dividing a terminal screen into windows or drawing a display on a screen for data entry and retrieval.

The `curses` package is split into three parts: screen updating, screen updating with user input, and cursor motion optimization. Screen-updating routines are used when parts of the screen need to be changed but the overall image remains the same. The cursor motion part of the package can be used separately for tasks such as defining how the cursor moves in response to tabs and newline characters

The `curses` routines do not write directly to the terminal screen (the physical screen): instead, they write to a window, a two-dimensional array of characters which represents all or part of the terminal screen. A window can be as big as the terminal screen or any smaller size down to a single character.

The `<cursesX.h>` header file supplies two default windows, `stdscr` (standard screen) and `curscr` (current screen) for all programs using `curses` routines. The `stdscr` window is the size of the current terminal screen. The `curscr` window is not normally accessed directly by the screen-management program; changes are made to the appropriate window and then the `refresh` routine is called. The screen program keeps track of what is on the physical screen and what is on `stdscr`. When `refresh` is called, it compares the two screen images and then sends a stream of characters to the terminal to make the physical screen look like `stdscr`.

The header file `<cursesX.h>` defines `stdscr` to be of the type `WINDOW*`. This is a pointer to a C structure which includes the starting position of the window on the screen and the window size.

Some `curses` routines are designed to work with a `pad`. A pad is a type of window whose size is not restricted by the size of the screen. Use a pad when you only need part of a window on the screen at any one time, for example when running a spreadsheet application.

Other windows can be created with `newwin` and used instead of `stdscr` for maintaining several different screen images, for example, one window can control input/output and another can display error messages. The routine `subwin` creates subwindows within windows. When windows overlap, the contents of the current screen show the most recently refreshed window.

Among the most basic routines are move and addch. These routines are used to move the cursor around and to add characters to the default window, stdscr.

All curses data is manipulated using the routines provided by the curses library. You should not use routines or system calls from other libraries in a curses program as they may cause undesirable results when you run the program.

Using Curses

The curses library has three types of routines; Main routines, TERMINFO routines and TERMCAP compatibility routines

The terminfo routines are a group of routines within the curses library which provide a database containing descriptions of many terminals that can be used with curses programs. The termcap compatibility routines are provided as a conversion aid for programs using termcap.

Most screen handling can be achieved using the Main routines. The following hints should help you make the most of the screen-handling routines.

The <cursesX.h> header file must always be included whenever curses functions are used in a program. Note that the header file includes <sgtty.h> to enable the terminal to use the features provided by ULTRIX. All the manual definitions assume that <cursesX.h> has been included in the code.

The header file defines global variables and data structures, and defines several of the routines as macros. The integer variables LINES and COLS are defined so that when a curses program is run on a particular terminal, initscr assigns the vertical and horizontal dimensions of the terminal screen to these variables.

A curses program must start by calling the routine initscr to allocate memory space for the windows. It should only be called once in a program, as it can overflow core memory if it is called repeatedly. The routine endwin is used to exit from the screen-handling routines.

Most interactive screen-oriented programs need character-at-a-time input without echoing. To achieve this, you should call:

```
nonl();
cbreak();
noecho();
```

immediately after calling initscr. All curses routines that move the cursor, move it relative to the home position in the upper left corner of the screen. The (LINES, COLS) coordinate at this position is (1,1). Note that the vertical coordinate y is given first and the horizontal coordinate x is given second. The -1 in the example program takes the home position into account to place the cursor on the centre line of the terminal screen. The example program displays **MIDSCREEN** in the centre of the screen. Use the refresh routine after changing a screen to make the terminal screen look like stdscr.

Example Program

```
#include <cursesX.h>
main ()
{

initscr();      /*initialize terminal settings, data
               ** structures and variables*/
move(LINES/2 -1, COLS/2 -4);
```

```
addstr("MID");
refresh();      /* send output to update terminal
                 **  screen */
addstr("SCREEN");
refresh();      /* send more output to terminal
                 **  screen */
endwin();       /*restore all terminal settings */

}
```

Main Routines

Routines listed here can be called when using the `curses` library. Routines that are preceded by a **w** affect a specified window, those preceded by a **p** affect a specified pad. All other routines affect the default window `stdscr.` Windows are specified by a numeric argument, for example: `winch` (*win*) where *win* is the specified window.

addch(ch)	Add a character to stdscr (like putchar wraps to next line at end of line)
addstr(str)	Call addch with each character in *str*
attroff(attrs)	Turn off named attributes
attron(attrs)	Turn on named attributes
attrset(attrs)	Set current attributes to *attrs*
baudrate( )	Display current terminal speed
beep( )	Sound beep on terminal
box(win, vert, hor)	Draw a box around edges of *win*, *vert* and *hor* are characters to use for vertical and horizontal edges of box
clear( )	Clear stdscr
clearok(win, bf)	Clear screen before next redraw of *win*
clrtobot( )	Clear to bottom of stdscr
clrtoeol( )	Clear to end of line on stdscr
cbreak( )	Set cbreak mode
delay_output(ms)	Insert *ms* millisecond pause in output
delch( )	Delete a character
deleteln( )	Delete a line
delwin(win)	Delete *win*
doupdate( )	Update screen from all wnoutrefresh
echo( )	Set echo mode
endwin( )	End window modes
erase( )	Erase stdscr
erasechar( )	Return user's erase character
fixterm( )	Restore tty to in "curses" state
flash( )	Flash screen or beep
flushinp( )	Throw away any typeahead
getch( )	Get a character from tty
getstr(str)	Get a string through stdscr
gettmode( )	Establish current tty modes
getyx(win, y, x)	Get (y, x) coordinates
has_ic( )	True if terminal can do insert character
has_il( )	True if terminal can do insert line
idlok(win, bf)	Use terminal's insert/delete line if bf != 0

inch( )	Get character at current (y, x) coordinates
initscr( )	Initialize screens
insch(c)	Insert a character
insertln( )	Insert a line
intrflush(win, bf)	Interrupt flush output if bf is TRUE
keypad(win, bf)	Enable keypad input
killchar( )	Return current user's kill character
leaveok(win, flag)	Leave cursor anywhere after refresh if flag!=0 for *win*. Otherwise cursor must be left at current position
longname( )	Return verbose name of terminal
meta(win, flag)	Allow meta characters on input if flag != 0
move(y, x)	Move to (y, x) on stdscr

NOTE: The following routines prefixed with **mv** require `y` and `x` coordinates to move to, before performing the same functions as the standard routines. As an example, `mvaddch` performs the same function as `addch`, but `y` and `x` coordinates must be supplied first. The routines prefixed with **mvw** also require a window or pad argument.

mvaddch(y, x, ch)	
mvaddstr(y, x, str)	
mvcur(oldrow, oldcol, newrow, newcol)	low level cursor motion
mvdelch(y, x)	
mvgetch(y, x)	
mvgetstr(y, x)	
mvinch(y, x)	
mvinsch(y, x, c)	
mvprintw(y, x, fmt, args)	
mvscanw(y, x, fmt, args)	
mvwaddch(win, y, x, ch)	
mvwaddstr(win, y, x, str)	
mvwdelch(win, y, x)	
mvwgetch(win, y, x)	
mvwgetstr(win, y, x)	
mvwin(win, by, bx)	
mvwinch(win, y, x)	
mvwinsch(win, y, x, c)	
mvwprintw(win, y, x, fmt, args)	
mvwscanw(win, y, x, fmt, args)	
newpad(nlines, ncols)	Create a new pad with given dimensions
newterm(type, fd)	Set up new terminal of given type to output on fd
newwin(lines, cols, begin_y, begin_x)	Create a new window
nl( )	Set newline mapping
nocbreak( )	Unset cbreak mode
nodelay(win, bf)	Enable nodelay input mode through getch
noecho( )	Unset echo mode
nonl( )	Unset newline mapping
noraw( )	Unset raw mode

overlay(win1, win2)	Overlay win1 on win2
overwrite(win1, win2)	Overwrite win1 on top of win2
pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)	Like prefresh but with no output until doupdate called
prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)	Refresh from pad starting with given upper left corner of pad with output to given portion of screen
printw(fmt, arg1, arg2, ...)	printf on stdscr
raw( )	Set raw mode
refresh( )	Make current screen look like stdscr
resetterm( )	Set tty modes to ''out of curses'' state
resetty( )	Reset tty flags to stored value
saveterm( )	Save current modes as ''in curses'' state
savetty( )	Store current tty flags
scanw(fmt, arg1, arg2, ...)	scanf through stdscr
scroll(win)	Scroll *win* one line
scrollok(win, flag)	Allow terminal to scroll if flag != 0
set_term(new)	Switch between different terminals
setscrreg(t, b)	Set user scrolling region to lines t through b
setupterm(term, filenum, errret)	Low level terminal setup
standend( )	Clear standout mode attribute
standout( )	Set standout mode attribute
subwin(win, lines, cols, begin_y, begin_x)	Create a subwindow
touchwin(win)	''change'' all of *win*
traceoff( )	Turn off debugging trace output
traceon( )	Turn on debugging trace output
typeahead(fd)	Use file descriptor fd to check typeahead
unctrl(ch)	Produce printable version of *ch*
waddch(win, ch)	Add character to *win*
waddstr(win, str)	Add string to *win*
wattroff(win, attrs)	Turn off attrs in *win*
wattron(win, attrs)	Turn on attrs in *win*
wattrset(win, attrs)	Set attrs in *win* to attrs
wclear(win)	Clear *win*
wclrtobot(win)	Clear to bottom of *win*
wclrtoeol(win)	Clear to end of line on *win*
wdelch(win, c)	Delete char from *win*
wdeleteln(win)	Delete line from *win*
werase(win)	Erase *win*
wgetch(win)	Get a character through *win*
wgetstr(win, str)	Get a string through *win*
winch(win)	Get character at current (y, x) in *win*
winsch(win, c)	Insert char into *win*
winsertln(win)	Insert line into *win*
wmove(win, y, x)	Set current (y, x) coordinates on *win*
wnoutrefresh(win)	Refresh but no screen output
wprintw(win, fmt, arg1, arg2, ...)	printf on *win*
wrefresh(win)	Make screen look like *win*
wscanw(win, fmt,	scanf through *win*

arg1, arg2, ...)	
wsetscrreg(win, t, b)	Set scrolling region of *win*
wstandend(win)	Clear standout attribute in *win*
wstandout(win)	Set standout attribute in *win*

Caution

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

TERMINFO Level Routines

If the environment variable TERMINFO is defined, any program using curses will check for a local terminal definition before checking in the standard libraries. For example, if the standard place is /usr/lib/terminfo, and set to **vt100**, the compiled file will normally be found in /usr/lib/terminfo/v/vt100. The **v** is copied from the first letter of vt100 to avoid creating huge directories. However, if TERMINFO is set to /usr/mark/myterms, curses will first check /usr/mark/myterms/v/vt100, and if that fails, will then check /usr/lib/terminfo/v/vt100. This is useful for developing experimental definitions or when there is no write permission for /usr/lib/terminfo.

These routines should be called by programs that need to deal directly with the terminfo database, but as this is a low level interface, it is not recommended.

Initially, the routine setupterm should be called. This will define the set of terminal-dependent variables defined in terminfo(5). The include files <cursesX.h> and <term.h> should be included to get the definitions for these strings, numbers, and flags. Parameterized strings should be passed through tparm to instantiate them. All terminfo strings (including the output of tparm) should be printed with tputs or putp. Before exiting, resetterm should be called to restore the tty modes.

Programs which want shell escapes or <CTRL/Z> suspending can call resetterm before the shell is called and fixterm after returning from the shell.

fixterm( )	Restore tty modes for terminfo use (called by setupterm)
resetterm( )	Reset tty modes to state before program entry
setupterm(term, fd, rc)	Read in database. Terminal type is the character string term, all output is to ULTRIX System file descriptor fd. A status value is returned in the integer pointed to by rc: 1 is normal. The simplest call would be setupterm(0, 1, 0) which uses all defaults
tparm(str, p1, p2, ..., p9)	Instantiate string str with parms p_i
tputs(str, affcnt, putc)	Apply padding info to string str affcnt is the number of lines affected, or 1 if not applicable. Putc is a putchar-like function to which the characters are passed, one at a time

putp(str)	A function that calls tputs (str, 1, putchar)
vidputs(attrs, putc)	Output the string to put terminal in video attribute mode attrs, which is any combination of the attributes listed below Chars are passed to putchar-like function putc
vidattr(attrs)	Like vidputs but outputs through putchar

Termcap Compatibility Routines

The following routines were included as a conversion aid for programs that use termcap. Their parameters are the same as for termcap. They are emulated using the `terminfo` database.

DO NOT use these routines in new programs.

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

As an aid to compatibility, the object module `termcap.o` has been provided in `/usr/lib/termcap.o`. This module should be linked into an application before resolving against the `curses` library. If your application contains references such as `UP` then recompile using

`cc [`*options*`]` *files* `/usr/lib/termcap.o -lcursesX[`*libs*`]`

Errors

No errors are defined for the `curses` functions.

Return Values

For most `curses` routines, the `OK` value is returned if a routine is properly completed and the `ERR` value is returned if some error occurs.

See Also

ioctl(2), getenv(3), printf(3s), putchar(3s), scanf(3s), plot(3x), terminfo(5), tic(1), termcap(5)
Guide to X/Open Curses Screen-Handling

addch (3cur)

Name

addch, waddch, mvaddch, mvwaddch – add character to window

Syntax

```
#include <cursesX.h>

int addch(ch)
chtype ch;

int waddch(win, ch)
WINDOW *win;
chtype ch;

int mvaddch(y, x, ch)
int y, x;
chtype ch;

int mvwaddch(win, y, x, ch)
WINDOW *win;
int y, x;
chtype ch;
```

Description

The routine `addch` inserts the character `ch` into the default window at the current cursor position and the window cursor is advanced. The character is of the type `chtype` which is defined in the `<cursesX.h>` header file, as containing both data and attributes.

The routine `waddch` inserts the character `ch` into the specified window at the current cursor position. The cursor position is advanced.

The routine `mvaddch` moves the cursor to the specified (y, x) position and inserts the character `ch` into the default window. The cursor position is advanced after the character has been inserted.

The routine `mvwaddch` moves the cursor to the specified (y, x) position and inserts the character `ch` into the specified window. The cursor position is advanced after the character has been inserted.

All these routines are similar to `putchar`. The following information applies to all the routines.

If the cursor moves on to the right margin, an automatic newline is performed. If `scrollok` is enabled, and a character is added to the bottom right corner of the screen, the scrolling region will be scrolled up one line. If scrolling is not allowed, ERR will be returned.

If `ch` is a tab, newline, or backspace, the cursor will be moved appropriately within the window. If `ch` is a newline, the `clrtoeol` routine is called before the cursor is moved to the beginning of the next line. If newline mapping is off, the cursor will be moved to the next line, but the x coordinate will be unchanged. If `ch` is a tab the cursor is moved to the next tab position within the window. If `ch` is another control character, it will be drawn in the ^X notation. Calling the `inch` routine after adding a control character returns the representation of the control character, not the control character.

Video attributes can be combined with a character by or-ing them into the parameter. This will result in these attributes being set. The intent here is that text, including attributes, can be copied from one place to another using `inch` and `addch`. For further information, see `standout`(3cur).

The `addch`, `mvaddch`, and `mvwaddch` routines are macros.

Return Value

The `addch`, `waddch`, `mvaddch`, and `mvwaddch` functions return OK on success and ERR on error.

See Also

clrtoeol(3cur), inch(3cur), scrollok(3cur), standout(3cur), putchar(3s)

addstr(3cur)

Name

addstr, waddstr, mvaddstr, mvwaddstr – add string to window

Syntax

#include <cursesX.h>

int addstr(str)
char *str;

int waddstr(win, str)
WINDOW *win;
char *str;

int mvaddstr(y, x, str)
int y, x;
char *str;

int mvwaddstr(win, y, x, str)
WINDOW *win;
int y, x;
char *str;

Description

The `addstr` routine writes all the characters of the null-terminated character string `str` on the default window at the current (y, x) coordinates.

The routine `waddstr` writes all the characters of the null terminated character string `str` on the specified window at the current (y, x) coordinates.

The routine `mvaddstr` writes all the characters of the null terminated character string `str` on the default window at the specified (y, x) coordinates.

The routine `mvwaddstr` writes all the characters of the null terminated character string `str` on the specified window at the specified (y, x) coordinates.

The following information applies to all the routines. All the routines return `ERR` if writing the string causes illegal scrolling. In this case the routine will write as much as possible of the string on the window.

These routines are functionally equivalent to calling `addch` or `waddch` once for each character in the string.

The routines `addstr,` `mvaddstr,` and `mvwaddstr` are macros.

Return Value

The `addstr,` `waddstr,` `mvaddstr,` and `mvwaddstr` functions return OK on success and ERR on error.

See Also

addch(3cur), waddch(3cur)

Name

attroff, attron, attrset, standend, standout, wstandend, wstandout, wattroff, wattron, wattrset – attribute manipulation

Syntax

#include <cursesX.h>

int attroff(attrs)
int attrs;

int wattroff(win, attrs)
WINDOW *win;
int attrs;

int attron(attrs)
int attrs;

int wattron(win, attrs)
WINDOW *win;
int attrs;

int attrset(attrs)
int attrs;

int wattrset(win, attrs)
WINDOW *win;
int attrs;

int standend( )

wstandend(win)
WINDOW *win;

int standout( )

int wstandout(win)
WINDOW *win;

Description

These routines manipulate the current attributes of a window.

The routine `attroff` turns off the named attributes (`attrs`) of the default window without turning any other attributes on or off.

The routine `attron` turns on the named attributes of the default window without affecting any other attributes.

The routine `attrset` sets the current attributes of the default window to the named attributes `attrs`, which is of the type `chtype`, and is defined in the `<cursesX.h>` header file.

The routine `standout` switches on the best highlighting mode available on the terminal for the default window and it is functionally the same as `attron(A_STANDOUT1)`.

The routine standend switches off all highlighting associated with the default window. It is functionally the same as attrset(0), in that it turns off all attributes.

The routine wattroff switches off the named attributes, attrs, for the specified window. Other attributes are not changed.

The routine wattron turns on the named attributes of the specified window without affecting any others.

The routine wattrset sets the current attributes of the specified window to attrs.

The routine wstandout switches on the best highlighting mode available on the terminal for the specified window. Functionally it is the same as wattron(A_STANDOUT1).

The routine wstandend switches off all highlighting associated with the specified window. Functionally it is the same as wattrset(0); that is, it turns off all attributes.

Attributes

Attributes can be any combination of A_STANDOUT, A_REVERSE, A_BOLD, A_DIM, A_BLINK and A_UNDERLINE. These constants are defined in the <cursesX.h> header file. They are also described in the *Guide to X/Open Curses Screen-Handling*. Attributes can be combined with the C language | (or) operator.

The current attributes of a window are applied to all characters that are written into the window with addch or waddch. Attributes are properties of the character, and move with the character through any scrolling and insert/delete line/character operations. Within the restrictions set by the terminal hardware they will be displayed as the graphic rendition of characters put on the screen.

The routines attroff, attron and attrset are macros.

Return Value

The attroff, wattroff, attron, wattron, attrset, wattrset, standend, wstandend, standout, and wstandout functions return OK on success and ERR on error.

See Also

addch(3cur)
Guide to X/Open Curses Screen-Handling

Name

baudrate – return terminal baudrate

Syntax

int baudrate( )

Description

The `baudrate` routine returns the output speed of the terminal in bits per second, for example 9600, as an integer.

Return Value

The `baudrate` function returns the baudrate in bits per second.

beep (3cur)

Name

beep, flash – generate audiovisual alarm

Syntax

#include <cursesX.h>

int beep( )

int flash( )

Description

The `beep` routine sounds the audible alarm on the terminal, if possible, otherwise it flashes the screen.

The routine `flash` flashes the screen, if possible, otherwise it sounds the audible alarm.

If neither signal can be used on a particular terminal, nothing happens.

Return Value

The `beep` and `flash` functions return OK on success and ERR on error.

Name

box – draw box

Syntax

#include <cursesX.h>

int box(win, vert, hor)
WINDOW *win;
chtype vert, hor;

Description

The `box` routine draws a box around the edge of the window. The arguments `vert` and `hor` are the vertical and horizontal characters the box is to be drawn with.

If `vert` and `hor` are 0 or unspecified, then default characters are used.

If scrolling is disabled and the window encompasses the bottom right corner of the screen, all corners are left blank to avoid an illegal scroll.

Return Value

The `box` function returns OK on success and ERR on error.

cbreak (3cur)

Name

cbreak, nocbreak – set/clear cbreak mode

Syntax

int cbreak( )

int nocbreak( )

Description

The routine `cbreak` puts the terminal into CBREAK mode. In this mode, characters typed by the user are immediately available to the program and erase/kill character processing is not performed. Interrupt and flow control characters are unaffected by this mode.

The routine `nocbreak` disables CBREAK. In this case the terminal driver will buffer input until a newline or carriage return is typed.

The initial settings that determine whether or not a terminal is in CBREAK mode are dependent on the terminal driver implementation. As a result of this, it is not possible to determine if a terminal is in CBREAK mode, as it is an inherited characteristic. It is necessary to call `cbreak` to ensure that the terminal is set to the correct mode for the application.

Return Value

The `cbreak` and `nobreak` functions return OK on success and ERR on error.

Name

clear, wclear – clear window

Syntax

#include <cursesX.h>

int clear( )

int wclear(win)
WINDOW *win;

Description

The `clear` routine resets the entire default window to blanks and sets the current (y, x) coordinates to (0, 0).

The routine `wclear` resets the entire specified window to blanks and sets the current (y, x) coordinates to (0, 0).

The `clear` routine assumes that the screen may have garbage on it that it doesn't know about. The routine first calls `erase` which copies blanks to every position in the default window, and then `clearok`, which clears the physical screen completely on the next call to `refresh` for `stdscr`.

The routine `clear` is a macro.

Return Value

The `clear` and `wclear` functions return OK on success and ERR on error.

See Also

clearok(3cur), erase(3cur), refresh(3cur)

clearok(3cur)

Name

clearok – enable screen clearing

Syntax

#include <cursesX.h>

int clearok(win, bf)
WINDOW *win;
bool bf;

Description

If `bf` is `TRUE,` the next call to `refresh`(3cur) for the specified window will clear the window completely and redraw the entire window without changing the original screen's contents. This is useful when the contents of the screen are uncertain. If the window is `stdscr` the entire screen is redrawn.

Return Value

The `clearok` function returns OK on success and ERR on error.

See Also

refresh(3cur)

Name

clrtobot, wclrtobot – clear to end of screen

Syntax

#include <cursesX.h>

int clrtobot( )

int wclrtobot(win)
WINDOW *win;

Description

The `clrtobot` routine begins at the current cursor position in the default window and changes the remainder of the screen to blanks. The current cursor position is also changed to a blank.

The `wclrtobot` routine begins at the current cursor position in the specified window and changes the rest of the screen to blanks, including the current cursor position.

The routine `clrtobot` is a macro.

Return Value

The `clrtobot` and `wclrtobot` functions return OK on success and ERR on error.

Name

clrtoeol, wclrtoeol – clear to end of line

Syntax

#include <cursesX.h>

int clrtoeol( )

int wclrtoeol(win)
WINDOW *win;

Description

The `clrtoeol` routine erases the current line to the right of the cursor, inclusive, on the default window.

The routine `wclrtoeol` erases the current line to the right of the cursor, inclusive, on the specified window.

The routine `clrtoeol` is a macro.

Return Value

The `clrtoeol` and `wclrtoeol` functions return OK on success and ERR on error.

Name

def_prog_mode, def_shell_mode – save terminal modes

Syntax

int def_prog_mode( )

int def_shell_mode( )

Description

The `def_prog_mode` routine saves the current terminal modes as the **program** if the terminal is running under `curses`. The stored terminal modes are used by the `reset_prog_mode`(3cur) routine. This function is used when the user makes a temporary exit from `curses`.

The routine `def_shell_mode` saves the current terminal modes as the **shell** if the terminal is not running under `curses`. The stored terminal modes are used by the `reset_shell_mode`(3cur) routine.

Both routines are called automatically by `initscr`(3cur).

Return Value

The `def_prog_mode` and `def_shell_mode` functions return OK on success and ERR on error.

See Also

initscr(3cur), reset_prog_mode(3cur), reset_shell_mode(3cur)

delay_output (3cur)

Name

delay_output – cause short delay

Syntax

int delay_output(ms)
int ms;

Description

Insert 10 x `ms` millisecond pause in output. The largest number allowed for `ms` is 0.5 seconds (500 milleseconds).

Return Value

The `delay_output` function returns OK on success and ERR on error.

Name

delch, mvdelch, mvwdelch, wdelch – remove character from window

Syntax

#include <cursesX.h>

int delch( )

int wdelch(win)
WINDOW *win;

int mvdelch(y, x)
int y, x;

int mvwdelch(win, y, x)
WINDOW *win;
int y, x;

Description

The `delch` routine deletes the character under the cursor in the default window. All characters to the right on the same line are moved to the left one position and the last character on the line is filled with a blank. The cursor position does not change.

The routine `wdelch` deletes the character under the cursor in the specified window. All characters to the right on the same line are moved to the left one position and the last character on the line is filled with a blank. The cursor position does not change.

The routine `mvdelch` moves the cursor to the specified position in the default window. The character found at this location is deleted. All characters to the right on the same line are moved to the left one position and the last character on the line is filled with a blank. The cursor position does not change.

The routine `mvwdelch` moves the cursor to the specified position in the specified window. The character found at this location is deleted. All characters to the right on the same line are moved to the left one position and the last character on the line is filled with a blank. The cursor position does not change.

The routines `delch,` `mvdelch` and `mvwdelch` are macros.

Return Value

The `delch`, `mvdelch`, `mvwdelch` and `wdelch` functions return OK on success and ERR on error.

Name

deleteln, wdeleteln – remove line from window

Syntax

```
#include <cursesX.h>

int deleteln( )

int wdeleteln(win)
WINDOW *win;
```

Description

The `deleteln` routine deletes the current line of the default window. All lines below the current line are moved up one line. The bottom line of the window is cleared. The cursor position does not change.

The routine `wdeleteln` deletes the current line of the specified window. All lines below the current line are moved up one line. The bottom line of the window is cleared. The cursor position does not change.

The routine `deleteln` is a macro.

Return Value

The `deleteln` and `wdeleteln` functions return OK on success and ERR on error.

Name

delwin – delete window

Syntax

#include <cursesX.h>

int delwin(win)
WINDOW *win;

Description

The `delwin` routine deletes the named window, freeing all memory associated with it. Where windows overlap, subwindows should be deleted before the main window.

Return Value

The `delwin` function returns OK on success and ERR on error.

draino(3cur)

Name

draino – wait for output to drain

Syntax

draino(ms)
int ms;

Description

This function waits until there is only `ms` milliseconds worth of output left in the output queue. The restrictions on the number of milleseconds delay are determined by `napms`(3cur).

See Also

napms(3cur)

Name

echo, noecho – enable/disable terminal echo

Syntax

int echo( )

int noecho( )

Description

The `echo` routine enables echoing of characters typed by the user. The `noecho` routine disables echoing of characters typed by the user.

Initially, input characters are echoed. Subsequent calls to `echo` and `noecho` do not flush typeahead.

Return Value

The `echo` and `noecho` functions return OK on success and ERR on error.

endwin (3cur)

Name

endwin – restore initial terminal environment

Syntax

int endwin( )

Description

This routine restores tty modes, moves the cursor to the lower left corner of the screen and resets the terminal to the last non-`curses` mode.

A program should always call `endwin` before exiting or escaping from `curses` mode temporarily. Call `refresh` or `doupdate` to resume after a temporary escape.

Return Value

The `endwin` function returns OK on success and ERR on error.

See Also

doupdate(3cur), refresh(3cur)

Name

erase, werase – copy blanks into window

Syntax

#include <cursesX.h>

int erase( )

int werase(win)
WINDOW *win;

Description

The `erase` routine copies blanks to every position in the default window, the `werase` routine copies blanks to every position in the specified window.

The routine `erase` is a macro.

Return Value

The `erase` and `werase` functions return OK on success and ERR on error.

erasechar (3cur)

Name

erasechar – return current ERASE character

Syntax

#include <cursesX.h>

char erasechar( )

Description

The user's current erase character is returned.

Return Value

The `erasechar` function returns the user's current erase character.

Name

flushinp – discard typeahead

Syntax

#include <cursesX.h>

int flushinp( )

Description

Any typeahead input that has not been read by the program is discarded.

Return Value

The `flushinp` function returns OK on success and ERR on error.

See Also

typeahead(3cur)

getch (3cur)

Name

getch, mvgetch, mvwgetch, wgetch – read character

Syntax

#include <cursesX.h>

int getch( )

int wgetch(win)
WINDOW *win;

int mvgetch(y, x)
int y, x;

int mvwgetch(win, y, x)
WINDOW *win;
int y, x;

Description

The `getch` routine reads a character from the terminal associated with the default window.

The `wgetch` routine reads a character from the terminal associated with the specified window.

The routine `mvgetch` reads a character from the terminal associated with the default window at the specified position.

The routine `mvwgetch` reads a character from the terminal associated with the specified window at the specified position.

The following information applies to all the routines. In `nodelay` mode, if there is no input waiting, the integer `ERR` is returned. In `delay` mode, the program waits until the system passes text through to the program. Usually the program will restart after one character or after the first newline, but this depends on how `cbreak` is set. The character will be echoed on the designated window unless `noecho` has been set.

If `keypad` is `TRUE,` and a function key is pressed, the token for that function key is returned instead of the raw characters. Possible function keys are defined in the `<cursesX.h>` header file with integers beginning with 0401. The function key names begin with KEY_. Function keys and their respective integer values are described in the *Guide to X/Open Curses Screen-Handling*

If a character is received that could be the beginning of a function key (such as escape), `curses` sets a timer. If the remainder of the sequence does not come within the designated time, the character will be passed through, otherwise the function key value is returned. Consequently, there may be a delay after a user presses the escape key before the escape is returned to the program.

Using the escape key for a single character function is discouraged.

The routines `getch,` `mvgetch` and `mvwgetch` are macros.

Return Value

Upon successful completion, the `getch`, `mvgetch`, and `wgetch` functions return the character read.

If in delay mode and no data is available, ERR is returned.

See Also

cbreak(3cur), keypad(3cur), nodelay(3cur), noecho(3cur)
Guide to X/Open Curses Screen-Handling

getstr (3cur)

Name

getstr, mvgetstr, mvwgetstr, wgetstr – read string

Syntax

#include <cursesX.h>

int getstr(str)
char *str;

int wgetstr(win, str)
WINDOW *win;
char *str;

int mvgetstr(y, x, str)
int y, x;
char *str;

int mvwgetstr(win, y, x, str)
WINDOW *win;
int y, x;
char *str;

Description

The `getstr` routine reads characters from the terminal associated with the default window and stores them in a buffer until a carriage return or newline is received from `stdscr`. The routine `getch` B is called by `getstr` to read each character.

The routine `wgetstr` reads characters from the terminal associated with the specified window. The characters are read from the current cursor position until a newline or carriage return is received.

The routine `mvgetstr` reads characters from the terminal associated with the default window. The characters are read from the specified cursor position until a newline or carriage return is received.

The routine `mvwgetstr` reads characters from the terminal associated with the specified window. The characters are read from the specified cursor position until a newline or carriage return is received.

The following information applies to all the routines.

The resulting string is placed in the area pointed to by the character pointer `str`. The user's erase and kill characters are interpreted. The area used to hold the string is assumed to be large enough to handle it, as `getstr` does not check for buffer overflow. If the area is not large enough, the result will be unpredictable.

The routines `getstr`, `mvgetstr` and `mvwgetstr` are macros.

Return Value

The `getstr`, `mvgetstr`, `mvwgetstr` and `wgetstr` functions return OK on success and ERR on error.

See Also

getch(3cur)

getyx (3cur)

Name

getyx – get cursor position

Syntax

#include <cursesX.h>

int getyx(win, y, x)
WINDOW *win;
int y, x;

Description

The cursor coordinates of the window are placed in the two integer variables `y` and `x`. This routine is implemented as a macro, so no `&` is necessary before the variables.

Return Value

No return value is defined for this function.

Name

has_ic – determine whether insert/delete character available

Syntax

#include <cursesX.h>

bool has_ic( )

Description

True if the terminal has insert- and delete-character capabilities.

The routines `insch` and `delch` are always available in the `curses` library if the terminal does not have the required capabilities.

Return Value

This function returns TRUE if the terminal has insert character and delete character capabilities, otherwise it returns FALSE.

See Also

delch(3cur), insch(3cur)

Name

has_il – determine whether insert/delete line is available

Syntax

#include <cursesX.h>

bool has_il( )

Description

This function will return the value `TRUE` if the terminal has insert- and delete-line capabilities, or if it can simulate them using scrolling regions. This function might be used to check if it would be appropriate to turn on physical scrolling using the `scrollok` routine.

The routines `insertln` and `deleteln` are always available in the `curses` library if the terminal does not have the required facilities.

Return Value

This function returns TRUE if the terminal has insert line and delete line capabilities, or can simulate them using scrolling regions, otherwise it returns FALSE.

See Also

deleteln(3cur), insertln(3cur), scrollok(3cur)

Name

idlok – enable use of insert/delete line

Syntax

#include <cursesX.h>

int idlok(win, bf)
WINDOW *win;
bool bf;

Description

If enabled (`bf` is `TRUE`), `curses` uses the insert/delete line hardware of terminals if it is available. If disabled, `curses` will not use this feature. This option should be enabled only if the application needs insert/delete line; for example, for a screen editor. It is disabled by default as insert/delete line can be visually annoying when used in some applications.

If insert/delete line cannot be used, `curses` will redraw the changed portions of all lines.

NOTE

The terminal hardware insert/delete character feature is always used if available.

Return Value

The `idlok` function returns OK on success and ERR on error.

Name

inch, mvinch, mvwinch, winch – return character from window

Syntax

#include <cursesX.h>

chtype inch()

chtype winch(win)
WINDOW *win;

chtype mvinch(y, x)
int y, x;

chtype mvwinch(win, y, x)
WINDOW *win;
int y, x;

Description

The `inch` routine returns the character at the current cursor position in the default window. If any attributes are set for that character, their values will be `or`-ed into the value returned.

The routine `mvinch` returns the character at the specified position in the default window. If any attributes are set for that position, their values will be `or`-ed into the value returned.

The `winch` routine returns the character at the current position in the named window. If any attributes are set for that position, their values will be `or`-ed into the value returned.

The `mvwinch` routine returns the character at the specified position in the named window. If any attributes are set for that position, their values will be `or`-ed into the value returned.

The following information applies to all the routines.

The predefined constants A_CHARTEXT and A_ATTRIBUTES, defined in `<cursesX.h>`, can be used with the `&` (logical `and`) operator to extract the character or attributes alone.

The `inch, winch, mvinch` and `mvwinch` routines are macros.

Return Value

Upon successful completion, the `inch`, `mvinch`, `mvwinch` and `winch` functions return the character at the selected position. Otherwise, the `mvinch` and `mvwinch` functions return ERR.

Name

initscr – initialize terminal environment

Syntax

#include <cursesX.h>

WINDOW *initscr

Description

This routine determines the terminal type, initializes all `curses` data structures and allocates memory space for the windows. It also arranges that the first call to the `refresh` routine will clear the screen.

The first routine called in a program using `curses` routines should almost always be `initscr`. If errors occur, `initscr` will write an appropriate error message to standard error and exit. If the program needs an indication of error conditions, `newterm` should be used instead of `initscr`.

Note that the `curses` program should only call `initscr` once as it may overflow core memory if it is called repeatedly. If this does occur, ERR is returned.

Return Value

The `initscr` function returns `stdscr` on success, and calls `exit` on error.

See Also

newterm(3cur), refresh(3cur)

insch (3cur)

Name

insch, mvinsch, mvwinsch, winsch – insert character

Syntax

#include <cursesX.h>

int insch(ch)
chtype ch;

int winsch(win, ch)
WINDOW *win;
chtype ch;

int mvinsch(y, x, ch)
int y, x;
chtype ch;

int mvwinsch(win, y, x, ch)
WINDOW *win;
int y, x;
chtype ch;

Description

The `insch` routine inserts the character `ch` at the current cursor position on the default window.

The `mvinsch` routine inserts the character `ch` at the specified cursor position on the default window.

The `winsch` routine inserts the character `ch` at the current cursor position on the specified window.

The `mvwinsch` routine inserts the character `ch` at the specified cursor position on the specified window.

All the routines cause the following actions. All characters from the cursor position to the right edge are moved one space to the right. The last character on the line is always lost, even if it is a blank. The cursor position does not change after the insert is completed.

The `insch, mvinsch` and `mvwinsch` routines are macros.

Return Value

The `insch`, `mvinsch`, `mvwinsch`, and `winsch` functions return OK on success and ERR on error.

Name

insertln, winsertln – insert line

Syntax

#include <cursesX.h>

int insertln( )

int winsertln(win)
WINDOW *win;

Description

The `insertln` routine inserts a blank line above the current line in the default window. All lines below and including the current line are moved down. The bottom line is lost and the current line becomes blank. The (y, x) coordinates are unchanged.

The `winsertln` routine inserts a blank line above the current line on the specified window. All lines below and including the current line are moved down. The bottom line is lost and the current line becomes blank. The (y, x) coordinates are unchanged.

The routine `insertln` is a macro.

Return Value

The `insertln` and `winsertln` functions return OK on success and ERR on error.

intrflush (3cur)

Name

intrflush – enable flush on interrupt

Syntax

#include <cursesX.h>

int intrflush(win, bf)
WINDOW *win;
bool bf;

Description

If `intrflush` is enabled, pressing an interrupt key (interrupt, break, quit) flushes all output in the tty driver queue. This gives the effect of a faster response to the interrupt but causes the `curses` program to have an inaccurate picture of what is on the screen. Disabling the option prevents the flush.

The default for the option is dependent on the tty driver settings. You have to force the terminal into the state you require. The window argument is ignored.

Return Value

The `intrflush` function returns OK on success and ERR on error.

Name

keypad – enable keypad

Syntax

#include <cursesX.h>

int keypad(win, bf)
WINDOW *win;
bool bf;

Description

This option enables the keypad of the user's terminal. If the keypad is enabled, pressing a function key (such as an arrow key) will return a single value representing the function key. For example, pressing the left arrow key results in the value KEY_LEFT being returned.. For more information see the *Guide to X/Open Curses Screen-Handling.*

The routine `getch` is used to return the character. If the keypad is disabled, `curses` does not treat function keys as special keys and the program interprets the escape sequences itself. Keypad layout is terminal dependent; some terminals do not even have a keypad.

Return Value

The `keypad` function returns OK on success and ERR on error.

See Also

getch(3cur)
Guide to X/Open Curses Screen-Handling

killchar (3cur)

Name

killchar – return current kill character

Syntax

#include <cursesX.h>

char killchar( )

Description

The user's current line kill character is returned.

Return Value

The `killchar` function returns the user's current line kill character.

Name

leaveok – enable non-tracking cursor

Syntax

#include <cursesX.h>

int leaveok(win, bf)
WINDOW *win;
bool bf;

Description

This option allows the cursor to be left wherever the update happens to leave it. Normally, the cursor is left at the current location (y, x) of the window being refreshed. This routine is useful for applications where the cursor is not used, since it reduces the need for cursor motions. If possible, the cursor is made invisible when this option is enabled.

This option is initially disabled, and is not enabled until the value of `bf` is changed from `FALSE` to `TRUE`.

Return Value

The `leaveok` function returns OK on success and ERR on error.

longname (3cur)

Name

longname – return full terminal type name

Syntax

char *longname( )

Description

This routine returns a pointer to a static area containing a verbose description of the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to the `initscr` routine or the `newterm` routine.

The static area is overwritten by each call to `newterm` and is not restored by `set_term`. The value should be saved between calls to `newterm` if `longname` is going to be used with multiple terminals.

Return Value

The `longname` function returns a pointer to a verbose description of the current terminal on success and the null pointer on error.

See Also

initscr(3cur), newterm(3cur), set_term(3cur)

Name

meta – force the number of significant bits on input

Syntax

meta(win, bf)
WINDOW *win;
bool bf;

Description

This function forces the user's terminal to return 7 or 8 significant bits on input. To force 8 bits to be returned, invoke `meta` with `bf` as `TRUE`. To force 7 bits to be returned, invoke `meta` with `bf` as `FALSE`.

The window argument is always ignored, but it must still be a valid window to avoid compiler errors.

Name

move, wmove – move cursor in window

Syntax

move(y, x)

wmove(win, y, x)
WINDOW *win;
int y, x;

Description

The `move` routine moves the cursor associated with the default window to the given location (y, x), where `y` is the row, and `x` is the column. This routine does not move the physical cursor of the terminal until the `refresh` routine is called.

The `wmove` routine moves the cursor associated with the specified window to the given location (y, x). This does not move the physical cursor of the terminal until the `wrefresh` routine is called.

For both routines the position specified is relative to the upper left corner of the window, which is (0,0).

The routine `move` is a macro.

See Also

refresh(3cur), wrefresh(3cur)

Name

mvcur – low-level cursor movement

Syntax

mvcur(oldrow, oldcol, newrow, newcol)
int oldrow, oldcol, newrow, newcol;

Description

This function controls low-level cursor motion with optimization.

mvwin (3cur)

Name

mvwin – move window

Syntax

mvwin(win, y, x)
WINDOW *win;
int y, x;

Description

Move the window so that the upper left corner will be at position `(y, x)`. It is an error to move the window off the screen. If you try to do this the window is not moved.

Name

napms – sleep

Syntax

napms(ms)
int ms;

Description

This function causes the program to sleep for `ms` milliseconds. The number of milliseconds is limited to 1000.

newpad (3cur)

Name

newpad – create new pad

Syntax

#include <cursesX.h>

WINDOW *newpad(nlines, ncols)
int nlines, ncols;

Description

The `newpad` routine creates a new `pad` data structure. A pad differs from a window in that it is not restricted by the screen size, and it is not necessarily associated with a particular part of the screen. Pads can be used when large windows are needed. Only part of the pad will be on the screen at any one time.

Automatic refreshes of pads for example, from scrolling or echoing of input, do not occur.

You cannot call the `refresh` routine with a pad as an argument; use the routines `prefresh` or `pnoutrefresh` instead.

Note that these two routines require additional parameters to specify both the part of the pad to be displayed and the screen location for the display.

Return Value

On success the `newpad` function returns a pointer to the new WINDOW structure created. On failure the function returns a null pointer.

See Also

pnoutrefresh(3cur), prefresh(3cur), refresh(3cur)

Name

newterm – open new terminal

Syntax

#include <stdio.h>
#include <cursesX.h>

SCREEN *newterm(type, outfd, infd)
char *type;
FILE *outfd, *infd;

Description

Programs using more than one terminal should call the `newterm` routine for each terminal instead of `initscr`. The routine `newterm` should be called ONCE for each terminal.

The `newterm` routine returns a variable of type SCREEN * which should be saved as a reference to that terminal. There are three arguments. The first argument `type`, is the type of the terminal to be used in place of TERM. The second argument, `outfd`, is a file pointer for output to the terminal. The third argument, `infd`, is a file pointer for input from the terminal. The program must also call the `endwin` routine for each terminal, after each terminal has finished running a `curses` application.

Return Value

On success the `newterm` function returns a pointer to the new SCREEN structure created. On failure the function returns a null pointer.

See Also

endwin(3cur), initscr(3cur)

Name

newwin – create new window

Syntax

#include <cursesX.h>

WINDOW *newwin(nlines, ncols, begin_y, begin_x)
int nlines, ncols, begin_y, begin_x;

Description

The function `newwin` creates a new window with the number of lines, `nlines`, and columns, `ncols`. The upper left corner of the window is at line `begin_y`, column `begin_x`.

If either `nlines` or `ncols` is zero, they will be defaulted to LINES - `begin_y` and COLS - `begin_x`. A new full-screen window is created by calling `newwin`(0,0,0,0).

Return Value

On success the `newwin` function returns a pointer to the new WINDOW structure created. On failure the function returns a null pointer.

Name

nl, nonl – enable/disable newline control

Syntax

#include <cursesX.h>

int nl( )

int nonl( )

Description

The `nl` routine enables the newline control translations. When newline control is enabled, a newline is translated into a carriage return and a linefeed on output, and a return is translated into a newline on input. Initially, these translations do occur.

The `nonl` routine disables these translations, allowing the `curses` program to use the linefeed capability of the terminal, resulting in faster cursor motion. The `nl` routine is a macro.

Return Value

The `nl` and `nonl` functions return OK on success and ERR on error.

nodelay(3cur)

Name

nodelay – disable block during read

Syntax

#include <cursesX.h>

int nodelay(win, bf)
WINDOW *win;
bool bf;

Description

This option causes the `getch` routine to be a non-blocking call. If no input is ready, and `nodelay` is enabled, `getch` will return the integer `ERR`. If `nodelay` is disabled, `getch` will wait until input is ready.

Return Value

The `nodelay` function returns OK on success and ERR on error.

See Also

getch(3cur)

Name

overlay, overwrite – overlay windows

Syntax

#include <cursesX.h>

int overlay(srcwin, dstwin)
WINDOW *srcwin, *dstwin;

int overwrite(srcwin, dstwin)
WINDOW *srcwin, *dstwin;

Description

The `overlay` routine copies all the text from the source window `srcwin` on top of the destination window `dstwin`. The two windows are not required to be the same size. The copy starts at (0, 0) on both windows. The copy is non-destructive, so blanks are not copied.

The `overwrite` routine copies all of `srcwin` on top of `destwin`. The copy starts at (0, 0) on both windows. This is a destructive copy as blanks are copied.

Return Value

The `overlay` and `overwrite` functions return OK on success and ERR on error.

Name

prefresh, pnoutrefresh – refresh pad

Syntax

#include <cursesX.h>

int prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
WINDOW *pad;
int pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol;

int pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
WINDOW *pad;
int pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol;

Description

The `prefresh` routine copies the specified pad to the physical terminal screen. It takes account of what is already displayed on the screen to optimize cursor movement.

The `pnoutrefresh` routine copies the named pad to the virtual screen. It then compares the virtual screen with the physical screen and performs the actual update.

These routines are analogous to the routines `wrefresh` and `wnoutrefresh` except that pads, instead of windows, are involved. Additional parameters are also needed to indicate what part of the pad and screen are involved. The upper left corner of the part of the pad to be displayed is specified by `pminrow` and `pmincol`. The co-ordinates `sminrow`, `smincol`, `smaxrow`, and `smaxcol` specify the edges of the screen rectangle that will contain the selected part of the pad.

The lower right corner of the pad rectangle to be displayed is calculated from the screen co-ordinates. This ensures that the screen rectangle and the pad rectangle are the same size.

Both rectangles must be entirely contained within their respective structures.

Return Value

The `prefresh` and `pnoutrefresh` functions return OK on success and ERR on error.

See Also

wnoutrefresh(3cur), wrefresh(3cur)

Name

printw, mvprintw, mvwprintw, wprintw – formatted write to a window

Syntax

#include <cursesX.h>

int printw(fmt [, arg] ...)
char *fmt;

int wprintw(win, fmt [, arg] ...)
WINDOW *win;
char *fmt;

int mvprintw(y, x, fmt [, arg] ...)
int y, x;
char *fmt;

int mvwprintw(win, y, x, fmt [, arg] ...)
WINDOW *win;
int y, x;
char *fmt;

Description

The `printw` routine adds a string to the default window starting at the current cursor position. This routine causes the string that would normally be output by `printf` to be output by `addstr`.

The routine `wprintw` adds a string to the specified window starting at the current cursor position. This routine causes the string that would normally be output by `printf` to be output by `waddstr`.

The routine `mvprintw` adds a string to the default window starting at the specified cursor position. This routine causes the string that would normally be output by `printf` to be output by `addstr`.

The routine `mvwprintw` adds a string to the specified window starting at the specified cursor position. This routine causes the string that would normally be output by `printf` to be output by `waddstr`.

All these routines are analogous to `printf`. It is advisable to use the field width options of `printf` to avoid leaving unwanted characters on the screen from earlier calls.

Return Values

The `printw`, `mvprintw`, `mvwprintw`, and `wprintw` functions return OK on success and ERR on error.

See Also

addstr(3cur), waddstr(3cur), printf(3s)

Name

putp – pad and output a string

Syntax

putp(str)
char *str;

Description

The `putp` routine outputs the string `str` one character at a time. The routine `putchar` is used to control the output.

See Also

putchar(3s)

Name

raw, noraw – enable/disable raw mode

Syntax

int raw( )

int noraw( )

Description

The raw routine sets the terminal into RAW mode. RAW mode is similar to CBREAK mode, in that characters are immediately passed through to the user program as they are typed. In RAW mode, the interrupt, quit, suspend and flow control characters are passed through uninterpreted, and do not generate a signal.

The behavior of the BREAK key depends on the settings of bits that are not controlled by curses.

The noraw routine disables RAW mode.

Return Value

The raw and noraw functions return OK on success and ERR on error.

refresh (3cur)

Name

refresh, wrefresh – refresh window

Syntax

#include <cursesX.h>

int refresh( )

int wrefresh(win)
WINDOW *win;

Description

The routine `wrefresh` copies the named window to the physical terminal screen, taking into account what is already there in order to optimize cursor movement.

The routine `refresh` does the same, using `stdscr` as a default screen.

These routines **must** be called to get any output on the terminal, as other routines only manipulate data structures.

Unless `leaveok` has been enabled, the physical cursor of the terminal is left at the location of the window's cursor. The routine `refresh` is a macro.

Return Value

The `refresh` and `wrefresh` functions return OK on success and ERR on error.

See Also

leaveok(3cur)

Name

resetty, savetty – restore/save terminal modes

Syntax

int resetty( )

int savetty( )

Description

The `savetty` routine saves the current state of the terminal modes in a buffer. The routine `resetty` restores the state of the terminal modes to what it was at the last call to `savetty`.

Return Value

The `resetty` and `savetty` functions return OK on success and ERR on error.

reset_prog_mode (3cur)

Name

reset_prog_mode, reset_shell_mode – restore terminal mode

Syntax

int reset_prog_mode( )

int reset_shell_mode( )

Description

The `reset_prog_mode` routine restores the terminal modes to those saved by the `def_prog_mode` routine.

The `reset_shell_mode` routine restores the terminal modes saved by the `def_shell_mode` routine.

These routines are called automatically by `endwin` and `doupdate` after an `endwin`. Normally these routines would not be called before `endwin`.

Return Value

The `reset_prog_mode` and `reset_shell_mode` functions return OK on success and ERR on error.

See Also

def_prog_mode(3cur), def_shell_mode(3cur), doupdate(3cur), endwin(3cur)

Name

restartterm – restart terminal for curses application

Syntax

restartterm(term, filenum, errret)
char *term;
int filenum;
int *errret;

Description

This function sets up the current terminal `term` after a save/restore of a `curses` application program. `restartterm` assumes that the windows and modes are the same for the restarted application as when memory was saved. It assumes that the terminal type and dependent settings, such as baudrate, may have changed. The routine `setupterm` is called to extract the terminal information from the `terminfo` database and set up the terminal.

See Also

setupterm(3cur), terminfo(5)

scanw(3cur)

Name

scanw, mvscanw, mvwscanw, wscanw – formatted read from window

Syntax

#include <cursesX.h>

int scanw(fmt [, arg] ...)
char *fmt;

int wscanw(win, fmt [, arg] ...)
WINDOW *win;
char *fmt;

int mvscanw(y, x, fmt [, arg] ...)
int y, x;
char *fmt;

int mvwscanw(win, y, x, fmt [, arg] ...)
WINDOW *win;
int y, x;
char *fmt;

Description

These routines correspond to `scanf`. The function `scanw` reads input from the default window. The function `wscanw` reads input from the specified window. The function `mvscanw` moves the cursor to the specified position and then reads input from the default window. The function `mvwscanw` moves the cursor to the specified position and then reads input from the specified window.

For all the functions, the routine `wgetstr` is called to get a string from the window, and the resulting line is used as input for the scan. All character interpretation is carried out according to the `scanf` function rules.

Return Value

Upon successful completion, the `scanw`, `mvscanw`, `mvwscanw` and `wscanw` functions return the number of items successfully matched. On end-of-file, they return EOF. Otherwise they return ERR.

See Also

wgetstr(3cur), scanf(3s)

Name

scroll – scroll window

Syntax

#include <cursesX.h>

int scroll(win)
WINDOW *win;

Description

The window is scrolled up one line. This involves moving the lines in the window data structure.

You would not normally use this routine as the terminal scrolls automatically if `scrollok` is enabled. A typical case where `scroll` might be used is with a screen editor.

Return Value

The `scroll` function returns OK on success and ERR on error.

See Also

scrollok(3cur)

scrollok (3cur)

Name

scrollok – enable screen scrolling

Syntax

#include <cursesX.h>

int scrollok(*win, bf*)
WINDOW **win*;
bool *bf*;

Description

This option controls what happens when the cursor is moved off the edge of the specified window or scrolling region, either from a newline on the bottom line, or typing the last character of the last line. If disabled, (*bf* is FALSE) the cursor is left on the bottom line. If enabled, the window is scrolled up one line and then refreshed.

Return Value

The `scrollok` function returns OK on success and ERR on error.

Name

setscrreg, wsetscrreg – set scrolling region

Syntax

#include <cursesX.h>

int setscrreg(top, bot)
int top, bot;

int wsetscrreg(win, top, bot)
WINDOW *win;
int top, bot;

Description

The `setscrreg` routine sets the scrolling region for the default window.

The `wsetscrreg` routine sets the scrolling region for the named window. Use these routines to set a software scrolling region in a window.

For both routines, the line numbers of the top and bottom margins of the scrolling region are contained in `top` and `bot`. Line 0 is the top line of the window.

If this option and `scrollok` are enabled, an attempt to move off the bottom margin line will cause all lines in the scrolling region to scroll up one line. Only the text of the window is scrolled.

Return Value

No return values are defined for these functions.

See Also

scrollok(3cur)

setupterm (3cur)

Name

setupterm – perform low level terminal setup

Syntax

setupterm(term, filenum, errret)
char *term;
int filenum;
int *errret;

Description

This function sets up the terminal from the `terminfo` database. The parameter `term` is the terminal type. If this parameter is set to `NULL` then the environment variable `TERM` will be used. The `filenum` parameter is an ULTRIX file descriptor, not a `stdio` pointer. It is used for all the output generated by `setupterm`.

The `terminfo` boolean, numeric and string values are stored in a structure of type `TERMINAL`.

After `setupterm` returns successfully the variable `cur_term` is initialized. This variable points to the `TERMINAL` structure. The `cur_term` pointer can be saved before calling `setupterm` again as further calls to `setupterm` allocate new space; the space pointed to by `cur_term` is not overwritten.

See Also

restartterm(3cur)

Name

set_term – switch between terminals

Syntax

#include <cursesX.h>

SCREEN *set_term(new)
SCREEN *new;

Description

This routine is used to switch between different terminals. The screen reference `new` becomes the new current terminal. The previous terminal screen reference is returned by the routine.

This is the only routine which manipulates SCREEN pointers; all the others change the current terminal only.

Return Value

The `set_term` function returns a pointer to the previous SCREEN structure on success and a null pointer on error.

subwin (3cur)

Name

subwin – create subwindow

Syntax

#include <cursesX.h>

WINDOW *subwin(orig, nlines, ncols, begin_y, begin_x)
WINDOW *orig;
int nlines, ncols, begin_y, begin_x;

Description

This routine creates a new sub-window within a window. The dimensions of the sub-window are `nlines` lines and `ncols` columns. The sub-window is at position `(begin_y, begin_x)` on the screen. This position is relative to the screen, and not to the window `orig`.

The sub-window is made in the middle of the window `orig`, so that changes made to either window will affect both. When using this routine, it will often be necessary to call `touchwin` before calling `wrefresh`.

Return Value

On success the `subwin` function returns a pointer to the new WINDOW structure created. On failure the function returns a null pointer.

See Also

touchwin(3cur), wrefresh(3cur)

Name

tgetent, tgetnum, tgoto, tgetstr, tgetflag – emulate termcap for old programs

Syntax

int tgetent(bp, name)
char *bp, *name;

int tgetflag(id)
char *id;

tgetnum(id)
char *id;

tgetstr(id, area)
char *id, *area;

tgoto(cap, col, row)
char *cap;
int col, row;

Description

All these functions are included for compatibility with application programs that used the old `termcap` database.

Do not use these functions in new `curses` application programs.

touchwin (3cur)

Name

touchwin – touch window

Syntax

#include <cursesX.h>

int touchwin(win)
WINDOW *win;

Description

This routine discards all optimization information for the specified window and assumes that the entire window has been drawn on.

This is sometimes necessary when using overlapping windows, as a change to one window will affect the other window. The records of which lines have been changed may not be correct for the window which has not been changed directly.

Return Value

The `touchwin` function returns OK on success and ERR on error.

Name

tparm – instantiate a string

Syntax

char *tparm(str, p1, p2, ...)

Description

This function instantiates the string `str` with the parameters p1, p2, ... A pointer is returned which points to the result of `str` with the parameters applied.

tputs (3cur)

Name

tputs – pad and output string

Syntax

tputs(str, count, putc)
register char *str;
int count;
int (*putc)();

Description

This function adds padding to the string `str` and outputs it. The string must be either a `terminfo` string variable or the return value from `tparm`, `tgetstr` or `tgoto`. The variable `count` is the number of lines affected; this is set to 1 if not applicable. The function `putc` is a `putchar` style routine. The characters are passed to `putc` one at a time.

See Also

putchar(3s), terminfo(5), tparm(3cur)

Name

traceon, traceoff – enable or disable debug trace output

Syntax

traceon()

traceoff()

Description

These functions turn the debugging trace output on and off when you use the debug version of the `curses` library `/usr/lib/libdcursesX.a`.

typeahead (3cur)

Name

typeahead – check for typeahead

Syntax

int typeahead(fd)
int fd;

Description

If `typeahead` is enabled, the `curses` program looks for typeahead input periodically while updating the screen. If input is found, the current update will be postponed until `refresh` or `doupdate` is called again. This allows faster response to commands typed in advance.

Normally, the input FILE pointer passed to the `newterm` routine, will be used to do this typeahead checking. If the routine `initscr` was called, the input FILE pointer is passed to `stdin`.

The `typeahead` routine specifies that the file descriptor `fd` is to be used to check for typeahead. If `fd` is –1, then typeahead is disabled.

Return Value

No return values are defined for this function.

See Also

doupdate(3cur), initscr(3cur), newterm(3cur), refresh(3cur)

Name

unctrl – convert character to printable form

Syntax

#include <cursesX.h>

char *unctrl(c)
chtype c;

Description

The `unctrl` routine expands the character `c` into a character string which is a printable representation of the character.

Control characters are displayed in the ^X notation. Printing characters are displayed normally. The `unctrl` routine is a macro, defined in the `unctrl.h` header file. This header file is included by the `cursesX.h` header file described in `intro`(3cur), so you do not have to include it again.

Return Value

The `unctrl` macro returns a string.

See Also

intro(3cur)

Name

vidattr, vidputs – output a string that sets terminal display

Syntax

vidattr(attrs)
vidputs(attrs, putc)

Description

The `vidattr` routine outputs a string that sets the video attributes `attrs` for the terminal. The characters in the string are passed one at a time to the routine `putchar`.

The `vidputs` routine is similar, except that the string characters are passed to the routine `putc`. Video attributes are described in *The Guide to X/Open Curses Screen-Handling*

See Also

putchar(3s)
Guide to X/Open Curses Screen-Handling

Name

wnoutrefresh, doupdate – do efficient refresh

Syntax

#include <cursesX.h>

int wnoutrefresh(win)
WINDOW *win;

int doupdate( )

Description

The `wnoutrefresh` routine updates screens more efficiently than using the `wrefresh` routine by itself. The `wnoutrefresh` routine copies the named window to a data structure referred to as the virtual screen (stdscr). The virtual screen contains what a program intends to display on the physical terminal screen. The routine `doupdate` compares the virtual screen to the physical screen and then does the actual update. These two routines allow multiple updates with more efficiency than `wrefresh`.

The routine `wrefresh` works by calling `wnoutrefresh`, and then calling `doupdate`. If a programmer wants to output several windows at once, a series of calls to `wrefresh` will result in alternating calls to `wnoutrefresh` and `doupdate`, causing several bursts of output to the screen. If `wnoutrefresh` is called first for each window, `doupdate` only needs to be called once, resulting in only one burst of output. This usually results in fewer total characters being transmitted and less CPU time used.

Return Value

The `doupdate` and `wnoutrefresh` functions return OK on success and ERR on error.

See Also

wrefresh(3cur)

Internationalization Routines (3int)

Insert tabbed divider here.
Then discard this sheet.

Name

intro – introduction to international subroutines

Description

The internationalization package provides a convenient method of writing or converting applications so that they can operate in the application user's natural language.

The package consists of the following:

- Tools for the creation and modification of message catalogs
- An international function library, which is called *libi*
- A set of international functions available in the C library, *libc*
- An international compiler that creates language support databases from special source files
- An announcement and initialization mechanism
- A utility for converting data from one codeset to another codeset

When you use international library functions in a C program, compile it with the `-li` option to include *libi,* as shown:

```
% cc -o prog prog.c -li
```

Some of the international functions are available in the standard C library. You need not compile with the `-li` option if you use only those functions. The functions that are available in the standard C library are `setlocale`, `strftime`, `strxfrm`, and `strcoll`.

Libraries

	Internationalization Library Calls
catgetmsg	get message from a message catalog (provided for XPG–2 compatibility)
catgets	read a program message
catopen	open or close a message catalog
nl_init	set localization for internationalized program (provided for XPG–2 compatibility)
nl_langinfo	language information
nl_printf	print formatted output (provided for XPG–2 compatibility)
nl_scanf	convert formatted input (provided for XPG–2 compatibility)
printf	print formatted output
scanf	convert formatted input
vprintf	print formatted output of varargs argument list
	Standard C Library Calls
setlocale	set localization for internationalized program
strftime	convert time and date to string
strxfrm	string transformation
strcoll	string collation comparison

Header Files

i_defs.h	contains language support database structure
i_errno.h	contains error numbers and messages
langinfo.h	contains the langinfo definitions for the locale database
locale.h	contains the declarations used by the ANSI setlocale and localeconv functions
nl_types.h	contains the definitions for all the internationalization (libi) functions

See Also

iconv(1), extract(1int), gencat(1int), ic(1int), strextract(1int), strmerge(1int), trans(1int), ctype(3), setlocale(3), strcoll(3), strftime(3), strxfrm(3), catgets(3int), catopen(3int), nl_langinfo(3int), printf(3int), scanf(3int), vprintf(3int), environ(5int), lang(5int), nl_types(5int), patterns(5int)
Guide to Developing International Software

Name

catgetmsg – get message from a message catalog

Syntax

#include <nl_types.h>

nl_catd *catd*;
int *set_num, msg_num, buflen*;
char **buf*;

Description

The `catgetmsg` function has been superceded by the `catgets` function. You should use the `catgets` function to get messages from a message catalog. You might want to rewrite calls to the `catgetmsg` function so that they use the `catgets` function. The `catgetmsg` function is available for compatibility with XPG–2 conformant software and might not be available in the future. For more information on using `catgets`, see the `catgets`(3int) reference page.

The function `catgetmsg` attempts to read up to *buflen* –1 bytes of a message string into the area pointed to by *buf* . The parameter `buflen` is an integer value containing the size in bytes of *buf*. The return string is always terminated with a null byte.

The parameter *catd* is a catalog descriptor returned from an earlier call to `catopen` and identifies the message catalog containing the message set (*set_num*) and the program message (*msg_num*).

The arguments *set_num* and *msg_num* are defined as integer values for maximum portability. Where possible, you should use symbolic names for message and set numbers, rather hard-coding integer values into your source programs. If you use symbolic names, you must include the `#include` file `gencat -h` creates in all the program modules.

Return Value

If successful, `catgetmsg` returns a pointer to the message string in *buf*. Otherwise, if *catd* is invalid or if *set_num* or *msg_num* are not in the message catalog, `catgetmsg` returns a pointer to an empty (null) string.

See Also

intro(3int), gencat(1int), catopen(3int), catgets(3int), nl_types(5int)
Guide to Developing International Software

catgets (3int)

Name

catgets – read a program message

Syntax

#include <nl_types.h>

char *catgets (*catd, set_num, msg_num, s*)
nl_catd *catd*;
int *set_num, msg_num*;
char **s*;

Description

The function `catgets` attempts to read message *msg_num* in set *set_num* from the message catalog identified by *catd*. The parameter *catd* is a catalog descriptor returned from an earlier call to `catopen`. The pointer, *s*, points to a default message string. The `catgets` function returns the default message if the identified message catalog is not currently available.

The `catgets` function stores the message text it returns in an internal buffer area. This buffer area might be written over by a subsequent call to `catgets`. If you want to re-use or modify the message text, you should copy it to another location.

The arguments *set_num* and *msg_num* are defined as integer values to make programs that contain the `catgets` call portable. Where possible, you should use symbolic names for message and set numbers, instead of hard-coding integer values into your source programs. If you use symbolic names, you must include the header file that `gencat &-h` creates in all your program modules.

Examples

The following example shows using the `catgets` call to retrieve a message from a message catalog that uses symbolic names for set and message numbers:

```
nl_catd catd = catopen (messages.msf, 0)
message = catgets (catd, error_set, bad_value, "Invalid value")
```

When this call executes, `catgets` searches for the message catalog identified by the catalog descriptor stored in `catd`. The function searches for the message identified by the `bad_value` symbolic name in the set identified by the `error_set` symbolic name and stores the message text in `message`. If `catgets` cannot find the message, it returns the message Invalid value.

Return Values

If `catgets` successfully retrieves the message, it returns a pointer to an internal buffer area containing the null terminated message string. If the call is unsuccessful for any reason, *catgets* returns the default message in *s*.

See Also

intro(3int), gencat(1int), catgetmsg(3int), catopen(3int), nl_types(5int)
Guide to Developing International Software

Name

catopen, catclose – open/close a message catalog

Syntax

#include <nl_types.h>

nl_catd catopen (*name*, *oflag*)
char *name;
int oflag;

int catclose (*catd*)
nl_catd catd;

Description

The function `catopen` opens a message catalog and returns a catalog descriptor. The parameter *name* specifies the name of the message catalog to be opened. If *name* contains a slash (/), then *name* specifies a pathname for the message catalog. Otherwise, the environment variable `NLSPATH` is used with *name* substituted for *%N*. For more information, see `environ`(5int) in the *ULTRIX Reference Pages*. If `NLSPATH` does not exist in the environment, or if a message catalog cannot be opened in any of the paths specified by `NLSPATH`, the current directory is used.

The *oflag* is reserved for future use and must be set to zero (0). The results of setting this field to any other value are undefined.

The function `catclose` closes the message catalog identified by `catd`.

Restrictions

Using `catopen` causes another file descriptor to be allocated by the calling process for the duration of the `catopen` call.

Return Value

If successful, `catopen` returns a message catalog descriptor for use on subsequent calls to `catgetmsg`, `catgets` and `catclose`. If unsuccessful, `catopen` returns `(nl_catd)-1`.

The `catclose` function returns 0 if successful, otherwise -1.

See Also

intro(3int), setlocale(3), catgetmsg(3int), catgets(3int), environ(5int), nl_types(5int)
Guide to Developing International Software

nl_langinfo (3int)

Name

nl_langinfo – language information

Syntax

#include <nl_types.h>
#include <langinfo.h>

char *nl_langinfo (*item*)
nl_item *item*;

Description

The function `nl_langinfo` returns a pointer to a null-terminated string containing information relevant to a particular language or cultural area. The language is identified by the last successful call to the appropriate `setlocale` category. The categories are shown in the following table and are defined in `<langinfo.h>`.

For instance, the following example would return a pointer to the string representing the abbreviated name for the first day of the week, as defined by `setlocale` category `LC_TIME`:

```
nl_langinfo (ABDAY_1);
```

If the `setlocale` category has not been called successfully, `langinfo` data for a supported language is not available, or `item` is not defined, then `nl_langinfo` returns a pointer to an empty (null) string. In the C locale, the return value is the American English string defined in the following table:

Identifier	Meaning	C locale	Category
NOSTR	Negative response	no	LC_ALL
YESSTR	Positive response	yes	LC_ALL
D_T_FMT	Default date and time format	%a %b %d %H:%M:%S %Y	LC_TIME
D_FMT	Default date format	%m/%d/%y	LC_TIME
T_FMT	Default time format	%h:%m:%s	LC_TIME
DAY_1	Day name	Sunday	LC_TIME
DAY_2	Day name	Monday	LC_TIME
....	....	....	....
DAY_7	Day name	Saturday	LC_TIME
ABDAY_1	Abbreviated day name	Sun	LC_TIME
ABDAY_2	Abbreviated day name	Mon	LC_TIME
ABDAY_3	Abbreviated day name	Tue	LC_TIME
....	....	....	....
ABDAY_7	Abbreviated day name	Sat	LC_TIME
MON_1	Month name	January	LC_TIME
MON_2	Month name	February	LC_TIME
MON_3	Month name	March	LC_TIME
....	....	....	....

MON_12	Month name	December	LC_TIME
ABMON_1	Abbreviated month name	Jan	LC_TIME
ABMON_2	Abbreviated month name	Feb	LC_TIME
....	....	....	....
ABMON_12	Abbreviated month name	Dec	LC_TIME
RADIXCHAR	Radix character		LC_NUMERIC
THOUSEP	Thousands separator		LC_NUMERIC
CRNCYSTR	Currency format		LC_MONETARY
AM_STR	String for AM	AM	LC_TIME
PM_STR	String for PM	PM	LC_TIME
EXPL_STR	Lower case exponent character	e	LC_NUMERIC
EXPU_STR	Upper case exponent character	E	LC_NUMERIC

See Also

intro(3int), ic(1int), setlocale(3int), environ(5int), nl_types(5int)
Guide to Developing International Software

nl_printf(3int)

Name

nl_printf, nl_fprintf, nl_sprintf – print formatted output

Syntax

#include <stdio.h>

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

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

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

Description

The international functions `nl_printf`, `nl_fprintf`, and `nl_sprintf` are identical to and have been superceded by the international functions `printf`, `fprintf`, and `sprintf` in a library. You should use the `printf`, `fprintf`, and `sprintf` functions when you write new calls to print formatted output in an international program. For more information on these functions, see the `printf`(3int) reference page.

You can continue to use existing calls to the `nl_printf`, `nl_fprintf`, or `nl_sprintf` international functions. These functions remain available for compatibility with XPG–2 conformant software, but may not be supported in future releases of the ULTRIX system.

The `nl_printf`, `nl_fprintf`, and `nl_sprintf` international functions are similar to the `printf` standard I/O function. (For more information about the `printf` standard I/O function, see the `printf`(3s) reference page.) The difference is that the international functions allow you to use the *I%digit$* conversion sequence in place of the % character you use in the standard I/O functions. The *digit* is a decimal digit *n* from 1 to 9. The international functions apply conversions to the *n* th argument in the argument list, rather than to the next unused argument.

You can use % conversion character in the international functions. However, you cannot mix the % conversion character with the *%digit$* conversion sequence in a single call.

You can indicate a field width or precision by an asterisk (*), instead of a digit string, in `format` strings containing the % conversion character. If you use an asterisk, you can supply an integer argument that specifies the field width or precision. In `format` strings containing the *%digit$* conversion character, you can indicate field width or precision by the sequence **digit$*. You use a decimal digit from 1 to 9 to indicate which argument contains an integer that specifies the field width or precision.

The conversion characters and their meanings are identical to `printf`.

You must use each *digit* argument at least once. The results of not using an argument are undefined.

International Environment

LC_NUMERIC If this environment is set and valid, `nl_printf` uses the international language database named in the definition to determine radix character rules.

LANG If this environment variable is set and valid `nl_printf` uses the international language database named in the definition to determine collation and character classification rules. If `LC_NUMERIC` is defined, its definition supercedes the definition of LANG.

Examples

The following example illustrates using an argument to specify field width:

```
nl_printf ("%1$d:%2$.*3$d:%4$.*3$d\n",
                        hour, min, precision, sec);
```

The format string **3$* indicates that the third argument, which is named precision, contains the integer field width specification.

To print the language independent date and time format, use the following `nl_printf` statement:

```
nl_printf (format, weekday, month, day, hour, min);
```

For United States of America use, `format` could be a pointer to the following string:

```
"%1$s,  %2$s %3$d, %4$d:%5$.2d\n"
```

This `format` string produces the following message:

```
Sunday, July 3, 10:02
```

For use in a German environment, `format` could be a pointer to the following string:

```
"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

This `format` produces the following message:

```
Sonntag, 3. Juli, 10:02
```

See Also

intro(3int), setlocale(3), nl_scanf(3int), printf(3int), scanf(3int), printf(3s), putc(3s), scanf(3s), stdio(3s)
Guide to Developing International Software

Name

nl_scanf, nl_fscanf, nl_sscanf – convert formatted input

Syntax

#include <stdio.h>

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

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

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

Description

The international functions `nl_scanf`, `nl_fscanf`, and `nl_sscanf` are identical to and have been superceded by the international functions `scanf`, `fscanf`, and `sscanf` in *libi*. You should use the `scanf`, `fscanf`, and `sscanf` functions when you write new calls to convert formatted input in international programs. For more information on these functions, see the `scanf`(3int) reference page.

You can continue to use existing calls to the `nl_scanf`, `nl_fscanf`, or `nl_sscanf` functions. These functions remain available for compatibility with XPG–2 conformant software, but may not be supported in future releases of the ULTRIX system.

The `nl_scanf`, `nl_fscanf`, and `nl_sscanf` international functions are similar to the `scanf` standard I/O function. (For more information on the `scanf` standard I/O function, see `scanf`(3s) reference page.) The difference is that the international functions allow you to use the *%digit$* conversion character in place of the % character you use in the standard I/O functions. The *digit* is a decimal digit *n* from 1 to 9. The international functions apply conversions to the *n* th argument in the argument list, rather than to the next unused argument.

You can use the % conversion character in the international functions. However, you cannot mix the % conversion character with the *%digit$* conversion character in a single call.

International Environment

LC_NUMERIC If this environment is set and valid, `nl_scanf` uses the international language database named in the definition to determine radix character rules.

LANG If this environment variable is set and valid `nl_scanf` uses the international language database named in the definition to determine collation and character classification rules. If `LC_NUMERIC` is defined, its definition supersedes the definition of LANG.

Examples

The following shows an example of using the `nl_scanf` function:

```
nl_scanf("%2$s %1$d", integer, string)
```

If the input contains ‘‘ january 9 ’’, the `nl_scanf` function assigns 9 to *integer* and ‘‘january’’ to *string* .

Return Values

These functions return either the number of items matched or EOF on end of input, along with the number of missing or invalid data items.

See Also

intro(3int), setlocale(3), strtod(3), strtol(3), nl_printf(3int), printf(3int), scanf(3int), getc(3s), printf(3s), scanf(3s)
Guide to Developing International Software

Name

printf, fprintf, sprintf – print formatted output

Syntax

#include <stdio.h>

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

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

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

Description

The international functions `printf`, `fprintf`, and `sprintf` are similar to the `printf` standard I/O functions. The difference is that the international functions allow you to use the *%digit$* conversion character in place of the % character you use in the standard I/O functions. The *digit* is a decimal digit *n* from 1 to 9. The international functions apply conversions to the *n* th argument in the argument list, rather than to the next unused argument.

You can use the % conversion character in the international functions. However, you cannot mix the % conversion character with the *%digit$* conversion character in a single call.

You can indicate a field width or precision by an asterisk (*) instead of a digit string in format strings containing the % conversion character. If you use an asterisk, you can supply an integer *arg* that specifies the field width or precision. In format strings containing the *%digit$* conversion character, you can indicate field width or precision by the sequence **digit$*. You use a decimal digit from 1 to 9 to indicate which argument contains an integer that specifies the field width or precision.

The conversion characters and their meanings are identical to `printf`.

You must use each *digit* argument at least once.

In all cases, the radix character `printf` uses is defined by the last successful call to `setlocale` category `LC_NUMERIC`. If `setlocale` category `LC_NUMERIC` has not been called successfully or if the radix character is undefined, the radix character defaults to a period (.).

International Environment

LC_NUMERIC If this environment is set and valid, `printf` uses the international language database named in the definition to determine radix character rules.

LANG If this environment variable is set and valid `printf` uses the international language database named in the definition to determine collation and character classification rules. If `LC_NUMERIC` is defined, its definition supercedes the definition of LANG.

Examples

The following example illustrates using an argument to specify field width:

```
printf ("%1$d:%2$.*3$d:%4$.*3$d\n",
                            hour, min, precision, sec);
```

The format string **3$* indicates that the third argument, which is named precision, contains the integer field width specification.

To print the language independent date and time format use the following `printf` statement:

```
printf (format, weekday, month, day, hour, min);
```

For American use, *format* could be a pointer to the following string:

```
"%1$s,  %2$s %3$d, %4$d:%5$.2d\n"
```

This string gives the following date format:

```
Sunday, July 3, 10:02
```

For use in a German environment, *format* could be a pointer to the following string:

```
"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

This string gives the following date format:

```
Sonntag, 3. Juli, 10:02
```

Return Values

`printf` and `fprintf` return zero for success and EOF for failure. The `sprintf` subroutine returns its first argument for success and EOF for failure.

In the System V and POSIX environments, `printf`, `fprintf`, and `sprintf` return the number of characters transmitted for success. The `sprintf` function ignores the null terminator (\0) when calculating the number of characters transmitted. If an output error occurs, these routines return a negative value.

See Also

intro(3int), setlocale(3), scanf(3int), printf(3s), putc(3s), scanf(3s), stdio(3s)
Guide to Developing International Software

Name

scanf, fscanf, sscanf – convert formatted input

Syntax

#include <stdio.h>

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

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

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

Description

The international functions `scanf`, `fscanf`, and `sscanf` are similar to the `scanf` standard I/O functions. The difference is that the international functions allow you to use the *%digit$* conversion character in place of the *I%* character you use in the standard I/O functions. The *digit* is a decimal digit *n* from 1 to 9. The international functions apply conversions to the *n* th argument in the argument list, rather than to the next unused argument.

You can use % conversion character in the international functions. However, you cannot mix the % conversion character with the *%digit$* conversion character in a single call.

In all cases, `scanf` uses the radix character and collating sequence that is defined by the last successful call to `setlocale` category `LC_NUMERIC` or `LC_COLLATE`. If the radix or collating sequence is undefined, the `scanf` function uses the C locale definitions.

International Environment

LC_COLLATE Contains the user requirements for language, territory, and codeset for the character collation format. `LC_COLLATE` affects the behavior of regular expressions and the string collation functions in `scanf`. If `LC_COLLATE` is not defined in the current environment, `LANG` provides the necessary default.

LC_NUMERIC If this environment is set and valid, `scanf` uses the international language database named in the definition to determine radix character rules.

LANG If this environment variable is set and valid `scanf` uses the international language database named in the definition to determine collation and character classification rules. If `LC_NUMERIC` or `LC_COLLATE` is defined, their definitions supersede the definition of LANG.

Examples

The following shows an example of using the `scanf` function:

```
scanf("%2$s %1$d", integer, string)
```

If the input is " january 9 ", the `scanf` function assigns 9 to `integer` and "january" to `string`.

Return Values

The `scanf` function returns the number of successfully matched and assigned input fields. This number can be zero if the `scanf` function encounters invalid input characters, as specified by the conversion specification, before it can assign input characters.

If the input ends before the first conflict or conversion, `scanf` returns EOF. These functions return EOF on end of input and a short count for missing or invalid data items.

Environment

In POSIX mode, the **E**, **F**, and **X** formats are treated the same as the **e**, **f**, and **x** formats, respectively; otherwise, the upper-case formats expect double, double, and long arguments, respectively.

See Also

intro(3int), setlocale(3), strtod(3), strtol(3), printf(3int), getc(3s), printf(3s), scanf(3s)
Guide to Developing International Software

Name

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

Syntax

#include <stdio.h>
#include <varargs.h>

int vprintf (*format*, *ap*)
char **format*;
va list *ap*;

int vfprintf (*stream*, *format*, *ap*)
FILE **stream*;
char **format*;
va list *ap*;

int vsprintf (*s*, *format*, *ap*)
char **s*, **format*;
va list *ap*;

Description

The international functions `vprintf`, `vfprintf`, and `vsprintf` are similar to the `vprintf` standard I/O functions.

Likewise, the vprintf functions are similar to the printf functions except they are called with an argument list as defined by `varargs` instead of with a variable number of arguments.

The international functions allow you to use the *%digit$* conversion character in place of the % character you use in the standard I/O functions. The digit is a decimal digit *n* from 1 to 9. The international functions apply conversions to the *nth* argument in the argument list, rather than to the next unused argument.

You can use the % conversion character in the international functions. However, you cannot mix the % conversion character with the *%digit$* conversion character in a single call.

You can indicate a field width or precision by an asterisk (*) instead of a digit string in format strings containing the % conversion character. If you use an asterisk, you can supply an integer *arg* that specifies the field width or precision. In format strings containing the *%digit$* conversion character, you can indicate field width or precision by the sequence **digit$*. You use a decimal digit from 1 to 9 to indicate which argument contains an integer that specifies the field width or precision.

The conversion characters and their meanings are identical to `printf`.

You must use each digit argument at least once.

Examples

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

main()
{
char *function_name = "vpr";
char *arg1 = "hello world";
int arg2 = 2;
char *arg3 = "study";

char *i18nfmt = "%1$s %3$d\n";

test(function_name, i18nfmt, arg1, arg2, arg3);
}

test(va_alist)
va_dcl
{
va_list args;
char *fmt;
char string[1024];

va_start(args);

(void)printf("function %s: ", va_arg(args, char *));

fmt = va_arg(args, char *);

(void)vprintf(fmt, args);

va_end(args);
}
```

See Also

setlocale(3), scanf(3int), printf(3s), printf(3int), vprintf(3s), putc(3s), scanf(3s), stdio(3s), varargs(3)
Guide to Developing International Software

Kerberos Routines (3krb)

Insert tabbed divider here.
Then discard this sheet.

Name

intro – introduction to the Kerberos subroutines

Syntax

#include <krb.h>

#include <des.h>

cc [*options*] *files* **-lkrb -lknet**
-ldes -lacl [*libraries*]

Description

The Kerberos subroutines can provide for the authentication of and protection against the unauthorized modification of every message sent accross a TCP/IP network from one application to another. In addition, they provide a means to provide for the creation of access control lists (ACL) which an application can use with Kerberos authentication, to determine if another application is authorized to perform a particular action.

The `krb_svc_int(3krb)` routines are designed to initialize the Kerberos libraries so that the other Kerberos routines can function properly. The `krb_svc_init` routines are used to contact a Kerberos server to obtain a ticket-granting ticket that can be used by the `kerberos(3krb)`, `krb_sendmutual(3krb)`, and `krb_sendauth(3krb)` routines. They also initialize pieces of Kerberos library data. To use these routines, the libraries `libkrb.a`, `libknet.a`, and `libdes.a` must be linked with your application in the order listed.

The `kerberos(3krb)` routines `krb_mk_req` and `krb_rd_req` are designed to provide for the initial authentication of an application to another. They are designed to be used with applications that support "on-the-wire" protocols in which authentication information can be placed. The `kerberos(3krb)` routines `krb_mk_safe` and `krb_rd_safe` are designed to provide for the authentication of and protection against the modification of every message sent between two applications after the initial authentication message. To use these routines, the libraries `libkrb.a`, `libknet.a`, and `libdes.a` must be linked with your application in the order listed.

The `krb_sendmutual(3krb)` routines are designed to provide for the mutual authentication of two applications after the initial authentication of one application, X to another, Y. To provide mutual authentication, Y's identity is proven by the `krb_sendmutual` routines to X. To use these routines, the libraries `libkrb.a`, `libknet.a`, and `libdes.a` must be linked with your application in the order listed.

The `krb_sendauth(3krb)` routines are designed to provide both the initial authentication that `krb_mk_req` and `krb_rd_req` provide, as well as the mutual authentication of the `krb_sendmutal` routines. The `krb_sendauth` routines are designed to be used with applications that do not have room in the protocols they support for authentication information. To use these routines, the libraries `libkrb.a`, `libknet.a`, and `libdes.a` must be linked with your application in the order listed.

The `krb_get_lrealm(3krb)` routines are designed to provide information to the user about the Kerberos environment. To use these routines, the library `libkrb.a` must be linked with your application.

The `des_crypt(3krb)` routines are designed to provide support for the above routines with respect to Data Encryption Standard (DES) keys. The `des_quad_cksum` routine can be used to provide support for the authentication of and protection against the modification of every message sent between two applications after the initial authentication message. It is designed to be used only with applications that have room in their "on-the-wire" protocol for authentication information. To use these routines, the library `libdes.a` must be linked with your application.

The `krb_set_tkt_string(3krb)` routines are designed allow the user of the Kerberos libraries to modify some of the default settings of the Kerberos libraries. To use these routines, the library `libkrb.a` must be linked with your application.

The `acl_check(3krb)` routines are designed to provide for the creation and use of access control lists (ACL). After an application, X, correctly authenticates the identity of another, Y, the application X has the ability to assign access rights to Y, based on Y's identity. The routines above provide for the authentication of applications while the `acl_check(3krb)` routines provide the ability to store the access rights associated with each application. To use these routines, the library `libacl.a` must be linked with your application.

Files

`/usr/lib/libkrb.a`

`/usr/lib/libknet.a`

`/usr/lib/libdes.a`

`/usr/lib/libacl.a`

See Also

All the other Kerberos reference pages:

acl_check(3krb)
des_crypt(3krb)
kerberos(3krb)
krb_get_lrealm(3krb)
krb_sendauth(3krb)
krb_sendmutual(3krb)
krb_set_tkt_string(3krb)
krb_svc_init(3krb)
krb.conf(5krb)
krb_slaves(5krb)
krb_dbase(5krb)
ext_srvtab(8krb)
kdb_destroy(8krb)
kdb_edit(8krb)
kdb_init(8krb)
kdb_util(8krb)
kdestroy(8krb)

kerberos(8krb)
kinit(8krb)
klist(8krb)
kprop(8krb)
kpropd(8krb)
kstash(8krb)

acl_check (3krb)

Name

acl_check – Access control list (ACL) library routines.

Syntax

cc <files> -lacl -l krb

#include <krb.h>

acl_canonicalize_principal (*principal, buf*)
char **principal*;
char **buf*;

acl_check (*acl_file, principal*)
char **acl_file*;
char **principal*;

acl_exact_match (*acl_file, principal*)
char **acl_file*;
char **principal*;

acl_add (*acl_file, principal*)
char **acl_file*;
char **principal*;

acl_delete (*acl_file, principal*)
char **acl_file*;
char **principal*;

acl_initialize (*acl_file, mode*)
char **acl_file*;
int *mode*;

kname_parse (*primary_name, instance_name,*
realm_name, principal)
char **primary_name*;
char **instance_name*;
char **realm_name*;
char **principal*;

Arguments

principal The name of a principal. Principal names consist of from one to three fields. The first field must be included because it stores the primary name of the principal. The second field is not always required. It begins with a period (.), and stores the instance name of the principal. The third field is not always requred. It begins with an "at" sign (@), and stores the realm name of the principal. The principal name format can be expressed as:

```
name[.instance][@realm]
```

For example, all of the names below are legitimate principal names:

```
venus
venus.root
venus@dec.com
venus.@dec.com
venus.root@dec.com
```

buf Pointer to the buffer that stores the canonical form of a principal name. The canonical form is derived from the form of a principal name. Like a principal name, it includes a primary name in its first field. Unlike a principal name, it must include an instance name as its next field even if the instance name is blank. Also, unlike a principal name, it must contain a realm field. If a canonical name is derived from a principal name that has no realm field, the local realm returned by `krb_get_lrealm(3krb)` is used as the realm field in the canonical name. Of the above examples, only the last two are in canonical form.

acl_file The path name of the file in which the access control list (ACL) is stored.

mode If the ACL file, *acl_file*, does not currently exist when `acl_initialize` is called, the file *acl_file*, is created with read, write, and access mode bits set equal to *mode*.

primary_name
The primary name portion of *principal*, returned by `kname_parse`. ANAME_SZ bytes of storage space must be allocated for *primary_name*.

instance_name
The instance name of *principal*, returned by `kname_parse`. INST_SZ bytes of storage space must be allocated for *instance_name*.

realm_name
The realm name of *principal*, returned by `kname_parse`. REALM_SZ bytes of storage space must be allocated for *realm_name*.

Description

The routines of the `acl_check` library allow you to perform various administrative functions on an access control list (ACL). An ACL is a list of Kerberos principals in which each principal is represented by a text string. The routines of this library allow application programs to refer to named ACLs to test whether a principal is a member of an ACL, and to add or delete principals from the ACL file.

The routines of the `acl_check` library are:

acl_canonicalize_principal
Stores the canonical form of the principal name pointed to by *principal* in the buffer pointed to by *buf*. This buffer must contain enough space to store a full canonical principal name (MAX_PRINCIPAL_SIZE characters). No meaningful value is returned by `acl_canonicalize_principal`.

acl_check
Verifies that the principal name, *principal*, appears in the ACL file, *acl_file*. This routine returns a zero (0) if the principal does not appear in the ACL, or if there is an error condition. If the principal is a member of the ACL, a one (1) is returned. The `acl_check` routine always canonicalizes a principal before trying to find it in the ACL. `acl_check` will determine if there is an ACL entry in the *acl_file* which exactly matches principal, *principal*, or if *principal* matches an ACL entry which contains a wildcard. A wildcard appears in place of a field name in an ACL entry and is represented as an asterisk (*). A wildcard in a field name of an ACL entry allows the ACL entry to match a principal name

that contains anything in that particular field. For example, if there is an entry, `venus.*@dec.com` in the ACL, the principals, `venus.root@dec.com`, `venus.@dec.com`, and `venus.planet@dec.com` would be included in the ACL. The use of wildcards is limited, for they may be used in only the three following configurations in an ACL file:

```
name.*@realm
*.*@realm
*.*@*
```

acl_exact_match
Verifies that principal name, *principal*, appears in the ACL file, `acl_file`. This routine returns a zero (0) if the principal does not appear in the ACL, or if any error occurs. If the principal is a member of the ACL, `acl_exact_match` returns a non-zero. The `acl_exact_match` routine does not canonicalize a principal before the ACL checks are made, and it does not support wildcards. Only an exact match is acceptable. So, for example, if there is an entry, `venus.*@dec.com` in the ACL, only the principal `venus.*@dec.com` would match the ACL entry. This routine makes it easy to find ACL entries with wildcards.

acl_add Adds the principal name, *principal*, to the ACL file, *acl_file*. This routine returns a zero (0) if it successfully adds the principal to the ACL. Otherwise, if there was an internal error, or if the principal is already in the ACL, the `acl_add` routine returns a non-zero value. The `acl_add` routine canonicalizes a principal, but treats wildcards literally.

acl_delete
Deletes the principal, *principal*, from the ACL file, *acl_file*. The routine returns a zero (0) if it successfully deletes the principal from the ACL. Otherwise, if there was an internal error or if the principal is not in the ACL, the `acl_delete` routine returns a non-zero value. The `acl_delete` routine canonicalizes a principal, but treats wildcards literally.

acl_initialize
Initializes the ACL file, *acl_file*. If the named *acl_file* does not exist, `acl_initialize` creates one with the permissions specified by the *mode* argument. If the ACL exists, `acl_initialize` removes all previously stored principal members of the list. This routine returns a zero (0) if successful or a nonzero if it fails.

kname_parse
parses the principal name, *principal*, and stores the primary name of the principal in *principal_name*, the instance name of the principal in *instance_name*, and the realm name of the principal in *realm_name*. `kname_parse` returns KNAME_FMT if the principal name is incorrectly formatted or if it is too long to be a principal name. It returns KSUCCESS if the parsing of the principal name succeeded.

See Also

kerberos(3krb), krb_get_lrealm(3krb)

des_crypt(3krb)

Name

des_crypt – Data Encryption Standard (DES) encryption library routines.

Syntax

#include <des.h>

int des_string_to_key (*str, key*)
char **str*;
C_Block **key*;

int des_is_weak_key (*key*)
C_Block *key*;

unsigned long des_quad_cksum (*input, output, length, iterations, seed*)
unsigned char **input*;
unsigned long **output*;
long *length*;
int *iterations*;
C_Block **seed*;

int des_key_sched (*key, schedule*)
C_Block *key*;
Key_schedule *schedule*;

Arguments

key For `des_string_to_key`, *key* is a pointer to a `C_Block` of 8-byte length. For `des_quad_cksum`, `des_is_weak_key`, and `des_key_sched`, *key* is a pointer to a DES key.

str A string that is converted to an 8-byte DES key.

input Pointer to a block of data to which a quadratic checksum algorithm is applied.

output Pointer to a pre-allocated buffer that will contain the complete output from the quadratic checksum algorithm. For each iteration of the quadratic checksum applied to the input, eight bytes (two longwords) of data are generated.

length Length of the data to which the quadratic checksum algorithm will be applied. If input contains more than *length* bytes of data, then the quadratic checksum will only be applied to *length* bytes of input.

iterations The number of iterations of the `des_quad_cksum` algorithm to apply to *input*. If output is NULL, then one iteration of the algorithm will be applied to *input*, no matter what the value of *iterations* is. The maximum number of iterations is four.

seed An 8-byte quantity used as a seed to the *input* of the `des_quad_cksum` algorithm.

schedule A representation of a DES key in a form more easily used with encryption algorithms. It is used as input to the `krb_sendmutual` routines.

Description

The des_crypt routines are designed to provide the cryptographic routines which are used to support authentication. Specifically, des_quad_cksum and des_key_sched are designed to be used with the DES key which is shared between one Kerberos principal and its authenticated peer to provide an easy authentication method after the initial Kerberos authentication pass. des_string_to_key and des_is_weak_key are designed to enable the input and inspection of a key by a user before that key is used with the Kerberos authentication routines. The des_crypt routines are not designed for general encryption.

The library makes extensive use of the locally defined data types C_Block and Key_schedule. The C_Block struct is an 8-byte block used by the various routines of the des_crypt library as the fundamental unit for DES data and keys.

Routines

string_to_key

Converts a null-terminated string of arbitrary length to an 8-byte, odd-byte-parity DES key. The *str* argument is a pointer to the character string to be converted and *key* points to a C_Block supplied by the caller to receive the generated key. The one-way function used to convert the string to a key makes it very difficult for anyone to reconstruct the string from the key. No meaningful value is returned.

des_is_weak_key

des_is_weak_key checks a new key input by a user to determine if it belongs to the well known set of DES keys which do not provide good cryptographic behavior. If a key passes the inspection of des_is_weak_key, then it can be used with the des_quad_cksum routine. The input is a DES key and the output is equal to 1 if the key is not a safe key to use; it is equal to 0 if it is safe to use.

des_quad_cksum

Produces a checksum by chaining quadratic operations on cleartext data. des_quad_cksum can be used to produce a normal quadratic checksum and, if used with the DES key shared between two authenticated Kerberos principals, it can also provide for the integrity and authentication protection of data sent from one principal to another.

Input of *length* bytes are run through the des_quad_cksum routine *iterations* times to produce *output*. If *output* is NULL, one iteration is performed and *output* is not affected. If *output* is not NULL, the quadratic checksum algorithm will be performed *iterations* times on input, placing eight bytes (two longwords) of result in *output* for each iteration. At all times, the low-order bits of the last quadratic checksum algorithm pass are returned by des_quad_cksum.

The quadratic checksum algorithm performs a checksum on a few bytes of data and feeds the result into the algorithm as an addition input to the checksum on the next few bytes. The seed serves as the additional input for the first checksum operation and, therefore, the final checksum that results depends upon the seed input into the algorithm. If the DES key shared between two Kerberos principals is used as the initial seed, then since the checksum that results depends upon the seed, the ability to produce the checksum proves identity and authentication. Also, since the message cannot be altered without knowledge of the seed, it also provides for data integrity.

des_key_sched

`des_key_sched` is used to convert the key input into a new format that can be used readily with encryption functions. The result, schedule, can be used with the `krb_sendmutual` functions to enable mutual authentication of two Kerberos principals.

0 is returned from `des_key_sched` if sucessful.

-1 is returned if the each byte of the key does not have odd parity.

-2 is returned if the key is a weak key as defined by `des_is_weak_key`.

Name

kerberos – Kerberos authentication library routines

Syntax

```
#include <des.h>
#include <krb.h>

int krb_mk_req(tkt_authen_out, f_service, f_instance,
               f_realm, checksum)
KTEXT       tkt_authen_out;
char        *f_service;
char        *f_instance;
char        *f_realm;
u_long      checksum;

int krb_rd_req(tkt_authen_in, l_service, l_instance,
               f_hostaddr, ad, srvtab_file)
KTEXT       tkt_authen_in;
char        *l_service;
char        *l_instance;
u_long      f_hostaddr;
AUTH_DAT    *ad;
char        *srvtab_file;

int krb_get_cred(f_service, f_instance,
                 f_realm, cred)
char        *f_service;
char        *f_instance;
char        *f_realm;
CREDENTIALS     *cred;

long   krb_mk_safe(in, out, in_length, key,
                   l_addr, f_addr)
u_char                 *in;
u_char                 *out;
u_long                 in_length;
C_Block                *key;
struct sockaddr_in     *l_addr;
struct sockaddr_in     *f_addr;

long   krb_rd_safe(in, in_length, key, f_addr,
                   l_addr, msg_data)
u_char                 *in;
u_long                 in_length;
C_Block                *key;
struct sockaddr_in     *f_addr;
struct sockaddr_in     *l_addr;
MSG_DAT                *msg_data;
```

Arguments

f_service Character pointer to the primary name of the foreign principal. The local principal is the principal that calls the routines listed above. The local principal tries to communicate with the foreign principal.

f_instance Character pointer to the instance name of the foreign principal.

f_realm Character pointer to the realm name of the foreign principal.

l_service Character pointer to the primary name of the local principal.

l_instance Character pointer to the instance name of the local principal.

tkt_authen_out
Pointer to the text structure in which the Kerberos library routines build the ticket-authenticator pair. This structure is designed to be sent to the foreign principal to authenticate the local principal's identity to the foreign principal. Storage must be allocated for *tkt_authen_out*.

tkt_authen_in
Pointer to the ticket-authenticator pair that the Kerberos library uses to authenticate the foreign principal to the local principal. The data in this structure must have been generated by a call to `krb_mk_req` by the foreign principal and transmitted by the foreign principal to the local principal.

checksum The *checksum* parameter is input to `krb_mk_req`. It is packaged with the ticket-authenticator pair that is sent to the foreign principal. The *checksum* serves as a secret piece of data that can be known only to the foreign principal if the foreign principal is authenticated as the foreign principal. It is used to facilitate mutual authentication with `krb_sendmutual` and `krb_recvmutual`. See `krb_sendmutual(3krb)` for information about these two routines.

f_hostaddr Address of the machine from which the foreign principal sent the *tkt_authen_in* data.

f_addr Address of the socket that the foreign principal is using to communicate with the local principal.

l_addr Address of the socket that the local principal is using to communicate with the foreign principal.

ad Pointer to the AUTH_DAT structure that describes the authentication association between the local and foreign principals. The *ad* structure is output from `krb_rd_req`. You must allocate space for the *ad* structure.

srvtab_file The path name of the file that contains the key of the principal obtaining a ticket. If this value is set equal to a string of zero length, `srvtab_file[0]`='\0', the default service table (srvtab) file is used. If this value is set equal to the NULL pointer, then the key of the service is not read from the srvtab file, but is read from storage space internal to the libraries. The *srvtab_file* parameter cannot be set equal to the NULL string on the first call to `krb_rd_req`. The default srvtab file value is set to `/etc/srvtab`, although this value can be changed by a call to the `krb_set_srvtab_string` function. (See the `krb_set_tkt_string(3krb)` reference page).

key Pointer to the C_Block input to `krb_mk_safe` and `krb_rd_safe`. It contains a Data Encryption Standard (DES) key. The key that is usually used is the session key between the local and foreign principal.

cred A pointer to a credentials structure that is allocated by the caller of `krb_get_cred` and filled with data by `krb_get_cred`. The credentials structure includes the ticket that the local principal uses to authenticate the foreign principal. It also includes other authentication information associated with the foreign principal.

in Character pointer to the user data that must be included in a safe message.

out Character pointer to the safe message output by `krb_mk_safe`. The *in* parameter may not overlap with *out*.

in_length Length of the user data, *in*.

msg_data The *msg_data* parameter is a pointer to a MSG_DAT structure which must be allocated by the caller of `krb_rd_safe` and which is filled by `krb_rd_safe` with information about the safe message. A pointer to the user data sent within the safe message is also included in *msg_data*.

Description

The `krb_mk_req` calls are designed to be used by two principals that are attempting to authenticate themselves for the first time as well as by two principals that have authenticated once, but wish to authenticate all data passed between them.

The `krb_mk_req` and `krb_rd_req` routines are designed to be used by applications that communicate over a network, require the authentication of both parties across the communication path, and support "on-the-wire" protocols in which authentication data can be placed. These routines perform only the authentication of the first message sent between such applications. `krb_mk_req` creates a ticket-authenticator pair that can be included in the "on-the-wire" protocol of an application, and `krb_rd_req` reads the ticket-authenticator pair.

The `krb_mk_safe` and `krb_rd_safe` routines are used by applications that require that every message passed between them be authenticated and free from unauthorized modifications, and whose "on-the-wire" protocol has no room for authentication data. These routines only provide for the authentication and integrity protection of a message if the first authenticated message has already been sent by the `krb_mk_req/krb_rd_req` pair or the `krb_sendauth/krb_recvauth` pair. See `krb_sendauth(3krb)` for more information about the latter pair.

The `krb_mk_safe` routine encapsulates user data inside the `krb_mk_safe` "on-the-wire" message authentication protocol. `krb_rd_safe` can interpret the message authentication protocol and the message, and return the data encapsulated by `krb_mk_safe`. Since any application which is modified to use `krb_mk_safe` or krb_rd_safe must encapsulate its "on-the-wire" protocol within the "on-the-wire" protocol of krb_mk_safe, the application must develop a method of distinguishing between the old and new "on-the-wire" protocols.

The des_quad_cksum routine (see des_crypt (3krb)) can be used to provide some of the guarantees of the krb_mk_safe and krb_rd_safe routines without encapsulating the protocol of the application.

The routines of this library make extensive use of the following locally defined data types: KTEXT, AUTH_DAT, CREDENTIALS, C_Block, and MSG_DAT. For specific information on the definitions of these data types, see the des.h and krb.h files.

Routines and Structures

krb_mk_req

Used to produce the data necessary to authenticate a principal "A" to a principal "B". It takes as input a checksum and the primary name, instance name, and realm name of the service to which the principal "A" is attempting to authenticate itself. krb_mk_req outputs a text structure in which the ticket to communicate with principal "B" and an authenticator have been combined to form a ticket-authenticator pair.

The application "A" must pass the ticket-authenticator pair to the principal "B" where it can be read by krb_rd_req. Once the ticket-authenticator pair has been read and verified, "A" has been authenticated to "B". Unless an attacker possesses the session key contained in the ticket, the attacker will be unable to modify or replay the ticket-authenticator pair.

The checksum can be used with krb_sendmutual and krb_recvmutual to provide for the authentication of "B" to "A" after krb_rd_req authenticates "A" to "B". Although the checksum value can be any value known only to "A", it is recommended that the checksum value used differ every time krb_mk_req is called. The following is a list of the return values from krb_mk_req and, if they are error codes, their possible cause:

KFAILURE	/etc/krb.conf file (see krb.conf (5krb)) cannot be opened, or it is not properly formed.
NO_TKT_FIL	The ticket file does not exist.
TKT_FIL_ACC	The ticket file cannot be opened or the ticket file cannot be accessed.
TKT_FIL_LCK	The ticket file could not be locked for access.
TKT_FIL_FMT	The ticket file format is incorrect.
AD_NOTGT	There is no ticket-granting ticket in the ticket file that can be used to ask for a ticket to communicate with the foreign principal.
SKDC_CANT	A Kerberos server must be contacted so that krb_mk_req can perform its function, but the attempt cannot be made because a socket cannot be opened or bound, or because there is no Kerberos server listed in /etc/krb.conf.
SKDC_RETRY	A Kerberos server needs to be contacted, but none responded even after several attempts.

INTK_PROT	Kerberos protocol error.
KSUCCESS	All went well.

krb_rd_req

This routine is used to read the authentication data produced by principal "A" with `krb_mk_req` and sent by "A" to principal "B". It takes as input the primary name and instance name of the local principal "B", as well as the authentication data sent to "B", the address of the machine from which "A" sent the ticket-authenticator pair, and the name of the file in which to find the key of the local principal. If the authentication attempt is successful, `krb_rd_req` will fill the *ad* structure with data about the authenticated association between "A" and "B".

The `krb_rd_req` routine returns zero (RD_AP_OK) upon successful authentication. If a packet was forged, modified, or replayed, then authentication fails.

The following is a list of the error values returned from `krb_mk_req` and their possible causes:

RD_AP_VERSION	The versions of Kerberos used by the caller of `krb_mk_req` is incompatible with the `krb_rd_req` version.
RD_AP_MSG_TYPE	The ticket-authenticator pair given to `krb_rd_req` was not actually a ticket-authenticator pair.
RD_AP_UNDEC	The ticket was indecipherable. This error can be caused by a forged or a modified message.
RD_AP_INCON	The message given to `krb_rd_req` contains an internal inconsistency. This could occur if the ticket in the ticket-authenticator pair does not match the authenticator.
RD_AP_BADD	The ticket-authenticator pair cannot be used from the address, *f_hostaddr*.
RD_AP_TIME	The authenticator in the ticket-authenticator pair is too old to be used to authenticate the foreign principal.
RD_AP_NYV	The time at which the ticket of the ticket-authenticator pair was created, is too far ahead of the time of the local host of the local principal.
RD_AP_EXP	The ticket is too old to be used.

krb_get_cred

Searches the caller's ticket file for the authentication information associated with the principal specified by the *f_service*, *f_instance*, and *f_realm*. If `krb_get_cred` finds information in the ticket file, it fills a credentials structure with the information and returns the status, GC_OK.

The following is a list of the error values returned from `krb_mk_req` and their possible causes:

NO_TKT_FIL	The ticket file does not exist.

TKT_FIL_ACC	The ticket file cannot be opened or the ticket file cannot be accessed.
TKT_FIL_LCK	The ticket file could not be locked for access.
TKT_FIL_FMT	The ticket file format is incorrect.
GC_NOTKT	Information concerning the principal does not exist in the ticket file.

krb_mk_safe

Creates an authenticated but unencrypted message from text pointed to by *in*, of a length indicated by *in_length*. The routine uses the private session key (**key*) to seed the checksum algorithm, `des_quad_cksum`, that it uses as part of the authentication process. (For more information about `des_quad_cksum`, see the `des_crypt(3krb)` reference page.) The `krb_mk_safe` routine also uses the arguments *l_addr* and *f_addr* for authentication purposes.

A safe message does not provide privacy, but does provide protection against modifications in addition to providing authentication. The encapsulated message and header produced by `krb_mk_safe` are placed in the area pointed to by *out*. The routine returns the length of the output or a negative one (–1), indicating an error.

krb_rd_safe

Authenticates a received `krb_mk_safe` message and writes the appropriate fields in the message data structure **MSG_DAT**. The argument *in* points to the beginning of the received message. The argument *in_length* specifies the length of the message. The `krb_rd_safe` routine uses the private session key (**key*) to seed the `des_quad_cksum` routine (see the `des_crypt(3krb)` reference page) as part of its authentication process. The routine fills in the following **MSG_DAT** fields:

MSG_DAT Field	Description
`app_data`	Pointer to the application data
`app_length`	Length of the `app_data`
`time_sec`	Timestamp of the message in seconds
`time_5ms`	Timestamp of the message in 5-millisecond units
`swap`	A 1 if the byte order of the receiver is different from that of the sender

Note that the application must still determine if it is appropriate to byte-swap application data; the Kerberos protocol fields are already taken care of.

The `krb_rd_safe` routine returns RD_AP_OK if the message, *in*, is authenticated and has not been modified when it was sent between the foreign and the local principal. It is up to the caller to check the time sequence of messages and to check against recently replayed messages. The following is a list of the error values returned by `krb_rd_req` and their possible causes:

-1 A system call used by `krb_rd_safe` returned an error.

RD_AP_VERSION

The Kerberos version of the `krb_mk_safe` code that generated message, *in*, is not supported by the `krb_rd_safe` version used.

RD_AP_MSG_TYPE
: The message, *in*, is not really a message produced by `krb_mk_safe`.

RD_AP_MODIFIED
: The address of the machine from which *in* was sent does not match the address of the machine on which the `krb_mk_safe` message, *in*, was generated, or
The message was modified when it was sent from the foreign to the local principal, or
The message, *in*, is too small to be the message produced by `krb_mk_req`.

RD_AP_TIME
: The difference between the time at which the message, *in*, was produced by `krb_mk_req` and the time at which it was read by `krb_rd_req` is too large. The time difference must be within five minutes.

Restrictions

The caller of the functions, `krb_rd_req` and `krb_rd_safe`, must check the time order of messages and protect against replay attempts.

Files

```
/usr/include/krb.h

/usr/lib/libkrb.a

/usr/include/des.h

/usr/lib/libdes.a

/etc/srvtab
```

See Also

des_crypt(3krb), krb_sendmutual(3krb), krb_sendauth(3krb), krb_svc_init(3krb), krb_set_tkt_string(3krb), krb.conf(5krb)

krb_get_lrealm (3krb)

Name

krb_get_lrealm – Host/realm identification routines.

Syntax

#include <krb.h>
#include <des.h>

krb_get_lrealm (*realm, n*)
char **realm*;
int *n*;

char *krb_get_phost (*alias*)
char **alias*;

Arguments

alias Identifies a host whose name is to be converted to an instance name. The *alias* string is overwritten with the instance name. The *alias* string must be stored in a buffer of at least INST_SZ characters.

realm Identifies a specific realm.

n Specifies a specific position in a series of Kerberos hosts; must be set to 0.

Description

The routines of `krb_get_lrealm` allow an application to obtain information on host/realm relationships in a Kerberos network. The routines of this library are:

krb_get_phost
Converts the hostname pointed to by *alias*, which can be either an official name or an alias, into the instance name to be used in obtaining Kerberos tickets.

krb_get_lrealm
Initializes *realm* with the *n*th realm of the local host. The argument *realm* should be large enough to contain the maximum realm name determined by the constant REALM_SZ. The local realm name is stored in the `/etc/krb.conf` file. See the `krb.conf(5krb)` reference page.

Files

`/etc/krb.conf`

See Also

kerberos(3krb), krb.conf(5krb)

Name

krb_sendauth, krb_recvauth – Kerberos authentication library routines.

Syntax

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

int krb_sendauth (options, fd, tkt_authen, f_service,
                  f_inst, f_realm, checksum, msg_data,
                  cred, schedule, l_addr, f_addr,
                  version_in)
long                 options;
int                  fd;
KTEXT                tkt_authen;
char                 *f_service;
char                 *f_instance;
char                 *f_realm;
u_long               checksum;
MSG_DAT              *msg_data;
CREDENTIALS          *cred;
Key_schedule         schedule;
struct sockaddr_in   *l_addr;
struct sockaddr_in   *f_addr;
char                 *version_in;

int krb_recvauth (options, fd, tkt_authen_out, l_service,
                  l_instance, f_addr, l_addr, ad,
                  srvtab_file, schedule, version_out)
long                 options;
int                  fd;
KTEXT                tkt_authen_out;
char                 *l_service;
char                 *l_instance;
struct sockaddr_in   *f_addr;
struct sockaddr_in   *l_addr;
AUTH_DAT             *ad;
char                 *srvtab_file;
Key_schedule         schedule;
char                 *version_out;
```

Arguments

options Defined in `/usr/include/krb.h`. To specify multiple options, construct the *options* argument as a bitwise-OR of the desired options. The options are as follows:

KOPT_DONT_MK_REQ
: `krb_sendauth` will not use the `krb_mk_req` function (see `kerberos(3krb)`) to produce the ticket-authenticator pair, *authen_tkt*. Instead, the ticket-authenticator pair is read from the argument, *tkt_authen*.

KOPT_DONT_CANON
: `krb_sendauth` will not convert the instance name, *f_instance*, to canonical form. If KOPT_DONT_CANON is not set, the instance name used is the output from `krb_get_phost` (see `krb_get_lrealm(3krb)`) with argument *f_instance* as input.

KOPT_DO_MUTUAL
: `krb_sendauth` and `krb_recvauth` provide authentication on both ends of the network connection. Otherwise, the caller of `krb_sendauth` is authenticated to the caller of `krb_recvauth`, but the caller of `krb_recvauth` is not authenticated to the caller of `krb_sendauth`. For mutual authentication to occur, both `krb_sendauth` and `krb_recvauth` must be called with this option set.

f_service Character pointer to the primary name of the foreign principal. The local principal is the principal that calls the above routines. The foreign principal is the principal with which the local principal is attempting to communicate. If KOPT_DONT_MK_REQ is set and KOPT_DO_MUTUAL is not, then *f_service* should be set equal to the NULL pointer.

f_instance
: Character pointer to the instance name of the foreign principal. If KOPT_DONT_MK_REQ is set and KOPT_DO_MUTUAL is not, then *f_instance* should be set equal to the NULL pointer.

f_realm Character pointer to the realm name of the foreign principal. If the *f_realm* parameter is set equal to the NULL pointer, then the local realm is used as the *f_realm*. If KOPT_DONT_MK_REQ is set and KOPT_DO_MUTUAL is not, then *f_service* should be set equal to the NULL pointer.

l_service Character pointer to the primary name of the local principal.

l_instance
: Character pointer to the instance name of the local principal.

fd The file descriptor used to send data to the foreign principal, or the file descriptor from which data from the foreign principal can be read. In either case, the file descriptor must be associated with a socket that uses blocking I/O.

tkt_authen
: Pointer to the text structure in which the Kerberos library routines build the ticket-authenticator pair. This structure is designed to be included within the `krb_sendauth` message sent to the foreign principal to authenticate the local principal's identity to the foreign principal. This structure can be either input to `krb_sendauth` or output from `krb_sendauth` depending on whether KOPT_DONT_MK_REQ is set or not set. In either case, storage must be allocated for *tkt_authen.*

tkt_authen_out
: Pointer to the ticket-authenticator pair that `krb_recvauth` reads from within the `krb_sendauth` message. The `krb_sendauth` message is sent by `krb_sendauth` to the local principal to authenticate the foreign principal to the local principal. Storage must be allocated for *tkt_authen_out.*

checksum

Input to `krb_sendauth`; *checksum* is packaged in the `krb_sendauth` message that is sent to the foreign principal. It serves as a secret piece of data that can only be known to the foreign principal if the foreign principal is authenticated as the foreign principal. It is used to facilitate mutual authentication, so if the KOPT_DO_MUTUAL is not set, the value of this argument is inconsequential. If both KOPT_DONT_MK_REQ and KOPT_DO_MUTUAL are set, then the *checksum* parameter must be equal to the checksum value used by `krb_mk_req` in the creation of the ticket-authenticator pair, *authen_tkt*.

msg_data

Pointer to a structure which is filled with the mutual authentication message sent by `krb_recvauth` and interpreted by `krb_sendauth`. The message sent from `krb_sendauth` to `krb_recvauth`, the message that includes the ticket-authenticator pair, authenticates only the caller of `krb_sendauth` to the caller of `krb_recvauth`. An additional message, the one returned by `krb_sendauth` inside *msg_data*, must be sent by `krb_recvauth` and interpreted by `krb_sendauth` in order to authenticate the caller of `krb_recvauth` to the caller of `krb_sendauth`. If the KOPT_DO_MUTUAL option is set, space must be allocated for the *msg_data* structure. Otherwise, since no message will be sent from `krb_recvauth` to `krb_sendauth`, the *msg_data* parameter should be set equivalent to the NULL pointer.

cred a pointer to a credentials structure that is output from `krb_sendauth`. The credentials structure includes the ticket that the local principal uses to authenticate to the foreign principal as well as other authentication information associated with the foreign principal. If the KOPT_DO_MUTUAL option is set, space must be allocated for the *cred* structure and the *cred* structure will be filled in by `krb_sendauth`. Otherwise, the *cred* structure will not be filled in by `krb_sendauth`, so the *cred* parameter should be set equivalent to the NULL pointer.

schedule a key schedule, derived from the session key between the local and foreign principals, that is output from `krb_sendauth` and `krb_recvauth`. If the KOPT_DO_MUTUAL option is set, the key schedule will be filled in; otherwise, the key schedule will not be filled. In any case, space must be allocated for the key schedule.

f_addr the address of the socket that the foreign principal is using to communicate with the local principal. If the KOPT_DO_MUTUAL option is not set on a call to `krb_sendauth`, then the *f_addr* parameter should be set equivalent to the NULL pointer. *f_addr* should never be set to NULL on a call to `krb_recvauth`.

l_addr the address of the socket that the local principal is using to communicate with the foreign principal. If the KOPT_DO_MUTUAL option is not set, the *l_addr* parameter should be set equivalent to the NULL pointer.

ad a pointer to the AUTH_DAT structure that describes the authentication association between the local and foreign principals. Since it is output from `krb_recvauth`, space for the *ad* structure must be allocated.

srvtab_file
path name of the file that contains the key of the principal obtaining a ticket. If this value is set equal to a string of zero length, `srvtab_file[0]`='\0', the default service table file (srvtab) value is used. If this value is set equal to the NULL pointer, then the key of the service is not read from the srvtab file, but is read from storage space internal to the libraries. The *srvtab_file* parameter cannot be set to the NULL string on the first call to `krb_sendauth`. The default srvtab file value is set to `/etc/srvtab` although this value can be changed by a call to the `krb_set_srvtab_string` function (see `krb_set_tkt_string(3krb)`).

version_in
An application-specific version string input to `krb_sendauth`. This argument allows the caller of `krb_sendauth` to pass an application-specific version string, within the `krb_sendauth` message format, that the caller of `krb_recvauth` can use to match against its own version string. The version string can be up to KRB_SENDAUTH_VLEN characters long and, in addition, it can be set equal to the NULL string.

version_out
An application-specific version string output from `krb_recvauth`. This argument allows the caller of `krb_recvauth` to receive the application-specific version string included in the `krb_sendauth` message that was sent by the foreign principal. The version string can be up to KRB_SENDAUTH_VLEN characters long.

Description

The `krb_sendauth(3krb)` routines are designed to be used by applications that communicate over a network, require the authentication of both parties accross the communications path, and which support "on-the-wire" protocols that have no room for authentication information. The `krb_sendauth(3krb)` routines are designed to perform only the authentication of the first message sent between such applications. Therefore, the `krb_sendauth(3krb)` routines should be used before any other communication occurs between the authenticating principals.

After the communications channel between the applications has been established, but before any communication takes place, and before the "on-the-wire" protocol of the application comes into effect, `krb_sendauth` creates a message which can authenticate the caller of `krb_sendauth`, "A", to the caller of `krb_recvauth`, "B". `krb_sendauth` then sends the message to "B" where it is read from the communications channel by `krb_recvauth`.

Next, `krb_recvauth` attempts to authenticate "A" by producing a response to "A" which, depending upon the value of KOPT_DO_MUTUAL and the success of the authentication of "A" by `krb_recvauth`, will contain either an error code, a code indicating success, or a mutual authentication message. `krb_recvauth` sends the response and returns to "B". `krb_sendauth` receives the message from "B", tries to authenticate "B" if KOPT_DO_MUTUAL is set, and then returns to "A".

Since the authentication information is sent between the applications before the "on-the-wire" protocol of the application comes into effect, the application must develop some method of distinguishing between the new authenticated initial message exchange and an old unauthenticated initial message exchange.

The krb_sendauth(3krb) routines make extensive use of the locally defined data types KTEXT, MSG_DAT, CREDENTIALS, and Key_schedule. For specific information on the definitions of these data types, see the des.h and krb.h files.

The routines found in the krb_sendauth(3krb) library are krb_sendauth and krb_recvauth:

krb_sendauth

The krb_sendauth function is designed to authenticate a local principal, "A", to the principal specified by the *f_service*, *f_instance*, and *f_realm* parameters, "B", and to allow the authentication of "B" to "A" as well. krb_sendauth uses file descriptor *fd*, to send the authentication message that will authenticate "A" to principal "B". It returns, in the *tkt_authen* parameter, the ticket-authenticator pair used to authenticate "A" to "B". The *version_in* parameter contains an application-specific version string which is transmitted to "B" along with the authentication message.

If mutual authentication is selected as an option, the file descriptor, *fd* will be used to receive a mutual authentication message from "B". To allow the mutual authentication to take place, *l_addr* and *f_addr* must be set equal to the address of the sockets which the local and foreign principals use to communicate. A value known only to "A" must be input to krb_sendauth as the *checksum* parameter. As the result of mutual authentication, *cred* will be filled with data describing the authentication information associated with "B", *schedule* will be set equal to the key_schedule of the session key between "A" and "B", and *msg_data* will be set equal to the mutual authentication message sent from "B" to "A".

fd must be a file descriptor associated with a blocking socket. Otherwise, krb_sendauth will not function correctly.

If "A" has been correctly authenticated to "B" and mutual authentication was not chosen as an option, or if "A" has been correctly authenticated to "B", and "B" correctly authenticated to "A" and mutual authentication was chosen as an option, then KSUCCESS is returned by krb_sendauth.

The following is a list of most of the error values from krb_sendauth. Since krb_sendauth calls other section 3 Kerberos routines (3krb) to perform its function, some of the error codes are references to the error codes of other functions:

SENDAUTH_OPNOTSUP
: The *options* bits sent to krb_sendauth contain a bit which is set, but does not correspond to an option.

SENDAUTH_WR
: krb_sendauth could not write the authentication message to "B" using *fd*.

KFAILURE
: The /etc/krb.conf file cannot be opened, or
The /etc/krb.conf file (see krb.conf(5krb)) is not formed properly, or
An authentication message was sent from "A" to "B", but "B" could not successfully identify "A", or
A mutual authentication message was sent from "B" to "A", but "A" could not successfully identify "B".

-1
: Negative one is returned if each byte of the session key does not have odd parity.

-2	Negative two is returned if the session key is a weak key as defined by `des_is_weak_key` (see `des_crypt(3krb)`).
NO_TKT_FIL	The ticket file does not exist.
TKT_FIL_ACC	The ticket file cannot be opened or the ticket file cannot be accessed.
TKT_FIL_LCK	The ticket file could not be locked for access.
TKT_FIL_FMT	The ticket file format is incorrect.
AD_NOTGT	There is no ticket-granting-ticket in the ticket file that can be used to ask for a ticket to communicate with the foreign principal.
SKDC_CANT	A Kerberos server must be contacted in order for `krb_sendauth` to perform its function, but the attempt cannot be made because a socket cannot be opened or bound, or there is no Kerberos server listed in `/etc/krb.conf`.
SKDC_RETRY	A Kerberos server needs to be contacted, but none responded even after several retries.
INTK_PROT	Kerberos protocol error.
GC_NOTKT	Information concerning the foreign principal does not exist in the ticket file.
RECVMUT_OPNOTSUP	The *options* bits sent to `krb_recvmutal` (see `krb_sendmutual(3krb)`) contain a bit which is set, but does not correspond to an option.
RECVMUT_RD	If the message cannot be read from the file descriptor *fd*, SENDMUT_RD is returned.
RD_AP_VERSION	If the Kerberos version used to create the mutual authentication message is not supported by `krb_recvmutual`, then RD_AP_VERSION is returned.
RD_AP_MSG_TYPE	If the message read from the file descriptor, *fd*, is not a mutual authentcation message, RD_AP_MSG_TYPE is returned.
RD_AP_MODIFIED	If the mutual authentication message has been modified between the "B" and "A" or it was in some way incorrectly produced, RD_AP_MODIFIED is returned.
RD_AP_TIME	Returned if the mutual authentication message is too old.

krb_recvauth

The `krb_recvauth` function is designed to wait for a message from `krb_sendauth` on the file descriptor *fd*, receive the message and attempt to authenticate the foreign principal, "A", to the local principal determined by the *l_service* and *l_instance* parameters. The *srvtab_file* must contain the private key of principal "B". The *tkt_authen_out* parameter is filled with the ticket-authenticator pair sent within the `krb_sendauth` message received by "B" from "A". *ad* is filled with information that describes the authentication association between "A" and "B". *version_out* is filled with the application version string included in the `krb_sendauth` message.

If mutual authentication is selected as an option, the file descriptor *fd*, will be used to send a mutual authentication message to "A". To allow the mutual authentication to take place, *l_addr* and *f_addr* must be set equal to the address of the sockets that the local and foreign principals are using to communicate. As the result of mutual authentication, *schedule* will be set equal to the key_schedule of the session key between "A" and "B".

fd must be a file descriptor that is associated with a blocking socket. Otherwise, `krb_recvauth` will not function correctly.

If "A" has been correctly authenticated to "B" and mutual authentication was not chosen as an option, or if mutual authentication is an option and "A" has been correctly authenticated to "B" and "B" has sent a mutual authentication message to "B", then KSUCCESS is returned by `krb_recvauth`.

The following is a list of most of the error values from `krb_recvauth`. Since `krb_recvauth` calls other section 3 Kerberos routines (`3krb`) to perform its function, some of the error codes are references to the error codes of other functions.

RECVAUTH_OPNOTSUP
: The *options* bits sent to `krb_recvauth` contain a bit which is set but does not correspond to an option.

RECVAUTH_RD
: `krb_recvauth` could not read the authentication message sent to "B" using *fd*.

RECVAUTH_TKTLEN
: The length of the ticket-authenticator pair within the `krb_sendauth` message is longer than the maximum or less than or equal to 0.

RD_AP_VERSION
: The versions of Kerberos used by the caller of `krb_sendauth` is incompatible with the `krb_recvauth` version.

RD_AP_MSG_TYPE
: The ticket-authenticator pair given to `krb_recvauth` was not really a ticket-authenticator pair.

RD_AP_UNDEC
: The ticket could not be decyphered. This error can be caused by a forged or modified message.

RD_AP_INCON
: The message given to `krb_recvauth` contains an internal inconsistency. This could occur if the ticket in the ticket-authenticator pair does not match the authenticator.

RD_AP_BADD
: The ticket-authenticator pair cannot be used to authenticate a principal from the address specified by *f_addr*.

RD_AP_TIME
: The authenticator in the ticket-authenticator pair is too old to be used to authenticate the foreign principal.

RD_AP_NYV
: The time at which the ticket of the ticket-authenticator pair was created is too far ahead of the time of the local host of the local principal.

RD_AP_EXP
: The ticket is too old to be used.

-1 Negative one is returned if the each byte of the session key does not have odd parity.

-2 Negative two is returned if the session key is a weak key as defined by `des_is_weak_key`.

SENDMUT_OPNOTSUP
The options bits sent to `krb_sendmutal` contains a bit which is set but does not correspond to an option.

SENDMUT_MAKMSG
If there is an error in forming the mutual authentication message itself, SENDMUT_MAKMSG is returned.

SENDMUT_WR If the mutual authentication message cannot be written to the file descriptor *fd*, SENDMUT_WR is returned.

Restrictions

`krb_sendauth` and `krb_recvauth` will not work properly on sockets set to nonblocking I/O mode.

See Also

kerberos(3krb), krb_sendmutual(3krb), krb_svc_init(3krb), des_crypt(3krb, krb_get_lrealm(3krb), krb_set_tkt_string(3krb), krb.conf(5krb).

Name

krb_sendmutual, krb_recvmutual – Kerberos mutual authentication routines

Syntax

#include <krb.h>
#include <des.h>

int krb_sendmutual (*options, msg_out, success, fd, f_addr, l_addr, ad, schedule*)

long	*options*;
KTEXT	*msg_out*;
int	*success*;
int	*fd*;
struct sockaddr_in	**f_addr*;
struct sockaddr_in	**l_addr*;
AUTH_DAT	**ad*;
Key_schedule	*schedule*;

int krb_recvmutual (*options, fd, checksum, msg_in, msg_data, cred, schedule, l_addr, f_addr*)

long	*options*;
int	*fd*;
u_long	*checksum*;
KTEXT	*msg_in*;
MSG_DAT	**msg_data*;
CREDENTIALS	**cred*;
Key_schedule	*schedule*;
struct sockaddr_in	**l_addr*;
struct sockaddr_in	**f_addr*;

Arguments

options defined in `/usr/include/krb.h`. There is only one option currently supported, KOPT_NORDWR. If this option is not set, the mutual authentication information is read either from the supplied file descriptor, *fd*, or sent accross the supplied file descriptor, *fd*. If it is specified, then no data is read from or written to the file descriptor; instead, data is read from and written to the buffers supplied as parameters, *msg_in*, *msg_out*.

fd the file descriptor used to send data to the foreign principal, or it is the file descriptor from which data from the foreign principal can be read.

The foreign principal is the principal to which the principal that calls a `krb_sendmutual(3krb)` routine, the local principal, is attempting to mutually authenticate itself. The file descriptor must be associated with a socket that uses blocking I/O. The *fd* parameter is not used if the KOPT_NORDWR option is set.

f_addr the address of the socket that the foreign principal uses to communicate with the local principal.

l_addr the address of the socket that the local principal uses to communicate with the foreign principal.

msg_out If KOPT_NORDWR is sent as an option, *msg_out* is used as a buffer to store the mutual authentication data that should be sent to the foreign principal. If KOPT_NORDWR is not set, *msg_out* is not used and the mutual authentication message is written to *fd.*

success If success is not set to KSUCCESS, then the mutual authentication message generated by `krb_sendmutual` is a message indicating failure. This parameter is useful if the initial attempt to authenticate the foreign principal failed. Since this initial authentication attempt failed, then the attempt to authenticate the local principal to the foreign principal also must fail. If *success* is set to KSUCCESS, then a mutual authentication message is generated.

ad a pointer to the AUTH_DAT structure that describes the authentication association between the local and foreign principals. The *ad* structure is output from `krb_rd_req` (see `kerberos(3krb)`) and is used as input to `krb_sendmutual`. Space for the *ad* structure must be allocated.

checksum input to `krb_recvmutual`, it must have the same value as the *checksum* used as input to `krb_mk_req` (see `kerberos(3krb)`) or to `krb_sendauth` (see `krb_sendauth(3krb)`). The checksum is included in the ticket-authenticator pair produced by `krb_mk_req` and sent by `krb_sendauth` to the foreign principal. It serves as a secret piece of data that can only be known to the foreign principal if the foreign principal was authenticated as the foreign principal. It is included by `krb_sendmutual` in the mutual authentication message. If the checksum input to `krb_recvmutual` matches the one sent back by `krb_sendmutual`, then the caller of `krb_sendmutual` is authenticated to the caller of `krb_recvmutual`.

msg_in If KOPT_NORDWR is sent as an option, then data in *msg_in* is read as if it contained the mutual authentication bits sent to the local principal by the foreign principal. If KOPT_NORDWR is not set, then *msg_in* is not used and the mutual authentication message is read from *fd.*

msg_data a structure returned by `krb_recvmutal` that is filled with the mutual authentication message sent to the local principal as well as information about the status of the message. Space must be allocated for the *msg_data* structure.

cred a pointer to a credentials structure that is input to `krb_recvmutual`. The credentials structure that *cred* points to must be the credentials structure that includes the ticket that the local principal uses to authenticate the foreign principal. This credential structure is usually obtained through the use of `krb_get_cred` (See `kerberos(3krb)`).

schedule the key schedule derived from the session key between the local and foreign principals. It is input to both `krb_sendmutual` and `krb_recvmutual`, and it can be generated from the session key with `des_key_sched` (see `des_crypt(3krb)`).

Description

The krb_sendmutual (3krb) routines are designed to be used by applications which communicate over the network, support "on-the-wire" protocols in which authentication information can be placed, and require both parties in the communications process to be authenticated to the other (mutual authentication). They are best used with krb_mk_req and krb_rd_req. If a principal "A" calls krb_mk_req and sends the output to principal "B", which uses krb_rd_req to interpret the data successfully, then "B" will have authenticated principal "A". But, principal "A" will not know that the message it sent was really received by "B". To prove the identity of principal "B" to principal "A" after the calls to krb_mk_req and krb_rd_req are finished, the krb_sendmutual (3krb) calls are used.

krb_sendmutual and krb_recvmutual can also be used with krb_mk_req and krb_rd_req by applications which cannot tolerate additions to their "on-the-wire" protocols. After the communications channel between "A" and "B" is established, but before "A" and "B" communicate and before the "on-the-wire" protocol of the applications comes into effect, krb_mk_req and krb_rd_req can be used as described above to authenticate "A" to "B". krb_sendmutual and krb_recvmutual can then be used with the KOPT_NORDWR option not set to authenticate "B" to "A".

Since the authentication information is sent between the applications before the "on-the-wire" protocol of the application comes into effect, the application must develop some way to distinguish between the new authenticated initial message exchange and an old unauthenticated initial message exchange. This is not a recommended use for krb_sendmutual and krb_recvmutual. If you do not want to modify the "on-the-wire" protocol of an application, yet want to authenticate the application, then use the krb_sendauth (3krb) routines.

The routines of this library make extensive use of the following locally defined data types: KTEXT, AUTH_DAT, CREDENTIALS, Key_schedule, and MSG_DAT. For more specific information on the definitions of these data types, see the des.h and krb.h files.

krb_sendmutual

krb_sendmutual is used to produce and possibly send the data that will authenticate principal "B" to principal "A". If the authentication of principal "A" did not succeed, *success* should be set to KFAILURE, and krb_sendmutual produces a message indicating authentication failure. If it is set to KSUCCESS, then krb_sendmutual produces the data necessary to authenticate "B" to "A". If the option KOPT_NORDWR is set, the data is written to buffer *msg_out*; otherwise, it is written to file descriptor, *fd*.

The following is a list of the return values and, if they are error codes, their possible cause:

SENDMUT_OPNOTSUP
: The *options* bits sent to krb_sendmutal contain a bit that is set but does not correspond to an option.

SENDMUT_PARAM
: The *msg_out* structure must have space within it allocated to store the message. Otherwise, SENDMUT_PARAM is returned if the KOPT_NORDWR option is set.

SENDMUT_MAKMSG
If there is an error in forming the mutual authentication message itself, SENDMUT_MAKMSG is returned.

SENDMUT_WR If the message cannot be written to the file descriptor *fd*, SENDMUT_WR is returned.

KSUCCESS If the message has been correctly formed, KSUCCESS is returned.

krb_recvmutual

The `krb_recvmutual` routine interprets the mutual authentication message sent to principal "A" by principal "B". If the KOPT_NORDWR option is set, `krb_recvmutual` reads from buffer *msg_in*, the message sent from "B" to "A". Otherwise, it reads the message from file descriptor, *fd*. The *checksum* sent as input to `krb_recvmutual` must be the same checksum used as input to `krb_mk_req`. The checksum is an integral part of proving the identity of principal "B" to "A". The following is a list of the return values and, if they are error codes, their possible cause:

RECVMUT_OPNOTSUP
The *options* bits sent to `krb_recvmutal` contain a bit that is set, but does not correspond to an option.

RECVMUT_MSGLEN
The size of the *msg_in* buffer is incorrect.

RECVMUT_RD If the message cannot be read from the file descriptor *fd*, then SENDMUT_RD is returned.

RD_AP_VERSION If the Kerberos version used to create the mutual authentication message is not currently supported by `krb_recvmutual`, then RD_AP_VERSION is returned.

RD_AP_MSG_TYPE
If the message that is read from the file descriptor *fd*, or input as *msg_in* is not a mutual authentication message, RD_AP_MSG_TYPE is returned.

RD_AP_MODIFIED
If the message has been modified between principals "B" and "A", or if was incorrectly produced, then RD_AP_MODIFIED is returned.

RD_AP_TIME If the mutual authentication message is too old, RD_AP_TIME is returned.

KFAILURE If principal "A" was not authenticated to principal "B", or if the mutual authentication message fails to identify "B", KFAILURE is returned.

KSUCCESS If principal "B" has been correctly authenticated to principal "A", KSUCCESS is returned.

Restrictions

krb_sendmutal and krb_recvmutal will not work properly with sockets that do not use blocking I/O.

See Also

kerberos(3krb), krb_sendauth(3krb), des_crypt(3krb), krb_svc_init(3krb)

krb_set_tkt_string (3krb)

Name

krb_set_tkt_string, krb_set_srvtab_string – Environmental setup of the Kerberos libraries

Syntax

#include <krb.h>

void krb_set_tkt_string *(filename)*
char **filename*

void krb_set_srvtab_string *(filename)*
char **filename*

Arguments

filename The filename of the Kerberos ticket cache file or the name of the service table file.

Description

The `krb_set_tkt_string` routine sets the default name of the file that holds a cache of service tickets and associated session keys belonging to a Kerberos principal. The routine accepts a filename for the cache and copies this name into the local storage of `libkrb`. The default before any calls to `krb_set_tkt_string`, is `/var/dss/kerberos/tkt/tkt[`*uid*`]` where `uid` is the user ID of the process that calls `krb_set_tkt_string`.

You should call `krb_set_tkt_string` during Kerberos initialization to assure that any routines called later receive the proper name if they require the filename of the cache.

The `krb_set_srvtab_string` routine sets the default name of the file that stores the keys of the Kerberos applications running on the local host. The routine accepts a filename for the service table file and copies this name into the local storage of `libkrb`.

You should call `krb_set_srvtab_string` during the Kerberos initialization of a service to assure that any subsequently called routines that require the filename of the service table receive the proper name. The default, before any calls to the `krb_set_srvtab` string, is `/etc/srvtab`.

Files

`/var/dss/kerberos/tkt/tkt[`*uid*`]`

`/etc/srvtab`

See Also

kerberos(3krb), krb_sendauth(3krb), krb_sendmutual(3krb)

Name

krb_svc_init, krb_get_svc_in_tkt, krb_get_pw_in_tkt – Kerberos authentication initialization routines

Syntax

#include <krb.h>
#include <des.h>

krb_svc_init (*user, instance, realm, lifetime,*
srvtab_file, tkt_file)
char **user*, **instance*, **realm*;
int *lifetime*;
char **srvtab_file*, **tkt_file*;

krb_get_svc_in_tkt (*user, instance, realm, service,*
service_instance, lifetime,
srvtab_file)
char **user*, **instance*, **realm*, **service*,;
char **service_instance*;
int *lifetime*;
char **srvtab_file*;

krb_get_pw_in_tkt (*user, instance, realm, service,*
service_instance, lifetime,
password)
char **user*, **instance*, **realm*,;
char **service*, **service_instance*;
int *lifetime*;
char **password*;

Arguments

user For `krb_get_svc_in_tkt` and `krb_get_pw_in_tkt`, the primary name of the principal that is obtaining a ticket that will authenticate it to principal, *service*. For `krb_svc_init`, the primary name of the principal that is obtaining a ticket to communicate with the ticket-granting service.

instance For `krb_get_svc_in_tkt` and `krb_get_pw_in_tkt`, the instance name of the principal that is obtaining a ticket that will authenticate it to principal, *service*. For `krb_svc_init`, the instance name of the principal that is obtaining a ticket to communicate with the ticket-granting service.

realm For `krb_get_svc_in_tkt` and `krb_get_pw_in_tkt`, the realm name of the principal that is obtaining a ticket that will authenticate it to principal, *service*. For `krb_svc_init`, the realm name of the principal that is obtaining a ticket to communicate with the ticket-granting service.

service The primary name of the service for which a ticket will be obtained.

service_instance
The instance of the service for which a ticket will be obtained.

lifetime The number of five-minute intervals for which the obtained ticket should be valid. Values greater than 255 will be set to 255. Values greater than the maximum lifetime allowed for tickets given to the requesting principal will be set to the maximum lifetime allowed. The maximum lifetime of the tickets granted to a principal is determined when the principal is added to the Kerberos database.

srvtab_file The path name of the file that contains the key of the principal obtaining a ticket. If this value is set to the NULL pointer, the default service table (srvtab) file value is used. The default srvtab file value is set by default to `/etc/srvtab`, although this value can be changed by a call to the `krb_set_srvtab_string` function. (Refer to `krb_set_tkt_string(3krb)`).

tkt_file The path name of the file into which the credentials and tickets of the user or service should be placed. If the *tkt_file* parameter is equal to the NULL pointer, then the default ticket file value is used. The default ticket file value is set equal to `/var/dss/kerberos/tkt/tkt.`[*uid*] where `uid` is the user ID of the process that calls the above functions. The default ticket file value can be changed by the `krb_set_tkt_string(3krb)` function call.

password The password of the principal that is obtaining a ticket that will authenticate it to principal, *service*. If the password input is the NULL string, then `krb_get_pw_in_tkt` will prompt for a password on `stdout` and read the password from `stdin`.

Description

The `krb_svc_init(3krb)` routines are designed to obtain for the requesting principal a ticket to communicate with a specific service. They require that the password/key of the requesting principal be either available as an argument, or available from the *srvtab_file* argument or from `stdin`. Since the `krb_svc_init(3krb)` routines always require a password, they are best used to obtain the ticket used to communicate with the ticket-granting service. The ticket-granting ticket is used by the other Kerberos routines to obtain tickets to communicate with principals other than the ticket-granting service, without needing the key of the principal.

The `krb_sendauth(3krb)` routines as well as the `kerberos(3krb)` routines will not work as intended without the presence of a ticket-granting ticket.

The routines of `krb_svc_init(3krb)` are as follows:

krb_svc_init

For the principal with a primary name of *user*, an instance name of *instance*, and a realm name of *realm*, the `krb_svc_init` routine obtains a ticket that the principal can use to communicate with the ticket-granting service. The key of the principal is read from *srvtab_file* and the ticket obtained is placed in *tkt_file*.

If the *realm* argument is equivalent to the NULL string, then the realm of which the local host is a member, is used by default. If *lifetime* is equivalent to 0, then the default lifetime, 255, is used. If *srvtab_file* is not equivalent to the NULL string, then the *srvtab_file* parameter is used as the service table (srvtab) file name and the default srvtab file is set equal to the *srvtab_file* parameter. If *srvtab_file* is equivalent to NULL, then the default srvtab file is used. If the *tkt_file* parameter is not equivalent to the NULL string, then the *tkt_file* parameter is used as the ticket file name and the default ticket file is set equal to the *tkt_file* parameter. If the *tkt_file* parameter is NULL, then the default ticket file value is used.

`krb_svc_init` returns INT_OK if `krb_svc_init` has successfully obtained a ticket-granting ticket. The following is a list of most of the error values returned from `krb_svc_init` and their possible cause:

KFAILURE
: The `/etc/krb.conf` file (see `krb.conf(5krb)`) cannot be opened or it is not properly formed, or
The service table (srvtab) file does not exist, or
A read of the srvtab file failed, or
The srvtab file is badly formatted, or
The srvtab file did not contain the key of the principal with primary name, *user*, or
A write to the ticket file failed.

SKDC_CANT
: A Kerberos server must be contacted so that `krb_svc_init` can perform its function, but the attempt cannot be made because a socket cannot be opened or bound, or there is no Kerberos server listed in `/etc/krb.conf`.

SKDC_RETRY
: A Kerberos server needs to be contacted, but none responded even after several attempts.

INTK_PROT
: Kerberos protocol version mismatch. The version of the Kerberos protocol supported by `krb_svc_init` does not match the Kerberos protocol version supported by the `kerberos(8krb)` daemon.

INTK_BADPW
: The ticket returned by the `kerberos` daemon did not decrypt correctly. This is usually caused by an incorrect service password.

INTK_ERR
: The ticket sent from the `kerberos` daemon was not a ticket to communicate with the ticket-granting service, or
The ticket file cannot be accessed, or
The ticket file could not be created, or
A write operation to the ticket file failed.

TKT_FIL_LCK
: The ticket file could not be locked for access.

krb_get_svc_in_tkt

For the principal with a primary name of *user*, an instance name of *instance* and a realm name of *realm*, the `krb_get_svc_in_tkt` routine obtains a ticket to communicate with the principal that has a primary name of *service* and an instance name of *service_instance*. The key of the requesting primary is read from the file *srvtab_file* and the tickets are placed in the default ticket file. If the *srvtab_file* argument is equivalent to the NULL string, then the default srvtab file value is used instead of the *srvtab_file* parameter. The default srvtab file value and default ticket file value can be changed respectively by `krb_set_srvtab_sting` and `krb_set_tkt_string`. To obtain the ticket-granting ticket, the *service* parameter must be set equal to "krbtgt" and the *service_instance* argument must be set equal to the realm name of the local realm.

`krb_get_svc_in_tkt` returns INT_OK if `krb_get_svc_in_tkt` has successfully obtained a ticket to communicate with principal, *service*. The following is a list of most of the error values returned from `krb_get_svc_in_tkt` and their possible causes:

KFAILURE
: The `/etc/krb.conf` file cannot be opened or it is not properly formed, or
A read of the service table (srvtab) file failed, or
The srvtab file did not contain the key of the principal with primary name, *user*, or
A write to the ticket file failed.

SKDC_CANT
: A Kerberos server must be contacted in order for `krb_svc_init` to perform its function, but the attempt cannot be made because a socket cannot be opened or bound, or there is no Kerberos server listed in `/etc/krb.conf`.

SKDC_RETRY
: A Kerberos server needs to be contacted but none responded even after several attempts.

INTK_PROT
: Kerberos protocol version mismatch. The version of the Kerberos protocol supported by `krb_get_svc_in_tkt` does not match the Kerberos protocol version supported by the `kerberos` daemon.

INTK_BADPW
: The ticket returned by the `kerberos` daemon did not decrypt correctly. This is usually caused by an incorrect service password.

INTK_ERR
: The ticket sent from the `kerberos` daemon was not a ticket to communicate with the ticket-granting service, or
The ticket file cannot be accessed, or
The ticket file could not be created, or
A write operation to the ticket file failed.

TKT_FIL_LCK
: The ticket file could not be locked for access.

krb_get_pw_in_tkt

For the principal with a primary name of *user*, an instance name of *instance*, and a realm name of *realm*, the `krb_get_pw_in_tkt` routine obtains a ticket to communicate with the principal with a primary name of *service* and an instance name of *service_instance*. The key of the principal must be input either as the *password* parameter or, if the password field is equivalent to the NULL string, the password must be input from `stdin`.

The tickets that are obtained are placed in the default ticket file. The default ticket file can be changed by the `krb_set_tkt_string` function. To obtain the ticket-granting ticket, the *service* parameter must be set equal to "krbtgt" and the *service_instance* argument must be set equal to the realm name of the local realm.

`krb_get_pw_in_tkt` returns INT_OK if `krb_get_pw_in_tkt` has successfully obtained a ticket to communicate with principal, *service*. The following is a list of most of the error values returned from `krb_get_pw_in_tkt` and their possible causes:

KFAILURE
: `/etc/krb.conf` file cannot be opened or it is not properly formed. A write to the ticket file failed.

SKDC_CANT
: A Kerberos server must be contacted in order for `krb_svc_init` to perform its function but the attempt cannot be made because a socket cannot be opened or bound, or there is no Kerberos server listed in `/etc/krb.conf`.

SKDC_RETRY
: A Kerberos server needs to be contacted but none responded even after several attempts.

INTK_PROT
: Kerberos protocol version mismatch. The version of the Kerberos protocol supported by `krb_get_pw_in_tkt` does not match the Kerberos protocol version supported by the `kerberos` daemon.

INTK_BADPW
: The ticket returned by the `kerberos` daemon did not decrypt correctly. This is usually caused by an incorrect user password.

INTK_ERR
: The ticket sent from the `kerberos` daemon was not a ticket to communicate with the ticket-granting service, or
The ticket file cannot be accessed, or
The ticket file could not be created, or
A write operation to the ticket file failed.

TKT_FIL_LCK
: The ticket file could not be locked for access.

See Also

krb_get_lrealm(3krb), krb_set_tkt_string(3krb), kerberos(3krb), krb_sendauth(3krb), kerberos(8krb)

Math Routines (3m)

Insert tabbed divider here.
Then discard this sheet.

Name

intro – introduction to mathematical library functions

Description

These functions constitute the math library, *libm*. They are automatically loaded as needed by the FORTRAN compiler `f77`(1). The link editor searches this library under the ''–lm'' option. Declarations for these functions may be obtained from the include file <math.h>.

VAX Only

On VAX machines, the GFLOAT version of *libm* is used when you use the `ld`(1) command with the **lcg** option. Note that you must use the GFLOAT version of *libm* with modules compiled using the `cc`(1) with the **–Mg** option.

Also on VAX machines, note that neither the compiler nor the linker `ld`(1) can detect when mixed double floating point types are used, and the program may produce erroneous results if this occurs.

System V Compatibility

This library contains System V compatibility features that are available to general ULTRIX programs. For a discussion of how these features are documented, and how to specify that the System V environment is to be used in compiling and linking your programs, see `intro`(3).

Files

/usr/lib/libma
/usr/lib/libmg.a (VAX only)

Name

asinh, acosh, atanh – inverse hyperbolic functions

Syntax

#include <math.h>

double asinh(x)
double x;

double acosh(x)
double x;

double atanh(x)
double x;

Description

The `asinh`, `acosh`, and `atanh` functions compute the designated inverse hyperbolic functions for real arguments.

Errors Because of Roundoff, Etc.

These functions inherit much of their error from the `log1p`(3m) function.

Diagnostics

The `acosh` function returns the default quiet *NaN* if the argument is less than one.

The `atanh` function returns the default quiet *NaN* if the argument has an absolute value greater than or equal to one.

See Also

exp(3m), math(3m)

Name

asinh, acosh, atanh – inverse hyperbolic functions

Syntax

```
#include <math.h>

double asinh(x)
double x;

double acosh(x)
double x;

double atanh(x)
double x;
```

Description

These functions compute the designated inverse hyperbolic functions for real arguments.

Return Value

The function `acosh` returns 0.0 if the argument is less than 1.

The function `atanh` returns the HUGE value if the argument has absolute value greater than or equal to 1.

See Also

exp(3m), intro(3m)

Name

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

Syntax

#include <math.h>

double j0(*x*)
double *x;*

double j1(*x*)
double *x;*

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

double y0(*x*)
double *x;*

double y1(*x*)
double *x;*

double yn(*n,x*)
double *x;*

Description

These functions calculate bessel functions of the first and second kinds for real arguments and integer orders.

Return Value

Negative arguments cause `y0`, `y1`, and `yn` to return *NaN*. Arguments too large in magnitude cause `y0`, `y1`, and `yn` to return *NaN*.

Arguments too large in magnitude cause `j0`, `j1`, and `jn` to return zero.

Environment

When your program is compiled using the System V environment, nonpositive arguments cause `y0`, `y1` and `yn` to return the value HUGE and to set *errno* to EDOM. In addition, a message indicating DOMAIN error is printed on the standard error output.

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

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

See Also

math(3m)

Name

erf, erfc – error functions

Syntax

#include <math.h>

double erf(*x*)
double *x*;

double erfc(*x*)
double *x*;

Description

The `erf` function returns the error function of *x* defined as follows:

```
erf(x) = 2/sqrt(pi)*integral from 0 to x of exp(-t*t) dt.
```

The `erfc` function returns 1.0–erf(*x*).

The entry for the `erfc` function is provided because of the extreme loss of relative accuracy if erf(*x*) is called for large *x* and the result subtracted from 1. For example if $x = 10$, 12 places are lost.

Return Value

The `erf` and `erfc` functions return *NaN* when *x* is *NaN*.

See Also

math(3m)

Name

erf, erfc – error function and complementary error function

Syntax

#include <math.h>

double erf (x)
double x;

double erfc (x)
double x;

Description

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

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

See Also

exp(3m)
ULTRIX Programmer's Manual, Unsupported

Name

exp, expm1, log, log10, log1p, pow – exponential, logarithm, power

Syntax

#include <math.h>

double exp(*x*)
double *x*;

float fexp(*x*)
float *x*;

double expm1(*x*)
double *x*;

float fexpm1(*x*)
float *x*;

double log(*x*)
double *x*;

float flog(*x*)
float *x*;

double log10(*x*)
double *x*;

float flog10(*x*)
float *x*;

double log1p(*x*)
double *x*;

float flog1p(*x*)
float *x*;

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

Description

The `exp` and `fexp` functions return the exponential function of *x* for double and float data types, respectively.

The `expm1` and `fexpm1` functions return exp(*x*)-1 accurately, including tiny *x* for double and float data types, respectively.

The `log` and `flog` functions return the natural logarithm of *x* for double and float data types, respectively.

The `log10` and `flog10` functions return the logarithm of *x* to base 10 for double and float data types, respectively.

The `log1p` and `flog1p` functions return log(1+*x*) accurately, including tiny *x* for double and float data types, respectively.

The `pow` function returns *x****y*.

Error (due to roundoff)

The `exp`, `log`, `expm1`, and `log1p` functions are accurate to within an *ulp*, and `log10` is accurate to within approximately 2 *ulp*s; an *ulp* is one *U*nit in the *L*ast *P*lace.

The `pow` function is accurate to within 2 *ulp*s when its magnitude is moderate, but becomes less accurate as the `pow` result approaches the overflow or underflow thresholds. Theoretically, as these thresholds are approached, almost as many bits could be lost from the result as are indicated in the exponent field of the floating-point format for the resultant number. In other words, up to 11 bits for an IEEE 754 double-precision floating-point number. However, testing has never verified loss of precision as drastic as 11 bits. The worst cases have shown accuracy of results to within 300 *ulp*s for IEEE 754 double-precision floating-point numbers. In general, a `pow` (integer, integer) result is exact until it is larger than 2**53 (for IEEE 754 double-precision floating-point).

Return Value

All of the double precision functions return *NaN* if *x* or *y* is *NaN*.

The `exp` function returns HUGE_VAL when the correct value would overflow, and zero when the correct value would underflow.

The `log` and `log10` functions return *NaN* when *x* is less than or equal to zero or when the correct value would overflow.

The `pow` function returns *NaN* if *x* or *y* is *NaN*. When both *x* and *y* are zero, 1.0 is returned. When *x* is negative and *y* is not an integer, *NaN* is returned. If *x* is zero and *y* is negative, -HUGE_VAL is returned.

The `sqrt` function returns *NaN* when *x* is negative.

See Also

math(3m)

Name

exp, expm1, log, log10, log1p, pow, sqrt – exponential, logarithm, power, square root

Syntax

#include <math.h>

double exp(*x*)
double *x*;

double expm1(*x*)
double *x*;

double log(*x*)
double *x*;

double log10(*x*)
double *x*;

double log1p(*x*)
double *x*;

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

double sqrt(*x*)
double *x*;

Description

The `exp` function returns the exponential function of *x*.

The `expm1` function returns exp(*x*)-1 accurately even for tiny *x*.

The `log` function returns the natural logarithm of *x*; `log10` returns the base 10 logarithm.

The `log1p` function returns log(1+*x*) accurately even for tiny *x*.

The `pow` function returns *x* raised to the *y* power.

The `sqrt` function returns the square root of *x*.

Return Value

The `exp` function returns HUGE_VAL and sets *errno* to ERANGE when the correct value would overflow. When the correct value would underflow it returns zero and *errno* is set to ERANGE.

The `expm1` function returns HUGE_VAL and sets *errno* to ERANGE when the correct value would overflow. When the correct value would underflow it returns -1.

The `log` and `log10` functions return -HUGE_VAL and set *errno* to EDOM when *x* is less than or equal to zero. When the correct value would overflow flow they return -HUGE_VAL and *errno* is set to ERANGE.

The `log1p` function returns -HUGE_VAL and sets *errno* to EDOM when *x* is less than or equal to -1. When the correct value would overflow flow it returns -HUGE_VAL and *errno* is set to ERANGE.

The `pow` function has many special cases. When *x* and *y* are both zero it returns 1.0. When *x* is negative and *y* is not an integer value it returns zero and *errno* is set to EDOM. When *x* is zero and *y* is negative it returns -HUGE_VAL and *errno* is set to EDOM. When the correct value would overflow HUGE_VAL is returned and *errno* is set to ERANGE. When the correct value would underflow zero is returned and *errno* is set to ERANGE.

The `sqrt` function returns zero and sets *errno* to EDOM when *x* is negative.

Environment

When your program is compiled using the System V environment, `exp` returns HUGE when the correct value would overflow, and sets *errno* to ERANGE; `exp` returns zero when the correct value would underflow, and sets errno to ERANGE.

The `log` and `log10` functions return HUGE and set *errno* to EDOM when *x* is nonpositive. An error message is printed on the standard error output.

The `pow` function returns zero and sets *errno* to EDOM when *x* is non-positive and y is not an integer, or when *x* and *y* are both zero. In these cases, a message indicating DOMAIN error is printed on the standard error output. When the correct value for `pow` would overflow, `pow` returns HUGE and sets *errno* to ERANGE.

The `sqrt` function returns zero and sets *errno* to EDOM when *x* is negative. A message indicating DOMAIN error is printed on the standard error output.

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

NOTE

DOMAIN error is only printed in the System V environment.

See Also

hypot(3m), intro(3m), sinh(3m)

Name

floor, ffloor, fabs, ceil, ceil, trunc, ftrunc, fmod, rint – floor, absolute value, ceiling, truncation, floating point remainder and round-to-nearest functions

Syntax

#include <math.h>

double floor(*x*)
double *x*;

float ffloor(*x*)
float *x*;

double ceil(*x*)
double *x*;

float fceil(*x*)
float *x*;

double trunc(*x*)
double *x*;

float ftrunc(*x*)
float *x*;

double fabs(*x*)
double *x*;

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

double rint(*x*)
double *x*;

Description

The `floor` and `ffloor` routines return the largest integer which is not greater than x for double and float data types, respectively.

The `ceil` and `fceil` routines return the smallest integer which is not less than x for double and float data types, respectively.

The `trunc` and `ftrunc` routines return the integer (represented as a floating-point number) of x with the fractional bits truncated for double and float data types respectively.

The `fabs` routine returns the absolute value $|x|$.

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

The `rint` routine returns the integer (represented as a double precision number) nearest x in the direction of the prevailing rounding mode.

In the default rounding mode, to nearest, `rint(x)` is the integer nearest x with the additional stipulation that if |rint(x)–x|=1/2 then `rint(x)` is even. Other rounding modes can make `rint` act like `floor` or `ceil`, or round towards zero.

Another way to obtain an integer near x is to declare (in C)

```
double x;      int k;      k = x;
```

The C compiler rounds x towards 0 to get the integer k. Also note that, if x is larger than k can accommodate, the value of k and the presence or absence of an integer overflow are hard to predict.

The `fabs` routine is in libc.a rather than libm.a.

See Also

abs(3), ieee(3m), math(3m)

Name

fabs, floor, ceil, fmod, rint – absolute value, floor, ceiling, floating point remainder, and round-to-nearest functions

Syntax

#include <math.h>

double floor(*x*)
double *x*;

double ceil(*x*)
double *x*;

double fabs(*x*)
double *x*;

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

double rint(*x*)
double *x*;

Description

The `fabs` routine returns the absolute value $|x|$.

The `floor` routine returns the largest integer no greater than x.

The `ceil` routine returns the smallest integer no less than x.

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

The `rint` routine returns the integer (represented as a double precision number) nearest x in the direction of the prevailing rounding mode.

See Also

abs(3), intro(3m)

Name

gamma, lgamma, signgam – log gamma function

Syntax

#include <math.h>

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

double lgamma(*x*)
double *x*;

extern int `signgam;`

Description

The `gamma` function returns $\ln |\Gamma(|x|)|$. The sign of $\Gamma(|x|)$ is returned in the external integer `signgam`. The following C program might be used to calculate Γ:

```
y = gamma(x);
if (y > 88.0)
        error();
y = exp(y);
if(signgam)
        y = -y;
```

The `lgamma` function is another name for the `gamma` function.

Return Value

The `gamma` and `lgamma` functions return *NaN* when *x* is *NaN* or when it is an integer value less than or equal to zero. On overflow `gamma` and `lgamma` functions return HUGE_VAL.

Environment

When your program is compiled using the System V environment for nonpositive integer values, HUGE is returned, and *errno* is set to EDOM. A message indicating DOMAIN error is printed on the standard error output.

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

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

See Also

matherr(3m)

Name

gamma, lgamma, signgam – log gamma function

Syntax

#include <math.h>

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

double lgamma(*x*)
double *x*;

extern int `signgam;`

Description

The `gamma` function returns ln $|\Gamma(|x|)|$. The sign of $\Gamma(|x|)$ is returned in the external integer `signgam`. The following C program might be used to calculate Γ:

```
y = gamma(x);
if (y > 88.0)
       error();
y = exp(y);
if(signgam)
       y = -y;
```

The `lgamma` function is another name for the gamma function.

Return Value

The `gamma` and `lgamma` functions return HUGE_VAL and set *errno* to EDOM when *x* is an integer value less than or equal to zero. When the correct value would overflow they return HUGE_VAL and set *errno* to ERANGE.

Environment

When your program is compiled using the System V environment for nonpositive integer values, HUGE is returned, and *errno* is set to EDOM. A message indicating DOMAIN error is printed on the standard error output.

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

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

See Also

matherr(3m)

hypot(3m)

Name

hypot, cabs – Euclidean distance, complex absolute value

Syntax

#include <math.h>

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

float fhypot(float x, float y)

double cabs(*z***)**
struct {double x,y;} *z***;**

float fcabs(*z***)**
struct {float x,y;} *z***;**

Description

The `hypot`, `fhypot`, `cabs`, and `fcabs` functions return the following:

`sqrt(x*x+y*y)`

This computation prevents underflows and overflows only if the final result dictates it.

The functions `fhypot` and `fcabs` are equivalent to the `hypot` and `cabs` function with the exception of float data type.

Error

When rounding off, for example, below 0.97 *ulp*s. Consequently `hypot` (5.0,12.0) = 13.0 exactly; in general, `hypot` and `cabs` return an integer whenever an integer might be expected.

The same cannot be said for the shorter and faster version of `hypot` and `cabs` that is provided in the comments in cabs.c; its error can exceed 1.2 *ulp*s.

Return Value

If the correct value overflows, `hypot` and `cabs` return HUGE_VAL. If *x* or *y* is *NaN*, then *NaN* is returned.

See Also

math(3m), sqrt(3m)

Name

hypot, cabs – Euclidean distance

Syntax

#include <math.h>

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

double cabs(*z***)**
struct {double x,y;} *z***;**

Description

The `hypot` and `cabs` functions return

sqrt(x*x + y*y),

taking precautions against unwarranted overflows.

Return Value

The `hypot` and `cabs` functions return HUGE_VAL and sets *errno* to ERANGE when the correct value would overflow.

Environment

When your program is compiled using the System V environment, if the correct value would overflow, `hypot` returns HUGE and sets *errno* to ERANGE.

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

The `cabs` subroutine does not exist in the System V environment. For sqrt, see `exp`(3m).

See Also

exp(3m)

Name

copysign, drem, finite, logb, scalb – copysign, remainder, exponent manipulations

Syntax

#include <math.h>

double copysign(x,y)
double x,y;

double drem(x,y)
double x,y;

int finite(x)
double x;

double logb(x)
double x;

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

Description

These functions are required, or recommended by the IEEE standard 754 for floating–point arithmetic.

The `copysign` function returns x with its sign changed to y's.

The `drem(x,y)` function returns the remainder r := x – n*y where n is the integer nearest the exact value of x/y. Additionally if $|n-x/y|=1/2$, then n is even. Consequently the remainder is computed exactly and $|r| \le |y|/2$. Note that `drem(x,0)` is the exception (see DIAGNOTICS).

Finite(x) = 1 just when $-\infty < x < +\infty$,
= 0 otherwise (when $|x| = \infty$ or x is *NaN*)

The `logb(x)` returns a signed integer converted to double–precision floating–point and so chosen that $1 \le |x|/2^{**}n < 2$ unless x = 0 or $|x| = \infty$ or x lies between 0 and the Underflow Threshold.

Scalb(x,n) = x*(2**n) computed, for integer n, without first computing 2**N.

Diagnostics

IEEE 754 defines drem(x,0) and drem(∞,y) to be invalid operations that produce a *NaN*.

IEEE 754 defines logb(±∞) = +∞ and logb(0) = –∞, and requires the latter to signal Division–by–Zero.

Restrictions

IEEE 754 currently specifies that logb(denormalized no.) = logb(tiniest normalized no. > 0) but the consensus has changed to the specification in the new proposed IEEE standard p854, namely that logb(x) satisfy

$1 \le \mathrm{scalb}(|x|,-\mathrm{logb}(x)) < \mathrm{Radix}$ ... = 2 for IEEE 754 for every x except 0, ∞ and *NaN*. Almost every program that assumes 754's specification will work correctly if logb follows 854's specification instead.

IEEE 754 requires copysign(x,*NaN*) = ±x but says nothing else about the sign of a *NaN*.

See Also

floor(3M), fp_class(3), math(3M)

isnand(3m)

Name

isnand, isnanf – test for floating point NaN (Not-A-Number)

Syntax

#include <ieeefp.h>

int isnand (dsrc)
double dsrc;

int isnanf (fsrc)
float fsrc;

Description

The `isnand` and `isnanf` routines return the value 1 for true if the argument dsrc or fsrc is a NaN; otherwise they return the value 0 for false.

Neither routine generates any exception, even for signaling NaNs.

The `isnan` function is implemented as a macro included in <ieeefp.h>.

Name

math – introduction to mathematical library functions

Description

These functions constitute the C math library *libm*. There are two versions of the math library *libm.a* and *libm43.a*.

The first, *libm.a,* contains routines written in MIPS assembly language and tuned for best performance and includes many routines for the *float* data type. The routines in there are based on the algorithms of Cody and Waite or those in the 4.3 BSD release, whichever provides the best performance with acceptable error bounds. Those routines with Cody and Waite implementations are marked with a '*' in the list of functions below.

The second version of the math library, *libm43.a,* contains routines all based on the original codes in the 4.3 BSD release. The difference between the two version's error bounds is typically around 1 unit in the last place, whereas the performance difference may be a factor of two or more.

The link editor searches this library under the ''–lm'' (or ''–lm43'') option. Declarations for these functions may be obtained from the include file *<math.h>*. The Fortran math library is described in ''man 3f intro''.

List Of Functions

The cycle counts of all functions are approximate; cycle counts often depend on the value of argument. The error bound sometimes applies only to the primary range.

		Error Bound (ULPs)		Cycles	
Name	**Description**	**libm.a**	**libm43.a**	**libm.a**	**libm43.a**
acos	inverse trig function	3	3	?	?
acosh	inverse hyperbolic function	3	3	?	?
asin	inverse trig function	3	3	?	?
asinh	inverse hyperbolic function	3	3	?	?
atan	inverse trig function	1	1	152	260
atanh	inverse hyperbolic function	3	3	?	?
atan2	inverse trig function	2	2	?	?
cabs	complex absolute value	1	1	?	?
cbrt	cube root	1	1	?	?
ceil	integer no less than	0	0	?	?
copysign	copy sign bit	0	0	?	?
cos*	trig function	2	1	128	243
cosh*	hyperbolic function	?	3	142	294
drem	remainder	0	0	?	?
erf	error function	?	?	?	?

erfc	complementary error function	?	?	?	?
exp*	exponential	2	1	101	230
expm1	exp(x)–1	1	1	281	281
fabs	absolute value	0	0	?	?
fatan*	inverse trig function	3		64	
fcos*	trig function	1		87	
fcosh*	hyperbolic function	?		105	
fexp*	exponential	1		79	
flog*	natural logarithm	1		100	
floor	integer no greater than	0	0	?	?
fsin*	trig function	1		68	
fsinh*	hyperbolic function	?		44	
fsqrt	square root	1		95	
ftan*	trig function	?		61	
ftanh*	hyperbolic function	?		116	
hypot	Euclidean distance	1	1	?	?
j0	bessel function	?	?	?	?
j1	bessel function	?	?	?	?
jn	bessel function	?	?	?	?
lgamma	log gamma function	?	?	?	?
log*	natural logarithm	2	1	119	217
logb	exponent extraction	0	0	?	?
log10*	logarithm to base 10	3	3	?	?
log1p	log(1+x)	1	1	269	269
pow	exponential x**y	60–500	60–500	?	?
rint	round to nearest integer	0	0	?	?
scalb	exponent adjustment	0	0	?	?
sin*	trig function	2	1	101	222
sinh*	hyperbolic function	?	3	79	292
sqrt	square root	1	1	133	133
tan*	trig function	?	3	92	287
tanh*	hyperbolic function	?	3	156	293
y0	bessel function	?	?	?	?
y1	bessel function	?	?	?	?
yn	bessel function	?	?	?	?

In 4.3 BSD, distributed from the University of California in late 1985, most of the foregoing functions come in two versions, one for the double–precision "D" format in the DEC VAX–11 family of computers, another for double–precision arithmetic conforming to the IEEE Standard 754 for Binary Floating–Point Arithmetic. The two versions behave very similarly, as should be expected from programs more accurate and robust than was the norm when UNIX was born. For instance, the programs are accurate to within the numbers of *ulp*s tabulated above; an *ulp* is one *U*nit in the *L*ast *P*lace. And the programs have been cured of anomalies that afflicted the older math library *libm* in which incidents like the following had been reported:

> sqrt(–1.0) = 0.0 and log(–1.0) = –1.7e38.
> cos(1.0e–11) > cos(0.0) > 1.0.
> pow(x,1.0) ≠ x when x = 2.0, 3.0, 4.0, ..., 9.0.

pow(–1.0,1.0e10) trapped on Integer Overflow.
sqrt(1.0e30) and sqrt(1.0e–30) were very slow.

RISC machines conform to the IEEE Standard 754 for Binary Floating–Point Arithmetic, to which only the notes for IEEE floating-point apply and are included here.

BIEEE STANDARD 754 Floating–Point Arithmetic:

This standard is on its way to becoming more widely adopted than any other design for computer arithmetic.

The main virtue of 4.3 BSD's *libm* codes is that they are intended for the public domain; they may be copied freely provided their provenance is always acknowledged, and provided users assist the authors in their researches by reporting experience with the codes. Therefore no user of UNIX on a machine that conforms to IEEE 754 need use anything worse than the new *libm.*

Properties of IEEE 754 Double–Precision:

Wordsize: 64 bits, 8 bytes. **Radix:** Binary.
Precision: 53 significant bits, roughly like 16 significant decimals.
If x and x' are consecutive positive Double–Precision numbers (they differ by 1 *ulp*), then
1.1e–16 < 0.5**53 < (x'–x)/x ≤ 0.5**52 < 2.3e–16.
Range: Overflow threshold = 2.0**1024 = 1.8e308
Underflow threshold = 0.5**1022 = 2.2e–308
Overflow goes by default to a signed ∞.
Underflow is *Gradual,* rounding to the nearest integer multiple of 0.5**1074 = 4.9e–324.
Zero is represented ambiguously as +0 or –0.
Its sign transforms correctly through multiplication or division, and is preserved by addition of zeros with like signs; but x–x yields +0 for every finite x. The only operations that reveal zero's sign are division by zero and copysign(x,±0). In particular, comparison (x > y, x ≥ y, etc.) cannot be affected by the sign of zero; but if finite x = y then ∞ = 1/(x–y) ≠ –1/(y–x) = –∞.
∞ is signed.
it persists when added to itself or to any finite number. Its sign transforms correctly through multiplication and division, and (finite)/±∞ = ±0 (nonzero)/0 = ±∞. But ∞–∞, ∞*0 and ∞/∞ are, like 0/0 and sqrt(–3), invalid operations that produce *NaN*. ...
Reserved operands:
there are 2**53–2 of them, all called *NaN* (*N*ot *a N*umber). Some, called Signaling *NaN*s, trap any floating–point operation performed upon them; they could be used to mark missing or uninitialized values, or nonexistent elements of arrays. The rest are Quiet *NaN*s; they are the default results of Invalid Operations, and propagate through subsequent arithmetic operations. If x ≠ x then x is *NaN*; every other predicate (x > y, x = y, x < y, ...) is FALSE if *NaN* is involved.

NOTE

Trichotomy is violated by *NaN*. Besides being FALSE, predicates that entail ordered comparison, rather than mere (in)equality, signal Invalid Operation when *NaN* is involved.

Rounding:

Every algebraic operation (+, –, *, /, √) is rounded by default to within half an *ulp*, and when the rounding error is exactly half an *ulp* then the rounded value's least significant bit is zero. This kind of rounding is usually the best kind, sometimes provably so; for instance, for every x = 1.0, 2.0, 3.0, 4.0, ..., 2.0**52, we find (x/3.0)*3.0 == x and (x/10.0)*10.0 == x and ... despite that both the quotients and the products have been rounded. Only rounding like IEEE 754 can do that. But no single kind of rounding can be proved best for every circumstance, so IEEE 754 provides rounding towards zero or towards +∞ or towards –∞ at the programmer's option. And the same kinds of rounding are specified for Binary–Decimal Conversions, at least for magnitudes between roughly 1.0e–10 and 1.0e37.

Exceptions:

IEEE 754 recognizes five kinds of floating–point exceptions, listed below in declining order of probable importance.

Exception	Default Result
Invalid Operation	*NaN*, or FALSE
Overflow@±∞	
Divide by Zero	±∞
Underflow	Gradual Underflow
Inexact	Rounded value

NOTE

An Exception is not an Error unless handled badly. What makes a class of exceptions exceptional is that no single default response can be satisfactory in every instance. On the other hand, if a default response will serve most instances satisfactorily, the unsatisfactory instances cannot justify aborting computation every time the exception occurs.

For each kind of floating–point exception, IEEE 754 provides a Flag that is raised each time its exception is signaled, and stays raised until the program resets it. Programs may also test, save and restore a flag. Thus, IEEE 754 provides three ways by which programs may cope with exceptions for which the default result might be unsatisfactory:

1) Test for a condition that might cause an exception later, and branch to avoid the exception.

2) Test a flag to see whether an exception has occurred since the program last reset its flag.

3) Test a result to see whether it is a value that only an exception could have produced.

NOTE

The only reliable ways to discover whether Underflow has occurred are to test whether products or quotients lie closer to zero than the underflow threshold, or to test the Underflow flag. (Sums and differences cannot underflow in IEEE 754; if $x \neq y$ then $x-y$ is correct to full precision and certainly nonzero regardless of how tiny it may be.) Products and quotients that underflow gradually can lose accuracy gradually without vanishing, so comparing them with zero (as one might on a VAX) will not reveal the loss. Fortunately, if a gradually underflowed value is destined to be added to something bigger than the underflow threshold, as is almost always the case, digits lost to gradual underflow will not be missed because they would have been rounded off anyway. So gradual underflows are usually *provably* ignorable. The same cannot be said of underflows flushed to 0.

At the option of an implementor conforming to IEEE 754, other ways to cope with exceptions may be provided:

4) ABORT. This mechanism classifies an exception in advance as an incident to be handled by means traditionally associated with error–handling statements like "ON ERROR GO TO ...". Different languages offer different forms of this statement, but most share the following characteristics:

— No means is provided to substitute a value for the offending operation's result and resume computation from what may be the middle of an expression. An exceptional result is abandoned.

— In a subprogram that lacks an error–handling statement, an exception causes the subprogram to abort within whatever program called it, and so on back up the chain of calling subprograms until an error–handling statement is encountered or the whole task is aborted and memory is dumped.

5) STOP. This mechanism, requiring an interactive debugging environment, is more for the programmer than the program. It classifies an exception in advance as a symptom of a programmer's error; the exception suspends execution as near as it can to the offending operation so that the programmer can look around to see how it happened. Quite often the first several exceptions turn out to be quite unexceptionable, so the programmer ought ideally to be able to resume execution after each one as if execution had not been stopped.

6) ... Other ways lie beyond the scope of this document.

The crucial problem for exception handling is the problem of Scope, and the problem's solution is understood, but not enough manpower was available to implement it fully in time to be distributed in 4.3 BSD's *libm*. Ideally, each elementary function should act as if it were indivisible, or atomic, in the sense that ...

i) No exception should be signaled that is not deserved by the data supplied to that function.

ii) Any exception signaled should be identified with that function rather than with one of its subroutines.

iii) The internal behavior of an atomic function should not be disrupted when a calling program changes from one to another of the five or so ways of handling exceptions listed above, although the definition of the function may be correlated intentionally with exception handling.

Ideally, every programmer should be able *conveniently* to turn a debugged subprogram into one that appears atomic to its users. But simulating all three characteristics of an atomic function is still a tedious affair, entailing hosts of tests and saves–restores; work is under way to ameliorate the inconvenience.

Meanwhile, the functions in *libm* are only approximately atomic. They signal no inappropriate exception except possibly ...

Over/Underflow
: when a result, if properly computed, might have lain barely within range, and

Inexact in *cabs*, *cbrt*, *hypot*, *log10* and *pow*
: when it happens to be exact, thanks to fortuitous cancellation of errors.

Otherwise, ...

Invalid Operation is signaled only when
: any result but *NaN* would probably be misleading.

Overflow is signaled only when
: the exact result would be finite but beyond the overflow threshold.

Divide–by–Zero is signaled only when
: a function takes exactly infinite values at finite operands.

Underflow is signaled only when
: the exact result would be nonzero but tinier than the underflow threshold.

Inexact is signaled only when
: greater range or precision would be needed to represent the exact result.

Exceptions on RISC machines:

The exception enables and the flags that are raised when an exception occurs (as well as the rounding mode) are in the floating–point control and status register. This register can be read or written by the routines described on the man page *fpc*(3). This register's layout is described in the file <*mips/fpu.h*> in UMIPS–BSD releases and in <*sys/fpu.h*> in UMIPS–SYSV releases.

What is currently available is only the raw interface which was only intended to be used by the code to implement IEEE user trap handlers. IEEE floating–point exceptions are enabled by setting the enable bit for that exception in the floating–point control and status register. If an exception then occurs the UNIX signal SIGFPE is sent to the process. It is up to the signal handler to determine the instruction that caused the exception and to take the action specified by the user. The instruction that caused the exception is in one of two places. If the floating–point board is used (the floating–point implementation revision register indicates this in it's implementation field) then the instruction that caused the exception is in the floating–point exception instruction register. In all other implementations the instruction that caused the exception is at the address of the program counter as modified by the branch delay bit in the cause register. Both the program counter and cause register are in the sigcontext structure passed to the signal handler (see `signal`(3)). If the program is to be continued past the instruction that caused the exception the program counter in the signal context must be advanced. If the instruction is in a branch delay slot then the branch must be emulated to determine if the branch is taken and then the resulting program counter can be calculated (see `emulate_branch`(3) and `signal`(3)).

Restrictions

When signals are appropriate, they are emitted by certain operations within the codes, so a subroutine–trace may be needed to identify the function with its signal in case method 5) above is in use. And the codes all take the IEEE 754 defaults for granted; this means that a decision to trap all divisions by zero could disrupt a code that would otherwise get correct results despite division by zero.

See Also

fpc(3), signal(3), emulate_branch(3)
R2010 Floating Point Coprocessor Architecture
R2360 Floating Point Board Product Description

An explanation of IEEE 754 and its proposed extension p854 was published in the IEEE magazine MICRO in August 1984 under the title "A Proposed Radix– and Word–length–independent Standard for Floating–point Arithmetic" by W. J. Cody et al.

Articles in the IEEE magazine COMPUTER vol. 14 no. 3 (Mar. 1981), and in the ACM SIGNUM Newsletter Special Issue of Oct. 1979, may be helpful although they pertain to superseded drafts of the standard.

Name

matherr – error-handling function for System V math library

Syntax

#include <math.h>

int matherr (x)
struct exception *x;

Description

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

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

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

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

The element *name* points to a string containing the name of the function that had the error. The variables *arg1* and *arg2* are the arguments to the function that had the error. The *retval* is a double that is returned by the function having the error. If it supplies a return value, the user's `matherr` must return nonzero. If the default error value is to be returned, the user's `matherr` must return 0.

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

Examples

```
matherr(x)
register struct exception *x;
{
       switch (x->type) {
       case DOMAIN:
       case SING: /* print message and abort */
             fprintf(stderr, "domain error in %s\n", x->name);
             abort( );
```

```
        case OVERFLOW:
                if (!strcmp("exp", x->name)) {
                        /* if exp, print message, return the argument */
                        fprintf(stderr, "exp of %f\n", x->arg1);
                        x->retval = x->arg1;
                } else if (!strcmp("sinh", x->name)) {
                        /* if sinh, set errno, return 0 */
                        errno = ERANGE;
                        x->retval = 0;
                } else
                        /* otherwise, return HUGE */
                        x->retval = HUGE;
                break;
        case UNDERFLOW:
                return (0); /* execute default procedure */
        case TLOSS:
        case PLOSS:
                /* print message and return 0 */
                fprintf(stderr, "loss of significance in %s\n", x->name);
                x->retval = 0;
                break;
        }
        return (1);
}
```

DEFAULT ERROR HANDLING PROCEDURES

	Types of Errors					
	DOMAIN	SING	OVERFLOW	UNDERFLOW	TLOSS	PLOSS
BESSEL:	–	–	H	0	M, 0	*
y0, y1, yn (neg. no.)	M, –H	–	–	–	–	–
EXP:	–	–	H	0	–	
POW:	–	–	H	0	–	–
(neg.)**(non-int.), 0**0	M, 0	–	–	–	–	–
LOG:						
log(0):	–	M, –H	–	–	–	–
log(neg.):	M, –H	–	–	–	–	–
SQRT:	M, 0	–	–	–	–	–
GAMMA:	–	M, H	–	–	–	–
HYPOT:	–	–	H	–	–	–
SINH, COSH:	–	–	H	–	–	–
SIN, COS:	–	–	–	–	M, 0	*
TAN:	–	–	H	–	M, 0	*
ACOS, ASIN:	M, 0	–	–	–	–	–

ABBREVIATIONS

*	As much as possible of the value is returned.
M	Message is printed.
H	HUGE is returned.
–H	–HUGE is returned.
0	0 is returned.

sin(3m)

Name

sin, cos, tan, asin, acos, atan, atan2 – trigonometric functions and their inverses

Syntax

#include <math.h>

double sin(*x***)**
double *x***;**

float fsin(*x***)**
float *x***;**

double cos(*x***)**
double *x***;**

float fcos(*x***)**
float *x***;**

double tan(*x***)**
double *x***;**

float ftan(*x***)**
float *x***;**

double asin(*x***)**
double *x***;**

float fasin(*x***)**
float *x***;**

double acos(*x***)**
double *x***;**

float facos(*x***)**
float *x***;**

double atan(*x***)**
double *x***;**

float fatan(*x***)**
float *x***;**

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

float fatan2(*y,x***)**
float *y,x***;**

Description

The `sin`, `cos`, and `tan` functions return trigonometric functions of radian arguments *x* for double data types.

The `fsin`, `fcos`, and `ftan` functions return trigonometric functions for float data types.

The `asin` and `fasin` functions return the arc sine in the range $-\pi/2$ to $\pi/2$ for double and float data types, respectively.

The `acos` and `facos` functions return the arc cosine in the range 0 to π for double and float data types, respectively.

The `atan` and `fatan` functions return the arc tangent in the range $-\pi/2$ to $\pi/2$ for double and float data types, respectively.

The `atan2` and `fatan2` functions return the arc tangent of y/x in the range $-\pi$ to π, using the signs of both arguments to determine the quadrant of the return value for double and float data types, respectively.

Error (due to roundoff)

When P stands for the number stored in the computer in place of π = 3.14159 26535 89793 23846 26433 ... . and "trig" stands for one of "sin", "cos" or "tan", then the expression "trig(x)" in a program actually produces an approximation to trig($x*\pi$/P), and "atrig(x)" approximates (P/π)*atrig(x). The approximations are close.

P differs from π by a fraction of an *ulp*; the difference is apparent only if the argument x is huge, and even then the difference is likely to be swamped by the uncertainty in x. Every trigonometric identity that does not involve π explicitly is satisfied equally well regardless of whether P = π. For example, $\sin^2(x)+\cos^2(x) = 1$ and $\sin(2x) = 2\sin(x)\cos(x)$ to within a few *ulp*s regardless of how big x is. Therefore, the difference between P and π is unlikely to effect scientific and engineering computations.

Return Value

All the double functions return *NaN* if *NaN* is passed in.

If $|x| > 1$ then `asin` (x) and `acos` (x) will return the default quiet *NaN*.

The `atan2` function defines `atan2` (0,0) = *NaN*.

See Also

hypot(3m), math(3m), sqrt(3m)

Name

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

Syntax

#include <math.h>

double sin(*x*)
double *x*;

double cos(*x*)
double *x*;

double tan(*x*)
double *x*;

double asin(*x*)
double *x*;

double acos(*x*)
double *x*;

double atan(*x*)
double *x*;

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

Description

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

The `asin` subroutine returns the arc sin in the range $-\pi/2$ to $\pi/2$.

The `acos` subroutine returns the arc cosine in the range 0 to π.

The `atan` subroutine returns the arc tangent of x in the range $-\pi/2$ to $\pi/2$.

The `atan2` subroutine returns the arc tangent of x/y in the range $-\pi$ to π.

Restrictions

The value of `tan` for arguments greater than about 2**31 is unreliable.

Return Value

Arguments of magnitude greater than 1 cause `asin` and `acos` to return zero and set *errno* to EDOM.

The `atan2` function returns zero and sets *errno* to EDOM when x and y are both zero.

Environment

When your program is compiled using the System V environment, `sin`, `cos` and `tan` lose accuracy when their argument is far from zero. For arguments sufficiently large, these functions return 0 when there would otherwise be a complete loss of significance. In this case a message indicating TLOSS error is printed on the standard error output. For less extreme arguments, a PLOSS error is generated but no message is printed. In both cases, *errno* is set to ERANGE.

The `tan` subroutine returns HUGE for an argument which is near an odd multiple of $\pi/2$ when the correct value would overflow, and sets *errno* to ERANGE.

Arguments of magnitude greater than 1.0 cause `asin` and `acos` to return 0 and to set *errno* to EDOM. In addition, a message indicating DOMAIN error is printed on the standard error output.

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

sinh(3m)

Name

sinh, cosh, tanh – hyperbolic functions

Syntax

#include <math.h>

double sinh(x)
double x;

float fsinh(x)
float x;

double cosh(x)
double x;

float fcosh(x)
float x;

double tanh(x)
double x;

float ftanh(x)
float x;

Description

These functions compute the designated hyperbolic functions for double and float data types.

Error

Below 2.4 *ulps* (unit in the last place).

Diagnostics

The `sinh` and `cosh` functions return $+\infty$ (and *sinh* may return $-\infty$ for negative x) if the correct value would overflow.

See Also

math(3m)

Name

sinh, cosh, tanh – hyperbolic functions

Syntax

#include <math.h>

double sinh(*x*)

double cosh(*x*)
double *x*;

double tanh(*x*)
double *x*;

Description

These functions compute the designated hyperbolic functions for real arguments.

Return Value

The `sinh` and `cosh` functions return HUGE_VAL and set *errno* to ERANGE when the correct value would overflow.

Environment

When your program is compiled using the System V environment, `sinh` and `cosh` return HUGE (and `sinh` may return HUGE or negative *x*) when the correct value would overflow and set *errno* to ERANGE.

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

Name

cbrt, sqrt – cube root, square root

Syntax

#include <math.h>

double cbrt(*x*)
double *x*;

double sqrt(*x*)
double *x*;

float fsqrt(float *x*)
float *x*;

Description

The `cbrt` function returns the cube root of *x*.

The `sqrt` and `fsqrt` functions return the square root of *x* for double and float data types respectively.

Error Due to Roundoff and Other Reasons

The `cbrt` function is accurate to within 0.7 *ulp*s.

The `sqrt` function on this machine conforms to IEEE 754 and is correctly rounded in accordance with the rounding mode in force; the error is less than half an *ulp* in the default mode (round–to–nearest). An *ulp* is one *U*nit in the *L*ast *P*lace carried.

Diagnostics

The `sqrt` function returns the default quiet *NaN* when *x* is negative indicating the invalid operation.

See Also

math(3m)

Network Routines (3n)

Insert tabbed divider here.
Then discard this sheet.

Name

intro – introduction to network library functions

Description

This section describes functions that are available for interprocess communication (IPC). IPC takes place using sockets. The `socket`(2) system call creates a communications channel based on domain, type, and protocol.

Sockets are created without names. The `bind`(2) system call is used to connect a name to a socket.

A connection with another process must be made before data can be transferred on a bound socket. The `connect`(2) system call is used to rendezvous with another process. This process must be listening on a bound socket using the `listen`(2) system call. This listening process can accept a connection request using the `accept`(2) system call.

Once two processes have connected and accepted an IPC, data can be transferred with the following system calls: `read`(2); `write`(2); `send`(2), and `recv`(2).

Connectionless sockets are also possible (a socket is bound and data can be transferred). They use the following system calls to transfer data: `sendto` and `recvfrom`.

IPC operates in three domains:

UNIX	Local node
INTERNET	Local area network (LAN)
DECNET	DECnet network

These types of sockets are available for IPC:

stream	Sequenced, reliable, unduplicated data CONNECTED socket record boundaries not preserved all domains
datagram	Not guaranteed to be sequenced, reliable, or unduplicated user protocol needed to give guarantees UNCONNECTED socket record boundaries preserved UNIX and INTERNET domains
sequenced packet	Like stream socket, except record boundaries preserved DECNET domain only
raw	Access to communications protocols

Internet Addresses Routines

The *inet* routines manipulate Internet addresses.

Network Data Base File Routines

Standard mapping routines are used to retrieve entries in network data base files. Several routines operating on each data base file are identified by a group name:

`gethostent`	Retrieves entries from `/etc/hosts`
`getnetent`	Retrieves entries from `/etc/networks`
`getprotoent`	Retrieves entries from `/etc/protocols`
`getservent`	Retrieves entries from `/etc/services`

Specific routines perform particular operations on each data base file:

`get...ent`	Reads the next line of the file; opens the file, if necessary.
`set...ent`	Opens and rewinds the file.
`end...ent`	Closes the file.
`get...byname`	Searches the file sequentially from the beginning until a matching *name* is found, or EOF is encountered.
`get...byaddr`	Searches the file sequentially from the beginning until a matching *address* is found, or EOF is encountered.
`get...byport`	Searches the file sequentially from the beginning until a matching *port number* is found, or EOF is encountered.
`get...bynumber`	Searches the file sequentially from the beginning until a matching *protocol number* is found, or EOF is encountered.

Each network library routine returns a pointer to a structure reflecting individual fields of a line in one of the network data base files. The structure for each data base file contains some of the fields in the following list, with the prefix *x* replaced by a different letter in each file:

`x_addr`	pointer to a network address, returned in network-byte order
`x_addrtype`	address family of the address being returned
`x_aliases`	alternate names
`x_length`	length of an address, in bytes
`x_name`	official name
`x_net`	network number, returned in machine-byte order
`x_port`	resident port
`x_proto`	protocol number

Name

htonl, htons, ntohl, ntohs – convert values between host and network byte order

Syntax

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

netlong = htonl(hostlong);
u_long netlong, hostlong;

netshort = htons(hostshort);
u_short netshort, hostshort;

hostlong = ntohl(netlong);
u_long hostlong, netlong;

hostshort = ntohs(netshort);
u_short hostshort, netshort;

Description

These routines convert 16 and 32 bit quantities between network byte order and host byte order. These routines are defined as null macros in the include file <netinet/in.h>.

These routines are most often used in conjunction with Internet addresses and ports as returned by `gethostbyname`(3n) and `getservent`(3n).

See Also

gethostbyname(3n), getservent(3n)

byteorder (3n)

Name

htonl, htons, ntohl, ntohs – convert values between host and network byte order

Syntax

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

netlong = htonl(hostlong);
u_long netlong, hostlong;

netshort = htons(hostshort);
u_short netshort, hostshort;

hostlong = ntohl(netlong);
u_long hostlong, netlong;

hostshort = ntohs(netshort);
u_short hostshort, netshort;

Description

These routines convert 16-bit and 32-bit quantities between network byte order and host byte order. On some non-ULTRIX machines these routines are defined as null macros in the include file <netinet/in.h>.

These routines are most often used with Internet addresses and ports as returned by `gethostent`(3n) and `getservent`(3n).

Restrictions

The VAX handles bytes in the reverse from most everyone else.

See Also

gethostent(3n), getservent(3n)

Name

gethostent, gethostbyaddr, gethostbyname, sethostent, endhostent – get hosts entry

Syntax

#include <netdb.h>

struct hostent *gethostent()

struct hostent *gethostbyname(*name***)**
char ***name*;**

struct hostent *gethostbyaddr(*addr***,** *len***,** *type***)**
char ***addr*; int** *len***,** *type***;**

sethostent(*stayopen***)**
int *stayopen***;**

endhostent()

Description

The `gethostent`, `gethostbyname`, and `gethostbyaddr` subroutines return a pointer to an object with the following structure containing the broken-out fields reflecting information obtained from the `hosts` database.

```
struct   hostent {
         char   *h_name;        /* official name of host */
         char   **h_aliases;    /* alias list */
         int    h_addrtype;     /* address type */
         int    h_length;       /* length of address */
         char   **h_addr_list; /* list of addresses from name server */
#define  h_addr h_addr_list[0] /* address for backward compatibility */
};
```

The members of this structure are:

h_name — Official name of the host.

h_aliases — A zero terminated array of alternate names for the host.

h_addrtype — The type of address being returned; currently always AF_INET.

h_length — The length, in bytes, of the address.

h_addr — A pointer to the network address for the host. Host addresses are returned in network byte order.

If the *stayopen* flag on a `sethostent` subroutine is NULL, the hosts database is opened. Otherwise the `sethostent` has the effect of rewinding the `hosts` database. The `endhostent` may be called to close the `hosts` database when processing is complete.

The `gethostent` subroutine simply reads the next line while `gethostbyname` and `gethostbyaddr` search until a matching *name,* or *addr, len, type* is found (or until EOF is encountered). The `gethostent` subroutine keeps a pointer in the database, allowing successive calls to be used to search the entire file.

The `gethostbyname` and `gethostbyaddr` subroutines query the `hosts` database.

A call to `sethostent` must be made before a `while` loop using `gethostent` in order to perform initialization and an `endhostent` must be used after the loop. Both `gethostbyname` and `gethostbyaddr` make calls to `sethostent` and `endhostent.`

Restrictions

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet address format is currently understood.

If YP is running, `gethostent` does not return the entries in any particular order. See the *Guide to the Yellow Pages Service* for setup information.

The `hosts` database may also be distributed via the BIND/Hesiod naming service. See the *Guide to the BIND/Hesiod Service* for more information.

Return Value

Null pointer (0) returned on EOF or error.

Files

`/etc/hosts`

See Also

hosts(5), svc.conf(5)
Guide to the BIND/Hesiod Service
Guide to the Yellow Pages Service

Name

getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent – get networks entry

Syntax

#include <netdb.h>

struct netent *getnetent()

struct netent *getnetbyname(*name***)**
char ****name*;**

struct netent *getnetbyaddr(*net***,** *type***)**
long *net***; int** *type***;**

setnetent(*stayopen***)**
int *stayopen***;**

endnetent()

Description

The `getnetent`, `getnetbyname`, and `getnetbyaddr` subroutines each return a pointer to an object with the following structure containing the broken-out fields of a line in the `networks` database.

```
struct netent {
       char   *n_name;      /* official name of net */
       char   **n_aliases;  /* alias list */
       int    n_addrtype;   /* net number type */
       long   n_net;        /* net number */
};
```

The members of this structure are:

n_name The official name of the network.

n_aliases A zero terminated list of alternate names for the network.

n_addrtype The type of the network number returned: AF_INET.

n_net The network number. Network numbers are returned in machine byte order.

If the *stayopen* flag on a `setnetent` subroutine is NULL, the `networks` database is opened. Otherwise the `setnetent` has the effect of rewinding the `networks` database. The `endnetent` may be called to close the `networks` database when processing is complete.

The `getnetent` subroutine simply reads the next line while `getnetbyname` and `getnetbyaddr` search until a matching *name* or *net* number is found (or until EOF is encountered). The *type* must be AF_INET. The `getnetent` subroutine keeps a pointer in the database, allowing successive calls to be used to search the entire file.

A call to `setnetent` must be made before a `while` loop using `getnetent` in order to perform initialization and an `endnetent` must be used after the loop. Both `getnetbyname` and `getnetbyaddr` make calls to `setnetent` and `endnetent`.

Restrictions

All information is contained in a static area so it must be copied if it is to be saved. Only Internet network numbers are currently understood.

If YP is running, `getnetent` does not return the entries in any particular order. See the *Guide to the Yellow Pages Service* for setup information.

The networks database may also be distributed via the BIND/Hesiod naming service. See the *Guide to the BIND/Hesiod Service* for more information.

Return Value

Null pointer (0) returned on EOF or error.

Files

`/etc/networks`

See Also

networks(5), svc.conf(5)
Guide to the BIND/Hesiod Service
Guide to the Yellow Pages Service

Name

getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent – get protocols entry

Syntax

#include <netdb.h>

struct protoent *getprotoent()

struct protoent *getprotobyname(*name***)**
char ***name*;**

struct protoent *getprotobynumber(*proto***)**
int *proto***;**

setprotoent(*stayopen***)**
int *stayopen***;**

endprotoent()

Description

The `getprotoent`, `getprotobyname`, and `getprotobynumber` subroutines each return a pointer to an object with the following structure containing the broken-out fields of a line in the `protocols` database.

```
struct protoent {
       char   *p_name;      /* official name of protocol */
       char   **p_aliases;  /* alias list */
       long   p_proto;      /* protocol number */
};
```

The members of this structure are:

p_name The official name of the protocol.

p_aliases A zero terminated list of alternate names for the protocol.

p_proto The protocol number.

If the *stayopen* flag on a `setprotoent` subroutine is NULL, the protocols database is opened. Otherwise the `setprotoent` has the effect of rewinding the protocols database. The `endprotoent` may be called to close the protocols database when processing is complete.

The `getprotoent` subroutine simply reads the next line while `getprotobyname` and `getprotobynumber` search until a matching *name* or *proto* number is found (or until EOF is encountered). The `getprotoent` subroutine keeps a pointer in the database, allowing successive calls to be used to search the entire file.

A call to `setprotoent` must be made before a `while` loop using `getprotoent` in order to perform initialization and an `endprotoent` must be used after the loop. Both `getprotobyname` and `getprotobynumber` make calls to `setprotoent` and `endprotoent`.

Restrictions

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet protocols are currently understood.

If YP is running, `getprotoent` does not return the entries in any particular order. See the *Guide to the Yellow Pages Service* for setup information.

The services database may also be distributed using the BIND/Hesiod naming service. See the *Guide to the BIND/Hesiod Service* for more information.

Return Value

Null pointer (0) returned on EOF or error.

Files

`/etc/protocols`

See Also

protocols(5), svc.conf(5)
Guide to the BIND/Hesiod Service
Guide to the Yellow Pages Service

Name

getservent, getservbyname, getservbyport, setservent, endservent – get services entry

Syntax

#include <netdb.h>

struct servent *getservent()

struct servent *getservbyname*(name, proto)*
char **name*, **proto*;

struct servent *getservbyport*(port, proto)*
int *port*; **char** **proto*;

setservent*(stayopen)*
int *stayopen*

endservent()

Description

The `getservent`, `getservbyname`, and `getservbyport` subroutines each return a pointer to an object with the following structure containing the broken-out fields of a line in the network `services` database.

```
struct servent {
       char   *s_name;         /* official name of service */
       char   **s_aliases;     /* alias list */
       long   s_port;          /* port service resides at */
       char   *s_proto;        /* protocol to use */
};
```

The members of this structure are:

s_name The official name of the service.

s_aliases A zero terminated list of alternate names for the service.

s_port The port number at which the service resides. Port numbers are returned in network byte order.

s_proto The name of the protocol to use when contacting the service.

If the *stayopen* flag on a `setservent` subroutine is NULL, the `services` database is opened. Otherwise, the `setservent` has the effect of rewinding the `services` database. The `endservent` subroutine may be called to close the `services` database when processing is complete.

The `getservent` subroutine reads the next line; `getservbyname` and `getservbyport` search until a matching *name* or *port* is found (or until EOF is encountered). The `getservent` subroutine keeps a pointer in the database, allowing successive calls to be used to search the entire file. If a non-NULL protocol name, `proto`, is also supplied, searches must also match the protocol.

The `setservent` routine must be called before a `while` loop that uses `getservent` in order to initialize variables in the `setservent` routine and an `endservent` must be used after the loop. Both `getservbyport` and `getservbyname` make calls to `setservent` and `endservent`.

Restrictions

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

If the Yellow Pages Service is running, `getservent` does not return the entries in any particular order. See the *Guide to the Yellow Pages Service* for setup information.

The `services` database can also be distributed by the BIND/Hesiod naming service. See the *Guide to the BIND/Hesiod Service* for more information.

Return Value

Null pointer (0) returned on EOF or error.

Files

`/etc/services`

See Also

services(5), svc.conf(5)
Guide to the BIND/Hesiod Service
Guide to the Yellow Pages Service

Name

inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof – Internet address manipulation routines

Syntax

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_addr(cp)
char *cp;

unsigned long inet_network(cp)
char *cp;

char *inet_ntoa(in)
struct in_addr in;

struct in_addr inet_makeaddr(net, lna)
int net, lna;

int inet_lnaof(in)
struct in_addr in;

int inet_netof(in)
struct in_addr in;

Description

The routines `inet_addr` and `inet_network` each interpret character strings representing numbers expressed in the Internet standard ''.'' notation, returning numbers suitable for use as Internet addresses and Internet network numbers, respectively. The routine `inet_ntoa` takes an Internet address and returns an ASCII string representing the address in ''.'' notation. The routine `inet_makeaddr` takes an Internet network number and a local network address and constructs an Internet address from it. The routines `inet_netof` and `inet_lnaof` break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet address 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.

Internet Addresses

Values specified using the ''.'' 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. Note that when an Internet address is viewed as a 32-bit integer quantity on the VAX, the bytes referred to above appear as ''d.c.b.a''. That is, VAX bytes are ordered from right to left.

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 ''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 ''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 a ''.'' notation may be decimal, octal, or hexadecimal, as specified in the C language (i.e. a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).

Return Value

The value –1 is returned by `inet_addr` and `inet_network` for malformed requests.

See Also

gethostent(3n), getnetent(3n), hosts(5), networks(5)

Name

snmpextregister, snmpextgetreq, snmpextrespond, snmpexterror – library routines available for building the Extended ULTRIX SNMP Agent (Extended Agent)

Syntax

```
#include <protocols/snmp.h>
#include <protocols/snmperrs.h>

struct objident {
        short           ncmp;                /* number of components */
        unsigned long   cmp[SNMPMXID];       /* components */
};

struct snmpareg {
        short           oidtype;             /* object id type */
        objident        oid;                 /* object id/*
};

struct snmparspdat {
        short           type;                /* response data type */
        short           octets;              /* number of octets in response data */
        char            *rspdat;             /* response data */
};

snmpextregister(reg, community)
struct snmpareg *reg;
char *community;

snmpextgetreq(reqoid, reqinst)
objident *reqoid;
objident *reqinst;

snmpextrespond(reqoid, rspinst, rspdat)
objident *reqoid;
objident *rspinst;
struct snmparspdat *rspdat;

snmpexterror(error)
long error;
```

Description

The following library routines are available for building the Extended Agent:

snmpextregister
Used to register the Extended Agent's Management Information Base (MIB) to the ULTRIX SNMP Agent (Agent). The *reg* parameter is provided by the caller with the object identifiers to be registered. The *community* parameter is provided by the caller with the community name (a null-terminated string).

This library routine waits for a registration confirmation from the Agent. The process is blocked until the confirmation arrives. When the confirmation arrives, the routine returns the status of the registration.

The program issues this call before any other Extended SNMP Library calls. It does this because the `snmpextregister` library routine creates a UNIX domain socket to the Agent on behalf of the caller.

snmpextgetreq
Used to receive a request for a MIB variable from the Agent. If there is no outstanding request from the Agent, the process is blocked until a request arrives from the Agent.

When the Extended Agent receives a request from the Agent, the *reqoid* parameter contains the object identifier for the requested variable. The *reqinst* parameter contains the object instance identifier for the requested variable. If the request does not contains an object instance, the *reqinst->ncmp* record contains a zero.

snmpextrespond
Used to return the requested variable to the Agent. The *reqoid* parameter is the object identifier from the `snmpextgetreq` library call. The *rspinst* parameter is the object instance associated with the returning variable. If there is no object instance associated with the returning variable, a null parameter must be supplied. The *rspdat* parameter is the returning variable.

Note that the Agent maintains a configurable timer for outstanding requests to the Extended Agent. Therefore, the Extended Agent must be able to respond within the Agent's timeout interval in order to prevent a premature timeout in the Agent.

See the `/etc/snmpd.conf` file for your system's default timeout value.

snmpexterror
Used to return an error to the Agent. The *error* parameter is the error code to be returned to the Agent. The error code is one of the following:

NOERR—successful SNMP *get-next-request end-of-table*. This happens when the requested instance does not exist.

NOSUCH—Unknown requested object identifier.

GENERRS—Generic error.

BADVAL—Bad variable value.

Restrictions

For the `snmpextregister` routine, the object identifier must have the prefix 1.3.6.1 to be registered. If it does not, the registration is rejected.

Return Value

If an error occurs, a negative value is returned.

Diagnostics

[BADVERSION]	Bad or obsolete protocol version
[BINDERR]	Failed to bind the socket
[GENSUC]	MIB successfully registered
[NOSOCK]	Socket does not exist
[NOSVC]	MIB registration was rejected
[PKTLENERR]	Maximum size message exceeded or community name is too large
[RCV_ERR]	Reception failed
[SND_ERR]	Transmission failed

Files

`/etc/snmpd.conf` SNMP configuration file

See Also

snmpd.conf(5n), snmpd(8n), snmpsetup(8n)
Guide to Network Programming

Network Computing System Routines (3ncs)

Insert tabbed divider here.
Then discard this sheet.

Name

intro – introduction to the Network Computing System's (NCS) library routines

Description

This section describes the NCS library routines.

NOTE

The Title, Name, and See Also sections of the NCS reference pages do not contain the dollar ($) sign in the command names and library routines. The actual NCS commands and library routines do contain the dollar ($) sign.

The NCS commands and library routines are as follows:

- Error Text Database Operations (`error_$`)
- Interface to the Location Broker (`lb_$`)
- Fault Management (`pfm_$`)
- Program Management (`pgm_$`)
- Interface to the Remote Procedure Call Runtime Library (`rpc_$`)
- Remote Remote Procedure Call Interface (`rrpc_$`)
- Operations on Socket Addresses (`socket_$`)
- Operations on Universal Unique Identifiers (`uuid_$`)

Error Text Database Operations

The error text database operations use the `error_$c_get_text` and `error_$c_text` library routines to convert status codes into textual error messages. The runtime library reports operational problems back to the application following a call by setting the 'all' field of the **status_$t** structure. A value of **status_$ok** indicates that no errors were detected. Any other value implies that a problem occurred. The **status_$t** structure and the `error_$` routines can be used to display a textual representation of the error condition.

Data Types

This section describes the data types used in `error_$` routines.

The `error_$` routines take as input a status code in **status_$t** format.

status_$t A status code. Most of the NCS routines supply their completion status in this format. The **status_$t** type is defined as a structure containing a long integer:

```
struct status_$t {
   long all;
}
```

However, the routines can also use **status_$t** as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
 struct {
     unsigned fail : 1,
            subsys : 7,
            modc : 8;
     short    code;
 } s;
 long all;
} status_u;
```

all All 32 bits in the status code. If **all** is equal to **status_$ok**, the routine that supplied the status was successful.

fail If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.

subsys
This indicates the subsystem that encountered the error.

modc
This indicates the module that encountered the error.

code
This is a signed number that identifies the type of error that occurred.

Interface To The Location Broker

The `lb_$` library routines implement the programmatic interface to the Location Broker Client Agent. The file `/usr/include/idl/c/glb.h` defines this interface.

External Variables

This section describes the external variable used in `lb_$` routines.

uuid_$nil An external **uuid_$t** variable that is preassigned the value of the nil UUID. Do not change the value of this variable.

Constants

This section describes constants used in `lb_$` routines.

lb_$default_lookup_handle
Used as an input in Location Broker lookup routines. Specifies that a lookup is to start searching at the beginning of the database.

lb_$server_flag_local Used in the **flags** field of an **lb_$entry_t** variable. Specifies that an entry is to be registered only in the Local Location Broker (LLB) database. See the description of **lb_$server_flag_t** in the Data Types section.

status_$ok A constant used to check status. If a completion status is equal to **status_$ok**, then the routine that supplied it was successful.

Data Types

This section describes data types used in `lb_$` routines.

lb_$entry_t An identifier for an object, a type, an interface, and the socket address used to access a server exporting the interface to the object. The **lb_$entry_t** type is defined as follows:

```
typedef struct lb_$entry_t lb_$entry_t;
struct lb_$entry_t {
    uuid_$t object;
    uuid_$t obj_type;
    uuid_$t obj_interface;
    lb_$server_flag_t flags;
    ndr_$char annotation[64];
    ndr_$ulong_int saddr_len;
    socket_$addr_t saddr;
};
```

object A **uuid_$t**. The UUID for the object. Can be **uuid_$nil** if no object is associated.

obj_type A **uuid_$t**. The UUID for the type of the object. Can be **uuid_$nil** if no type is associated.

obj_interface A **uuid_$t**. The UUID for the interface. Can be **uuid_$nil** if no interface is associated.

flags An **lb_$server_flag_t**. Must be 0 or **lb_$server_flag_local**. A value of 0 specifies that the entry is to be registered in both the Local Location Broker (LLB) and global Location Broker (GLB) databases. A value of **lb_$server_flag_local** specifies registration only in the LLB database.

annotation A 64-character array. User-defined textual annotation.

saddr_len A 32-bit integer. The length of the **saddr** field.

saddr A **socket_$addr_t**. The socket address of the server.

lb_$lookup_handle_t A 32-bit integer used to specify the location in the database at which a Location Broker lookup operation will start.

lb_$server_flag_t A 32-bit integer used to specify the Location Broker databases in which an entry is to be registered. A value of 0 specifies registration in both the Local Location Broker (LLB) and Global Location Broker (GLB) databases. A value of **lb_$server_flag_local** specifies registration only in the LLB database.

socket_$addr_t A socket address record that uniquely identifies a socket.

status_$t A status code. Most of the NCS routines supply a completion code in this format. The **status_$t** type is defined as a structure containing a long integer:

```
struct status_$t {
    long all;
    }
```

However, the system calls can also use **status_$t** as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
    struct {
        unsigned fail : 1,
                 subsys : 7,
                 modc : 8;
        short    code;
    } s;
    long all;
} status_u;
```

all All 32 bits in the status code. If **all** is equal to **status_$ok**, the system call that supplied the status was successful.

fail If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.

subsys This indicates the subsystem that encountered the error.

modc This indicates the module that encountered the error.

code This is a signed number that identifies the type of error that occurred.

uuid_$t A 128-bit value that uniquely identifies an object, type, or interface for all time.

Example

The following statement looks up information in the GLB database about a matrix multiplication interface:

```
lb_$lookup_interface (&matrix_id, &lookup_handle, max_results,
    &num_results, &matrix_results, &st);
```

Fault Management

The `pfm_$` routines allow programs to manage signals, faults, and exceptions by establishing clean-up handlers.

A clean-up handler is a piece of code that ensures a program terminates gracefully when it receives a fatal error. A clean-up handler begins with a `pfm_$cleanup` call, and usually ends with a call to `pfm_$signal` or `pgm_$exit`, though it can also simply continue back into the program after the clean-up code.

A clean-up handler is not entered until all fault handlers established for a fault have returned. If there is more than one established clean-up handler for a program, the most recently established clean-up handler is entered first, followed by the next most recently established clean-up handler, and so on to the first established clean-up handler if necessary.

There is a default clean-up handler invoked after all user-defined handlers have completed. It releases any resources still held by the program, before returning control to the process that invoked it.

Constants

pfm_$init_signal_handlers
A constant used as the *flags* parameter to `pfm_$init`, causing C signals to be intercepted and converted to PFM signals.

Data Types

This section describes the data typed used in pfm_$ routines.

pfm_$cleanup_rec A record type for passing process context among clean-up handler routines. It is an opaque data type.

status_$t A status code. Most of the NCS routines supply a completion code in this format. The **status_$t** type is defined as a structure containing a long integer:

```
struct status_$t {
    long all;
    }
```

However, the system calls can also use **status_$t** as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
    struct {
        unsigned fail : 1,
                 subsys : 7,
                 modc : 8;
        short    code;
    } s;
    long all;
} status_u;
```

all All 32 bits in the status code. If **all** is equal to **status_$ok**, the system call that supplied the status was successful.

fail If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.

subsys This indicates the subsystem that encountered the error.

modc	This indicates the module that encountered the error.
code	This is a signed number that identifies the type of error that occurred.

Program Management

The NCS software products contain a portable version of the `pgm_$exit` routine. The include file for the PFM interface (see the Syntax section of the `pfm`(3ncs) reference pages) contains a declaration for this routine.

Interface To The Remote Procedure Call

The `rpc_$` library routines implement the NCS Remote Procedure Call (RPC) mechanism.

The `rpc_` interface is defined by the file `/usr/include/idl/rpc.idl.`

Most of the `rpc_$` routines can be used only by clients or only by servers. This aspect of their usage is specified at the beginning of each routine description, in the Name section.

External Variables
This section describes the external variable used in **rpc_$** routines.

uuid_$nil	An external **uuid_$t** variable that is preassigned the value of the nil UUID. Do not change the value of this variable.

Constants
This section describes constants used in **rpc_$** routines.

rpc_$mod	A module code indicating the RPC module.
status_$ok	A constant used to check status. If a completion status is equal to **status_$ok**, then the routine that supplied it was successful. See the description of the **status_$t** type.
rpc_$unbound_port	A port number indicating to the RPC runtime library that no port is specified. Identical to **socket_$unspec_port**.

The following 16-bit-integer constants are used to specify the communications protocol address families in **socket_$addr_t** structures. Note that several of the **rpc_$** and **socket_$** calls use the 32-bit-integer equivalents of these values.

socket_$unspec	Address family is unspecified.
socket_$internet	Internet Protocols (IP).

Data Types
This section describes data types used in **rpc_$** routines.

handle_t	An RPC handle.
rpc_$epv_t	An entry point vector (EPV). An array of **rpc_$server_stub_t**, pointers to server stub procedures.
rpc_$generic_epv_t	An entry point vector (EPV). An array of **rpc_$generic_server_stub_t**, pointers to generic server stub procedures.

rpc_$if_spec_t An RPC interface specifier. This opaque data type contains information about an interface, including its UUID, the current version number, any well-known ports used by servers that export the interface, and the number of operations in the interface.

rpc_$mgr_epv_t An entry point vector (EPV). An array of pointers to manager procedures.

rpc_$shut_check_fn_t A pointer to a function. If a server supplies this function pointer to **rpc_$allow_remote_shutdown**, the function will be called when a remote shutdown request arrives, and if the function returns true, the shutdown is allowed. The following C definition for **rpc_$shut_check_fn_t** illustrates the prototype for this function:

```
typedef boolean (*rpc_$shut_check_fn_t) (
    handle_t h,
    status_$t *st)
```

The handle argument can be used to determine information about the remote caller.

socket_$addr_t A socket address record that uniquely identifies a socket.

status_$t A status code. Most of the NCS system calls supply their completion status in this format. The **status_$t** type is defined as a structure containing a long integer:

```
struct status_$t {
    long all;
    }
```

However, the system calls can also use **status_$t** as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
    struct {
        unsigned fail : 1,
                 subsys : 7,
                 modc : 8;
        short    code;
    } s;
    long all;
} status_u;
```

all All 32 bits in the status code. If **all** is equal to **status_$ok**, the system call that supplied the status was successful.

fail If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.

subsys This indicates the subsystem that encountered the error.

modc — This indicates the module that encountered the error.

code — This is a signed number that identifies the type of error that occurred.

uuid_$t — A 128-bit value that uniquely identifies an object, type, or interface for all time.

The following statement allocates a handle that identifies the Acme company's payroll database object:

```
h = rpc_$alloc_handle (&acme_pay_id, socket_$internet, &st);
```

Remote Remote Procedure Call Interface

The `rrpc_$` library routines enable a client to request information about a server or to shut down a server.

The `rrpc_` interface is defined by the file `/usr/include/idl/rrpc.idl.`

Constants

This section describes constants used in **rrpc_$** calls.

The **rrpc_$sv** constants are indices for elements in an **rrpc_$stat_vec_t** array. Each element is a 32-bit integer representing a statistic about a server. The following list describes the statistic indexed by each **rrpc_$sv** constant:

rrpc_$sv_calls_in — The number of calls processed by the server.

rrpc_$sv_rcvd — The number of packets received by the server.

rrpc_$sv_sent — The number of packets sent by the server.

rrpc_$sv_calls_out — The number of calls made by the server.

rrpc_$sv_frag_resends — The number of fragments sent by the server that duplicated previous sends.

rrpc_$sv_dup_frags_rcvd — The number of duplicate fragments received by the server.

status_$ok — A constant used to check status. If a completion status is equal to **status_$ok**, then the system call that supplied it was successful.

Data Types

This section describes data types used in **rpc_$** routines.

handle_t — An RPC handle.

rrpc_$interface_vec_t — An array of **rpc_$if_spec_t**, RPC interface specifiers.

rrpc_$stat_vec_t — An array of 32-bit integers, indexed by **rrpc_$sv** constants, representing statistics about a server.

rpc_$if_spec_t — An RPC interface specifier. An opaque data type containing information about an interface, including the UUID, the version number, the number of operations in the interface, and any well-known ports used by servers that export the interface, and any well-known ports used by servers that export the interface. Applications may need to access two members of **rpc_$if_spec_t**:

id A **uuid_$t** indicating the interface UUID.

vers An unsigned 32-bit integer indicating the interface version.

Operations on Socket Addresses

The `socket_$` library routines manipulate socket addresses. Unlike the routines that operating systems such as BSD UNIX provide, the `socket_$` routines operate on addresses of any protocol family.

The file `/usr/include/idl/socket.idl` defines the `socket_` interface.

Constants

This section describes constants used in **socket_$** routines.

The **socket_$eq** constants are flags indicating the fields to be compared in a **socket_$equal** call.

socket_$eq_hostid Indicates that the host IDs are to be compared.

socket_$eq_netaddr Indicates that the network addresses are to be compared.

socket_$eq_port Indicates that the port numbers are to be compared.

socket_$eq_network Indicates that the network IDs are to be compared.

socket_$unspec_port A port number indicating to the RPC runtime library that no port is specified.

The following 16-bit-integer constants are values for the **socket_$addr_family_t** type, used to specify the address family in a **socket_$addr_t** structure. Note that several of the **rpc_$** and **socket_$** routines use the 32-bit-integer equivalents of these values.

socket_$unspec Address family is unspecified.

socket_$internet Internet Protocols (IP).

status_$ok A constant used to check status. If a completion status is equal to **status_$ok**, then the system call that supplied it was successful.

Data Types

This section describes data types used in **socket_$** routines.

socket_$addr_family_t
An enumerated type for specifying an address family. The Constants section lists values for this type.

socket_$addr_list_t An array of socket addresses in **socket_$addr_t** format.

socket_$addr_t
A structure that uniquely identifies a socket address. This structure consists of a **socket_$addr_family_t** specifying an address family and 14 bytes specifying a socket address.

socket_$host_id_t
A structure that uniquely identifies a host. This structure consists of a **socket_$addr_family_t** specifying an address family and 12 bytes specifying a host.

socket_$len_list_t
An array of unsigned 32-bit integers, the lengths of socket addresses in a **socket_$addr_list_t**.

socket_$local_sockaddr_t
An array of 50 characters, used to store a socket address in a format native to the local host.

socket_$net_addr_t
A structure that uniquely identifies a network address. This structure consists of a **socket_$addr_family_t** specifying an address family and 12 bytes specifying a network address. It contains both a host ID and a network ID.

socket_$string_t
An array of 100 characters, used to store the string representation of an address family or a socket address.

The string representation of an address family is a textual name such as **dds**, **ip**, or **unspec**.

The string representation of a socket address has the format *family*:*host*[*port*], where *family* is the textual name of an address family, *host* is either a textual host name or a numeric host ID preceded by a #, and *port* is a port number.

status_$t
A status code. Most of the NCS system calls supply their completion status in this format. The **status_$t** type is defined as a structure containing a long integer:

```
struct status_$t {
    long all;
    }
```

However, the system calls can also use **status_$t** as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
    struct {
        unsigned fail : 1,
                 subsys : 7,
                 modc : 8;
        short    code;
    } s;
    long all;
} status_u;
```

all
All 32 bits in the status code. If **all** is equal to **status_$ok**, the system call that supplied the status was successful.

fail	If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.
subsys	This indicates the subsystem that encountered the error.
modc	This indicates the module that encountered the error.
code	This is a signed number that identifies the type of error that occurred.

Operations On Universal Unique Identifiers

The `uuid_$` library routines operate on UUIDs (Universal Unique Identifiers).

The `uuid_` interface is defined by the file `/usr/include/idl/uuid.idl`.

The completion status. `/usr/include/idl/uuid.idl`

External Variables

This section describes external variables used in **uuid_$** routines.

uuid_$nil

An external **uuid_$t** variable that is preassigned the value of the nil UUID. Do not change the value of this variable.

Data Types

This section describes data types used in **uuid_$** routines.

status_$t A status code. Most of the NCS system calls supply their completion status in this format. The **status_$t** type is defined as a structure containing a long integer:

```
struct status_$t {
    long all;
    }
```

However, the system calls can also use **status_$t** as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
    struct {
        unsigned fail : 1,
                 subsys : 7,
                 modc : 8;
        short    code;
    } s;
    long all;
} status_u;
```

all	All 32 bits in the status code. If **all** is equal to **status_$ok**, the system call that supplied the status was successful.
fail	If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.

subsys This indicates the subsystem that encountered the error.

modc This indicates the module that encountered the error.

code This is a signed number that identifies the type of error that occurred.

uuid_$string_t
A string of 37 characters (including a null terminator) that is an ASCII representation of a UUID. The format is *cccccccccccc.ff.h1.h2.h3.h4.h5.h6.h7*, where *cccccccccccc* is the timestamp, *ff* is the address family, and *h1 ... h7* are the 7 bytes of host identifier. Each character in these fields is a hexadecimal digit.

uuid_$t A 128-bit value that uniquely identifies an object, type, or interface for all time. The **uuid_$t** type is defined as follows:

```
typedef struct uuid_$t {
    unsigned long time_high;
    unsigned short time_low;
    unsigned short reserved;
    unsigned char family;
    unsigned char (host)[7];
} uuid_$t;
```

time_high
The high 32 bits of a 48-bit unsigned time value which is the number of 4-microsecond intervals that have passed between 1 January 1980 00:00 GMT and the time of UUID creation.

time_low
The low 16 bits of the 48-bit time value.

reserved
16 bits of reserved space.

family
8 bits identifying an address family.

host 7 bytes identifying the host on which the UUID was created. The format of this field depends on the address family.

Example

The following routine returns as `foo_uuid` the UUID corresponding to the character-string representation in `foo_uuid_rep`:

```
uuid_$decode (foo_uuid_rep, &foo_uuid, &status);
```

Name

error_c_get_text – return subsystem, module, and error texts for a status code

Syntax

void error_$c_get_text(*status, subsys, subsysmax, module, modulemax,*
error, errormax)
status_$t *status*;
char **subsys*;
long *subsysmax*;
char **module*;
long *modulemax*;
char **error*;
long *errormax*;

Arguments

status	A status code in **status_$t** format.
subsys	A character string. The subsystem represented by the status code.
subsysmax	The maximum number of bytes to be returned in *subsys*.
module	A character string. The module represented by the status code.
modulemax	The maximum number of bytes to be returned in *module*.
error	A character string. The error represented by the status code.
errormax	The maximum number of bytes to be returned in *error*.

Description

The `error_$c_get_text` routine returns predefined text strings that describe the subsystem, the module, and the error represented by a status code. The strings are null terminated. See the `intro`(3ncs) reference page which lists all of the possible diagnostics that could be returned in `status.all`.

Files

`/usr/lib/stcode.db`

See Also

intro(3ncs)

error_c_text (3ncs)

Name

error_c_text – return an error message for a status code

Syntax

void error_$c_text(*status, message, messagemax*)
status_$t *status*;
char **message*;
int *messagemax*;

Arguments

status	A status code in **status_$t** format.
message	A character string. The error message represented by the status code.
messagemax	The maximum number of bytes to be returned in *message*.

Description

The `error_$c_text` routine returns a null terminated error message for reporting the completion status of a routine. The error message is composed from predefined text strings that describe the subsystem, the module, and the error represented by the status code. See the `intro`(3ncs) reference page which lists all of the possible diagnostics that could be returned in `status.all`.

Files

`/usr/lib/stcode.db`

See Also

intro(3ncs)

Name

lb_lookup_interface – look up information about an interface in the Global Location Broker database

Syntax

#include <idl/c/lb.h>

void lb_$lookup_interface(*obj_interface, lookup_handle, max_num_results, num_results, results, status*)
uuid_$t **obj_interface*;
lb_$lookup_handle_t **lookup_handle*;
unsigned long *max_num_results*;
unsigned long * *num_results*;
lb_$entry_t *results*[ ];
*status_$t *status;*

Arguments

obj_interface	The UUID of the interface being looked up.
lookup_handle	A location in the database. On input, the *lookup_handle* indicates the location in the database where the search begins. An input value of **lb_$default_lookup_handle** specifies that the search will start at the beginning of the database. On return, the *lookup_handle* indicates the next unsearched part of the database (that is, the point at which the next search should begin). A return value of **lb_$default_lookup_handle** indicates that the search reached the end of the database; any other return value indicates that the search found at most *max_num_results* matching entries before it reached the end of the database.
max_num_results	The maximum number of entries that can be returned by a single routine. This should be the number of elements in the *results* array.
num_results	The number of entries that were returned in the *results* array.
results	An array that contains the matching GLB database entries, up to the number specified by the *max_num_results* parameter. If the array contains any entries for servers on the local network, those entries appear first.
status	The completion status.

Description

The `lb_$lookup_interface` routine returns GLB database entries whose *obj_interface* fields match the specified interface. It returns information about objects that can be accessed through that interface.

The `lb_$lookup_interface` routine cannot return more than *max_num_results* matching entries at a time. The *lookup_handle* parameter enables you to find all matching entries by doing sequential lookups.

If you use a sequence of lookup routines to find entries in the database, it is possible that the returned results will skip or duplicate entries. This is because the Location Broker does not prevent modification of the database between lookups, and such modification can change the locations of entries relative to a *lookup_handle* value.

It is also possible that the results of a single lookup routine will skip or duplicate entries. This can occur if the size of the results exceeds the size of an RPC packet (64K bytes).

Examples

The following statement looks up information in the GLB database about a matrix multiplication interface:

```
lb_$lookup_interface (&matrix_id, &lookup_handle, max_results,
                          &num_results, &matrix_results, &st);
```

Diagnostics

This section lists status codes for errors returned by this `lb_$` routine.

lb_$database_invalid The format of the Location Broker database is out of date. The database may have been created by an old version of the Location Broker; in this case, delete the out-of-date database and reregister any entries that it contained. The LLB or GLB that was accessed may be running out-of-date software; in this case, update all Location Brokers to the current software version.

lb_$database_busy The Location Broker database is currently in use in an incompatible manner.

lb_$not_registered The Location Broker does not have any entries that match the criteria specified in the lookup or unregister routine. The requested object, type, interface, or combination thereof is not registered in the specified database. If you are using an `lb_$lookup_object_local` or `lb_$lookup_range` routine specifying an LLB, check that you have specified the correct LLB.

lb_$cant_access The Location Broker cannot access the database. Among the possible reasons:

1. The database does not exist.

2. The database exists, but the Location Broker cannot access it.

lb_$server_unavailable
The Location Broker Client Agent cannot reach the requested GLB or LLB. A communications failure occurred or the broker was not running.

Files

/usr/include/idl/c/glb.h

See Also

intro(3ncs), lb_lookup_object(3ncs), lb_lookup_range(3ncs), lb_lookup_type(3ncs)

lb_lookup_object(3ncs)

Name

lb_lookup_object – look up information about an object in the Global Location Broker database

Syntax

#include <idl/c/lb.h>

void lb_$lookup_object(*object, lookup_handle, max_num_results, num_results, results, status)*
*uuid_$t *object;*
*lb_$lookup_handle_t *lookup_handle;*
unsigned long max_num_results;
*unsigned long * num_results;*
lb_$entry_t results[ ];
*status_$t *status;*

Arguments

object The UUID of the object being looked up.

lookup_handle A location in the database.

On input, the *lookup_handle* indicates the location in the database where the search begins. An input value of **lb_$default_lookup_handle** specifies that the search will start at the beginning of the database.

On return, the *lookup_handle* indicates the next unsearched part of the database (that is, the point at which the next search should begin). A return value of **lb_$default_lookup_handle** indicates that the search reached the end of the database; any other return value indicates that the search found at most *max_num_results* matching entries before it reached the end of the database.

max_num_results The maximum number of entries that can be returned by a single routine. This should be the number of elements in the *results* array.

num_results The number of entries that were returned in the *results* array.

results An array that contains the matching GLB database entries, up to the number specified by the *max_num_results* parameter. If the array contains any entries for servers on the local network, those entries appear first.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `lb_$lookup_object` routine returns GLB database entries whose *object* field matches the specified object UUID.

The `lb_$lookup_object` routine cannot return more than *max_num_results* matching entries at a time. The *lookup_handle* parameter enables you to find all matching entries by doing sequential lookups.

If you use a sequence of lookup routines to find entries in the database, it is possible that the returned results will skip or duplicate entries. This is because the Location Broker does not prevent modification of the database between lookups, and such modification can change the locations of entries relative to a *lookup_handle* value.

It is also possible that the results of a single lookup routine will skip or duplicate entries. This can occur if the size of the results exceeds the size of an RPC packet (64K bytes).

Examples

The following statement, looks up GLB database entries for the object identified by `bank_id`:

```
lb_$lookup_object(&bank_id, &lookup_handle,
                          MAX_LOCS, &n_locs, bank_loc, &status);
```

Diagnostics

This section lists status codes for errors returned by this `lb_$` routine in `status.all`.

lb_$database_invalid
The format of the Location Broker database is out of date. The database may have been created by an old version of the Location Broker; in this case, delete the out-of-date database and reregister any entries that it contained. The LLB or GLB that was accessed may be running out-of-date software; in this case, update all Location Brokers to the current software version.

lb_$database_busy
The Location Broker database is currently in use in an incompatible manner.

lb_$not_registered
The Location Broker does not have any entries that match the criteria specified in the lookup or unregister routine. The requested object, type, interface, or combination thereof is not registered in the specified database. If you are using an `lb_$lookup_object_local` or `lb_$lookup_range` routine specifying an LLB, check that you have specified the correct LLB.

lb_$cant_access
The Location Broker cannot access the database. Among the possible reasons:

1. The database does not exist.

2. The database exists, but the Location Broker cannot access it.

lb_$server_unavailable
The Location Broker Client Agent cannot reach the requested GLB or LLB. A communications failure occurred or the broker was not running.

Files

`/usr/include/idl/c/glb.h`

See Also

intro(3ncs), lb_lookup_interface(3ncs), lb_lookup_object_local(3ncs), lb_lookup_range(3ncs), lb_lookup_type(3ncs)

Name

lb_lookup_object_local – look up information about an object in a Local Location Broker database

Syntax

#include <idl/c/lb.h>

void lb_$lookup_object_local(*object, location, location_length, lookup_handle max_num_results, num_results, results, status)*
*uuid_$t *object;*
*socket_$addr_t *location;*
unsigned long location_length;
*lb_$lookup_handle_t *lookup_handle;*
unsigned long max_num_results;
*unsigned long *num_results;*
lb_$entry_t results[ ];
*status_$t *status;*

Arguments

object The UUID of the object being looked up.

location The location of the LLB database to be searched. The socket address must specify the network address of a host. However, the port number in the socket address is ignored, and the lookup request is always sent to the LLB port.

location_length The length, in bytes, of the socket address specified by the location field.

lookup_handle A location in the database.

On input, the *lookup_handle* indicates the location in the database where the search begins. An input value of **lb_$default_lookup_handle** specifies that the search will start at the beginning of the database.

On return, the *lookup_handle* indicates the next unsearched part of the database (that is, the point at which the next search should begin). A return value of **lb_$default_lookup_handle** indicates that the search reached the end of the database; any other return value indicates that the search found at most *max_num_results* matching entries before it reached the end of the database.

max_num_results The maximum number of entries that can be returned by a single routine. This should be the number of elements in the *results* array.

num_results The number of entries that were returned in the *results* array.

results — An array that contains the matching GLB database entries, up to the number specified by the *max_num_results* parameter. If the array contains any entries for servers on the local network, those entries appear first.

status — The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `lb_$lookup_object_local` routine searches the specified LLB database and returns all entries whose *object* field matches the specified object.

The `lb_$lookup_object_local` routine cannot return more than *max_num_results* matching entries at a time. The *lookup_handle* parameter enables you to find all matching entries by doing sequential lookups.

If you use a sequence of lookup routines to find entries in the database, it is possible that the returned results will skip or duplicate entries. This is because the Location Broker does not prevent modification of the database between lookups, and such modification can change the locations of entries relative to a *lookup_handle* value.

It is also possible that the results of a single lookup routine will skip or duplicate entries. This can occur if the size of the results exceeds the size of an RPC packet (64K bytes).

Examples

The following statement looks up information about the object **locobj**. Since there is only one entry on any host, the routine will return at most one result:

```
lb_$lookup_object_local (&locobj_id, &location, location_length,
                         &lookup_handle, 1, &num_results,
                         &results, &status);
```

Diagnostics

This section lists status codes for errors returned by this `lb_$` routine in `status.all`.

lb_$database_invalid — The format of the Location Broker database is out of date. The database may have been created by an old version of the Location Broker; in this case, delete the out-of-date database and reregister any entries that it contained. The LLB that was accessed may be running out-of-date software; in this case, update all Location Brokers to the current software version.

lb_$database_busy — The Location Broker database is currently in use in an incompatible manner.

lb_$not_registered — The Location Broker does not have any entries that match the criteria specified in the lookup or unregister routine. The requested object, type, interface, or combination thereof is not registered in the specified database. If you are using an `lb_$lookup_object_local` or `lb_$lookup_range` routine specifying an LLB, check that you have specified the correct LLB.

lb_$cant_access The Location Broker cannot access the database. Among the possible reasons:

1. The database does not exist.

2. The database exists, but the Location Broker cannot access it.

lb_$server_unavailable
The Location Broker Client Agent cannot reach the requested LLB. A communications failure occurred or the broker was not running.

Files

`/usr/include/idl/c/glb.h`

See Also

intro(3ncs), lb_lookup_range(3ncs)

Name

lb_lookup_range – look up information in a Global Location Broker or Local Location Broker database

Syntax

#include <idl/c/lb.h>

void lb_$lookup_range(*object, obj_type, obj_interface, location,*
location_length, lookup_handle, max_num_results,
num_results, results, status)
uuid_$t **object*;
uuid_$t **obj_type*;
uuid_$t **obj_interface*;
socket_$addr_t **location*;
unsigned long *location_length*;
lb_$lookup_handle_t **lookup_handle*;
unsigned long *max_num_results*;
unsigned long **num_results*;
lb_$entry_t *results[ ]*;
status_$t **status)*;

Arguments

object The UUID of the object being looked up.

obj_type The UUID of the type being looked up.

obj_interface The UUID of the interface being looked up.

location The location of the database to be searched. If the value of *location_length* is 0, the GLB database is searched. Otherwise, the LLB database at the host specified by *location* is searched; in this case, the port number in the socket address is ignored, and the lookup request is sent to the LLB port.

location_length The length, in bytes, of the socket address specified by the *location* field. A value of 0 indicates that the GLB database is to be searched.

lookup_handle A location in the database.

On input, the *lookup_handle* indicates the location in the database where the search begins. An input value of **lb_$default_lookup_handle** specifies that the search will start at the beginning of the database.

On return, the *lookup_handle* indicates the next unsearched part of the database (that is, the point at which the next search should begin). A return value of **lb_$default_lookup_handle** indicates that the search reached the end of the database; any other return value indicates that the search found at most *max_num_results* matching entries before it reached the end of the database.

max_num_results — The maximum number of entries that can be returned by a single routine. This should be the number of elements in the *results* array.

num_results — The number of entries that were returned in the *results* array.

results — An array that contains the matching GLB database entries, up to the number specified by the *max_num_results* parameter. If the array contains any entries for servers on the local network, those entries appear first.

status — The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `lb_$lookup_range` routine returns database entries whose *object, obj_type,* and *obj_interface* fields match the specified values. A value of **uuid_$nil** in any of these input parameters acts as a wildcard and will match any value in the corresponding entry field. You can specify wildcards in any combination of these parameters.

The `lb_$lookup_range` routine cannot return more than *max_num_results* matching entries at a time. The *lookup_handle* parameter enables you to find all matching entries by doing sequential lookups.

If you use a sequence of lookup routines to find entries in the database, it is possible that the returned results will skip or duplicate entries. This is because the Location Broker does not prevent modification of the database between lookups, and such modification can change the locations of entries relative to a *lookup_handle* value.

It is also possible that the results of a single lookup routine will skip or duplicate entries. This can occur if the size of the results exceeds the size of an RPC packet (64K bytes).

Examples

The following statement looks up information in the GLB database about servers that export the **matrix** interface for any objects of type **array.** The variable **glb** is defined elsewhere as a null pointer.

```
lb_$lookup_range(&uuid_$nil, &array_id, &matrix_id, glb, 0,
                    &lookup_handle, max_results,
                    &num_results, results, &status);
```

Diagnostics

This section lists status codes for errors returned by this `lb_$` routine in `status.all`.

lb_$database_invalid — The format of the Location Broker database is out of date. The database may have been created by an old version of the Location Broker; in this case, delete the out-of-date database and reregister any entries that it contained. The LLB or GLB that was accessed may be running out-of-date software; in this case, update all Location Brokers to the current software version.

lb_$database_busy The Location Broker database is currently in use in an incompatible manner.

lb_$not_registered The Location Broker does not have any entries that match the criteria specified in the lookup or unregister routine. The requested object, type, interface, or combination thereof is not registered in the specified database. If you are using an `lb_$lookup_object_local` or `lb_$lookup_range` routine specifying an LLB, check that you have specified the correct LLB.

lb_$cant_access The Location Broker cannot access the database. Among the possible reasons:

1. The database does not exist.

2. The database exists, but the Location Broker cannot access it.

lb_$server_unavailable
The Location Broker Client Agent cannot reach the requested LLB. A communications failure occurred or the broker was not running.

Files

`/usr/include/idl/c/glb.h`

See Also

intro(3ncs), lb_lookup_interface(3ncs), lb_lookup_object(3ncs), lb_lookup_object_local(3ncs), lb_lookup_type(3ncs)

Name

lb_lookup_type – look up information about a type in the Global Location Broker database

Syntax

#include <idl/c/lb.h>

void lb_$lookup_type(*obj_type, lookup_handle, max_num_results,*
num_results, results, status)
uuid_$t **obj_type;*
lb_$lookup_handle_t **lookup_handle;*
unsigned long max_num_results;
*unsigned long *num_results;*
lb_$entry_t results[ ];
*status_$t *status;*

Arguments

obj_type	The UUID of the type being looked up.
lookup_handle	A location in the database. On input, the *lookup_handle* indicates the location in the database where the search begins. An input value of **lb_$default_lookup_handle** specifies that the search will start at the beginning of the database. On return, the *lookup_handle* indicates the next unsearched part of the database (that is, the point at which the next search should begin). A return value of **lb_$default_lookup_handle** indicates that the search reached the end of the database; any other return value indicates that the search found at most *max_num_results* matching entries before it reached the end of the database.
max_num_results	The maximum number of entries that can be returned by a single routine. This should be the number of elements in the *results* array.
num_results	The number of entries that were returned in the *results* array.
results	An array that contains the matching GLB database entries, up to the number specified by the *max_num_results* parameter. If the array contains any entries for servers on the local network, those entries appear first.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

lb_lookup_type(3ncs)

Description

The `lb_$lookup_type` routine returns GLB database entries whose *obj_type* fields match the specified type. It returns information about all objects of that type and about all interfaces to each of these objects.

The `lb_$lookup_type` routine cannot return more than *max_num_results* matching entries at a time. The *lookup_handle* parameter enables you to find all matching entries by doing sequential lookups.

If you use a sequence of lookup routines to find entries in the database, it is possible that the returned results will skip or duplicate entries. This is because the Location Broker does not prevent modification of the database between lookups, and such modification can change the locations of entries relative to a *lookup_handle* value.

It is also possible that the results of a single lookup routine will skip or duplicate entries. This can occur if the size of the results exceeds the size of an RPC packet (64K bytes).

Examples

The following statement looks up information in the GLB database about the type **array** :

```
lb_$lookup_type (&array_id, &lookup_handle, max_results,
                        &num_results, &results, &status);
```

Diagnostics

This section lists status codes for errors returned by this `lb_$` routine in `status.all`.

lb_$database_invalid The format of the Location Broker database is out of date. The database may have been created by an old version of the Location Broker; in this case, delete the out-of-date database and reregister any entries that it contained. The LLB or GLB that was accessed may be running out-of-date software; in this case, update all Location Brokers to the current software version.

lb_$database_busy The Location Broker database is currently in use in an incompatible manner.

lb_$not_registered The Location Broker does not have any entries that match the criteria specified in the lookup or unregister routine. The requested object, type, interface, or combination thereof is not registered in the specified database. If you are using an `lb_$lookup_object_local` or `lb_$lookup_range` routine specifying an LLB, check that you have specified the correct LLB.

lb_$cant_access The Location Broker cannot access the database. Among the possible reasons:

1. The database does not exist, and the Location Broker cannot create it.

2. The database exists, but the Location Broker cannot access it.

3. The GLB entry table is full.

lb_$server_unavailable
The Location Broker Client Agent cannot reach the requested GLB or LLB. A communications failure occurred or the broker was not running.

Files

`/usr/include/idl/c/glb.h`

See Also

intro(3ncs), lb_lookup_interface(3ncs), lb_lookup_object(3ncs), lb_lookup_range(3ncs)

lb_register (3ncs)

Name

lb_register – register an object and an interface with the Location Broker

Syntax

#include <idl/c/lb.h>

void lb_$register(*object, obj_type, obj_interface, flags, annotation,*
location, location_length, entry, status)
*uuid_$t *object;*
uuid_$t **obj_type;*
uuid_$t **obj_interface;*
lb_$server_flag_t *flags;*
unsigned char *annotation[64];*
socket_$addr_t **location;*
unsigned long *location_length;*
lb_$entry_t **entry;*
status_$t **status;*

Arguments

object	The UUID of the object being registered.
obj_type	The UUID of the type of the object being registered.
obj_interface	The UUID of the interface being registered.
flags	Must be either **lb_$server_flag_local** (specifying registration with only the LLB at the local host) or 0 (specifying registration with both the LLB and the GLB).
annotation	A character array used only for informational purposes. This field can contain a textual description of the object and the interface. For proper display by the `lb_admin` tool, the *annotation* should be terminated by a null character.
location	The socket address of the server that exports the interface to the object.
location_length	The length, in bytes, of the socket address specified by the *location* field.
entry	A copy of the entry that was entered in the Location Broker database.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `lb_$register` routine registers with the Location Broker an interface to an object and the location of a server that exports that interface. This routine replaces any existing entry in the Location Broker database that matches *object, obj_type, obj_interface,* and both the address family and host in *location;* if no such entry exists, the routine adds a new entry to the database.

If the *flags* parameter is `lb_$server_flag_local`, the entry is registered only in the LLB database at the host where the call is issued. Otherwise, the flag should be 0 to register with both the LLB and the GLB databases.

Examples

The following statement registers the `bank` interface to the object identified by `bank_id`:

```
lb_$register (&bank_id, &bank_$uuid, &bank_$if_spec.id, 0,
                      BankName, &saddr, slen, &entry, &status);
```

Diagnostics

This section lists status codes for errors returned by this `lb_$` routine in `status.all`.

lb_$database_invalid The format of the Location Broker database is out of date. The database may have been created by an old version of the Location Broker; in this case, delete the out-of-date database and reregister any entries that it contained. The LLB or GLB that was accessed may be running out-of-date software; in this case, update all Location Brokers to the current software version.

lb_$database_busy The Location Broker database is currently in use in an incompatible manner.

lb_$update_failed The Location Broker was unable to register the entry.

lb_$cant_access The Location Broker cannot access the database. Among the possible reasons:

1. The database does not exist, and the Location Broker cannot create it.

2. The database exists, but the Location Broker cannot access it.

3. The GLB entry table is full.

lb_$server_unavailable
The Location Broker Client Agent cannot reach the requested GLB or LLB. A communications failure occurred or the broker was not running.

Files

`/usr/include/idl/c/glb.h`

See Also

intro(3ncs), lb_unregister(3ncs)

lb_unregister (3ncs)

Name

lb_unregister – remove an entry from the Location Broker database

Syntax

#include <idl/c/lb.h>

void lb_$unregister(*entry, status*)
lb_$entry_t **entry;*
status_$t **status;*

Arguments

entry The entry being removed from the Location Broker database.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `lb_$unregister` routine removes from the Location Broker database the entry that matches *entry*. The value of *entry* should be identical to that returned by the `lb_$register` routine when the database entry was created. However, `lb_$unregister` does not compare all of the fields in *entry*, the **annotation** field, and the port number in the **saddr** field.

This routine removes the entry from the LLB database on the local host (the host that issues the routine). If the **flags** field of *entry* is equal to 0, it removes the entry from the GLB database. If the **flags** field is equal to **lb_$server_flag_local,** it deletes only the LLB entry.

Examples

The following statement unregisters the entry specified by `BankEntry,` which was obtained from a previous `lb_$register` routine:

```
lb_$unregister (&BankEntry, &status);
```

Diagnostics

This section lists status codes for errors returned by this `lb_$` routine in `status.all`.

lb_$database_invalid The format of the Location Broker database is out of date. The database may have been created by an old version of the Location Broker; in this case, delete the out-of-date database and reregister any entries that it contained. The LLB or GLB that was accessed may be running out-of-date software; in this case, update all Location Brokers to the current software version.

lb_$database_busy The Location Broker database is currently in use in an incompatible manner.

lb_$not_registered The Location Broker does not have any entries that match the criteria specified in the unregister routine. The requested object, type, interface, or combination thereof is not registered in the specified database.

lb_$update_failed
The Location Broker was unable to register or unregister the entry.

lb_$cant_access
The Location Broker cannot access the database. Among the possible reasons:

1. The database does not exist.

2. The database exists, but the Location Broker cannot access it.

lb_$server_unavailable
The Location Broker Client Agent cannot reach the requested GLB or LLB. A communications failure occurred or the broker was not running.

Files

`/usr/include/idl/c/glb.h`

See Also

intro(3ncs), lb_register(3ncs)

pfm_cleanup (3ncs)

Name

pfm_cleanup – establish a clean-up handler

Syntax

#include <idl/c/base.h>
#include <idl/c/pfm.h>

status_$t pfm_$cleanup(*cleanup_record*)
pfm_$cleanup_rec **cleanup_record;*

Arguments

cleanup_record — A record of the context when `pfm_$cleanup` is called. A program should treat this as an opaque data structure and not try to alter or copy its contents. It is needed by `pfm_$rls_cleanup` and `pfm_$reset_cleanup` to restore the context of the calling process at the clean-up handler entry point.

Description

The `pfm_$cleanup` routine establishes a clean-up handler that is executed when a fault occurs. A clean-up handler is a piece of code executed before a program exits when a signal is received by the process. The clean-up handler begins where `pfm_$cleanup` is called; the `pfm_$cleanup` routine registers an entry point with the system where program execution resumes when a fault occurs. When a fault occurs, execution resumes after the most recent call to `pfm_$cleanup`.

There can be more than one clean-up handler in a program. Multiple clean-up handlers are executed consecutively on a last-in/first-out basis, starting with the most recently established handler and ending with the first clean-up handler. The system provides a default clean-up handler established at program invocation. The default clean-up handler is always called last, just before a program exits, and releases any system resources still held, before returning control to the process that invoked the program.

Diagnostics

When called to establish a clean-up handler, `pfm_$cleanup` returns the status **pfm_$cleanup_set** to indicate the clean-up handler was successfully established. When the clean-up handler is entered in response to a fault signal, `pfm_$cleanup` effectively returns the value of the fault that triggered the handler.

This section lists status codes for errors returned by this `pfm_$` routine in `status.all`.

pfm_$bad_rls_order — Attempted to release a clean-up handler out of order.

pfm_$cleanup_not_found — There is no pending clean-up handler.

pfm_$cleanup_set — A clean-up handler was established successfully.

pfm_$cleanup_set_signalled
Attempted to use **pfm_$cleanup_set** as a signal.

pfm_$invalid_cleanup_rec
Passed an invalid clean-up record to a routine.

pfm_$no_space Cannot allocate storage for a clean-up handler.

NOTE

Clean-up handler code runs with asynchronous faults inhibited. When `pfm_$cleanup` returns something other than **pfm_$cleanup_set** indicating that a fault has occurred, there are four possible ways to leave the clean-up code:

- The program can call `pfm_$signal` to start the next clean-up handler with a different fault signal.
- The program can call `pgm_$exit` to start the next clean-up handler with the same fault signal.
- The program can continue with the code following the clean-up handler. It should generally call `pfm_$enable` to reenable asynchronous faults. Execution continues from the end of the clean-up handler code; it does not resume where the fault signal was received.
- The program can reestablish the handler by calling `pfm_$reset_cleanup` before proceeding.

Files

```
/usr/include/idl/c/base.h
/usr/include/idl/base.idl
/usr/include/idl/c/pfm.h
```

See Also

intro(3ncs), pfm_signal(3ncs)

pfm_enable(3ncs)

Name

pfm_enable – enable asynchronous faults

Syntax

#include <idl/c/base.h>
#include <idl/c/pfm.h>

void pfm_$enable()

Description

The `pfm_$enable` routine enables asynchronous faults after they have been inhibited by a routine to `pfm_$inhibit;` `pfm_$enable` causes the operating system to pass asynchronous faults on to the calling process.

While faults are inhibited, the operating system holds at most one asynchronous fault. Consequently, when `pfm_$enable` returns, there can be at most one fault waiting on the process. If more than one fault was received between routines to `pfm_$inhibit` and `pfm_$enable,` the process receives the first asynchronous fault received while faults were inhibited.

See Also

intro(3ncs), pfm_enable_faults(3ncs), pfm_inhibit(3ncs)

Name

pfm_enable_faults – enable asynchronous faults

Syntax

#include <idl/c/base.h>
#include <idl/c/pfm.h>

void pfm_$enable_faults()

Description

The `pfm_$enable_faults` routine enables asynchronous faults after they have been inhibited by a call to `pfm_$inhibit_faults`; `pfm_$enable_faults` causes the operating system to pass asynchronous faults on to the calling process.

While faults are inhibited, the operating system holds at most one asynchronous fault. Consequently, when `pfm_$enable_faults` returns, there can be at most one fault waiting on the process. If more than one fault was received between routines to `pfm_$inhibit_faults` and `pfm_$enable_faults`, the process receives the first asynchronous fault received while faults were inhibited.

Diagnostics

This section lists the status codes for errors returned by this `pfm_$` routine.

pfm_$bad_rls_order Attempted to release a clean-up handler out of order.

pfm_$cleanup_not_found
There is no pending clean-up handler.

pfm_$cleanup_set A clean-up handler was established successfully.

pfm_$cleanup_set_signalled
Attempted to use **pfm_$cleanup_set** as a signal.

pfm_$invalid_cleanup_rec
Passed an invalid clean-up record to a routine.

pfm_$no_space Cannot allocate storage for a clean-up handler.

Files

```
/usr/include/idl/c/base.h
/usr/include/idl/base.idl
/usr/include/idl/c/pfm.h
```

See Also

intro(3ncs), pfm_enable(3ncs), pfm_inhibit_faults(3ncs)

pfm_inhibit(3ncs)

Name

pfm_inhibit – inhibit asynchronous faults

Syntax

#include <idl/c/base.h>
#include <idl/c/pfm.h>

void pfm_$inhibit()

Description

The `pfm_$inhibit` routine prevents asynchronous faults from being passed to the calling process. While faults are inhibited, the operating system holds at most one asynchronous fault. Consequently, a call to `pfm_$inhibit` can result in the loss of some signals. It is good practice to inhibit faults only when absolutely necessary.

NOTE

This routine has no effect on the processing of synchronous faults such as floating-point and overflow exceptions, access violations, and so on.

Files

```
/usr/include/idl/c/base.h
/usr/include/idl/base.idl
/usr/include/idl/c/pfm.h
```

See Also

intro(3ncs), pfm_enable(3ncs), pfm_inhibit_fault(3ncs)

Name

pfm_inhibit_faults – inhibit asynchronous faults

Syntax

#include <idl/c/base.h>
#include <idl/c/pfm.h>

void pfm_$inhibit_faults()

Description

The `pfm_$inhibit_faults` routine prevents asynchronous faults from being passed to the calling process. While faults are inhibited, the operating system holds at most one asynchronous fault. Consequently, a call to `pfm_$inhibit_faults` can result in the loss of some signals. It is good practice to inhibit faults only when absolutely necessary.

NOTE

This call has no effect on the processing of synchronous faults such as floating-point and overflow exceptions, access violations, and so on.

Files

```
/usr/include/idl/c/base.h
/usr/include/idl/base.idl
/usr/include/idl/c/pfm.h
```

See Also

intro(3ncs), pfm_enable_faults(3ncs), pfm_inhibit(3ncs)

pfm_init(3ncs)

Name

pfm_init – initialize the PFM package

Syntax

#include <idl/c/base.h>
#include <idl/c/pfm.h>

void pfm_$init(*flags*)
unsigned long *flags;*

Arguments

flags

pfm_init_signal_handlers
Currently the only valid flag value. A flag's variable must be set to contain this value or the call will perform no initialization . A call to **pfm_init_signal_handlers** causes C signals to be intercepted and converted to PFM signals. On ULTRIX and VMS systems, the signals intercepted are SIGINIT, SIGILL, SIGFPE, SIGTERM, SIGHUP, SIGQUIT, SIGTRAP, SIGBUS, SIGSEGV, and SIGSYS.

Description

The call to `pfm_$init()` establishes a default set of signal handlers for the routine. The call to `pfm_$init()` should be made prior to the application's use of all other runtime RPC routines. This enables the RPC runtime system to catch and report all fault and/or interrupt signals that may occur during normal operation. Additionally, the user may provide a fault processing clean-up handler for application-specific exit handling.

Files

```
/usr/include/idl/c/base.h
/usr/include/idl/base.idl
/usr/include/idl/c/pfm.h
```

See Also

intro(3ncs), pfm_cleanup(3ncs)

Name

pfm_reset_cleanup – reset a clean-up handler

Syntax

#include <idl/c/base.h>
#include <idl/c/pfm.h>

void pfm_$reset_cleanup(*cleanup_record, status)*
pfm_$cleanup_rec **cleanup_record*;
status_$t **status*;

Arguments

cleanup_record	A record of the context at the clean-up handler entry point. It is supplied by `pfm_$cleanup`, when the clean-up handler if first established.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `pfm_$reset_cleanup` routine reestablishes the clean-up handler last entered so that any subsequent errors enter it first. This procedure should only be used within clean-up handler code.

Diagnostics

This section lists status codes for errors returned by this `pfm_$` routine in `status.all`.

pfm_$bad_rls_order Attempted to release a clean-up handler out of order.

pfm_$cleanup_not_found
There is no pending clean-up handler.

pfm_$cleanup_set A clean-up handler was established successfully.

pfm_$invalid_cleanup_rec
Passed an invalid clean-up record to a routine.

pfm_$no_space Cannot allocate storage for a clean-up handler.

Files

```
/usr/include/idl/c/base.h
/usr/include/idl/base.idl
/usr/include/c/pfm.h
```

See Also

intro(3ncs)

pfm_rls_cleanup (3ncs)

Name

pfm_rls_cleanup – release clean-up handlers

Syntax

#include <idl/c/base.h>
#include <idl/c/pfm.h>

void pfm_$rls_cleanup(*cleanup_record, status)*
pfm_$cleanup_rec **cleanup_record*;
status_$t **status*;

Arguments

cleanup_record	The clean-up record for the first clean-up handler to release.
status	The completion status. If *status* is **pfm_$bad_rls_order**, it means that the caller attempted to release a clean-up handler before releasing all handlers established after it. This status is only a warning; the intended clean-up handler is released, along with all clean-up handlers established after it. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `pfm_$rls_cleanup` routine releases the clean-up handler associated with *cleanup_record* and all clean-up handlers established after it.

Diagnostics

This section lists the status codes for errors returned by this `pfm_$` routine in `status.all`.

pfm_$bad_rls_order Attempted to release a clean-up handler out of order.

pfm_$cleanup_not_found
There is no pending clean-up handler.

pfm_$cleanup_set A clean-up handler was established successfully.

pfm_$cleanup_set_signalled
Attempted to use **pfm_$cleanup_set** as a signal.

pfm_$invalid_cleanup_rec
Passed an invalid clean-up record to a routine.

Files

```
/usr/include/idl/c/base.h
/usr/include/idl/base.idl
/usr/include/idl/c/pfm.h
```

See Also

intro(3ncs)

pfm_signal (3ncs)

Name

pfm_signal – signal the calling process

Syntax

#include <idl/c/base.h>
#include <idl/c/pfm.h>

void pfm_$signal(*fault_signal*)
status_$t **fault_signal*;

Arguments

fault_signal	A fault code.

Description

The `pfm_$signal` routine signals the fault specified by *fault_signal* to the calling process. It is usually called to leave clean-up handlers.

Diagnostics

This section lists status codes for errors returned by this `pfm_$` routine.

pfm_$bad_rls_order Attempted to release a clean-up handler out of order.

pfm_$cleanup_not_found
There is no pending clean-up handler.

pfm_$cleanup_set A clean-up handler was established successfully.

pfm_$cleanup_set_signalled
Attempted to use **pfm_$cleanup_set** as a signal.

pfm_$invalid_cleanup_rec
Passed an invalid clean-up record to a routine.

pfm_$no_space Cannot allocate storage for a clean-up handler.

NOTE

This routine does not return when successful.

Files

```
/usr/include/idl/c/base.h
/usr/include/idl/base.idl
/usr/include/idl/c/pfm.h
```

See Also

intro(3ncs)

Name

pgm_exit – exit a program

Syntax

#include <idl/c/base.h>
#include <idl/c/pfm.h>

void pgm_$exit()

Description

The `pgm_$exit` routine exits from the calling program and returns control to the process that invoked it. When `pgm_$exit` is called any files left open by the program are closed, any storage acquired is released, and asynchronous faults are reenabled if they were inhibited by the calling program.

The `pgm_$exit` routine always calls `pfm_$signal`() with a status of **status_$ok.**

Files

```
/usr/include/idl/c/base.h
/usr/include/idl/base.idl
/usr/include/idl/c/pfm.h
```

See Also

intro(3ncs)

rpc_alloc_handle (3ncs)

Name

rpc_alloc_handle – create an RPC handle (client only)

Syntax

#include <idl/c/rpc.h>

handle_t rpc_$alloc_handle(*object, family, status)*
uuid_$t **object*;
unsigned long *family*;
status_$t **status*;

Arguments

object The UUID of the object to be accessed. If there is no specific object, specify **uuid_$nil**.

family The address family to use in communications to access the object. Currently, only **socket_$ internet** is supported.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$alloc_handle` routine creates an unbound RPC handle that identifies a particular object but not a particular server or host.

If a remote procedure call is made using the unbound handle, it will effect a broadcast to all Local Location Brokers (LLBs) on the local network. If the call's interface and the object identified by the handle are both registered with any LLB, that LLB forwards the request to the registering server. The client RPC runtime library returns the first response that it receives and binds the handle to the first responding server.

Examples

The following statement allocates a handle that identifies the Acme company's payroll database object:

```
h = rpc_$alloc_handle (&acme_pay_id, socket_$internet, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$comm_failure The client was unable to get a response from the server.

rpc_$unk_if The requested interface is not known. It is not registered in the server, the version number of the registered interface is different from the version number specified in the request, or the UUID in the request does not match the UUID of the registered interface.

rpc_$cant_create_sock
The RPC runtime library was unable to create a socket.

rpc_$cant_bind_sock The RPC runtime library created a socket but was unable to bind it to a socket address.

rpc_$wrong_boot_time
The server boot time value maintained by the client does not correspond to the current server boot time. The server was probably rebooted while the client program was running.

rpc_$not_in_call An internal error.

rpc_$you_crashed This error can occur if a server has crashed and restarted. A client RPC runtime library sends the error to the server if the client makes a remote procedure call before the server crashes, then receives a response after the server restarts.

rpc_$proto_error An internal protocol error.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_free_handle(3ncs), rpc_set_binding(3ncs)

rpc_allow_remote_shutdown (3ncs)

Name

rpc_allow_remote_shutdown – allow or disallow remote shutdown of a server (server only)

Syntax

#include <idl/c/rpc.h>

void rpc_$allow_remote_shutdown(*allow, checkproc, status)*
unsigned long allow;
rpc_$shut_check_fn_t checkproc;
status_$t **status*;

Arguments

allow	A value indicating 'false' if zero, 'true' otherwise.
checkproc	A pointer to a Boolean function.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$allow_remote_shutdown` routine allows or disallows remote callers to shut down a server using `rrpc_$shutdown`.

By default, servers do not allow remote shutdown via `rrpc_$shutdown`. If a server calls `rpc_$allow_remote_shutdown` with *allow* true (not zero) and *checkproc* nil, then remote shutdown will be allowed. If *allow* is true and *checkproc* is not nil, then when a remote shutdown request arrives, the function denoted by *checkproc* is called and the shutdown is allowed if the function returns true. If *allow* is false (zero), remote shutdown is disallowed.

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$not_in_call	An internal error.
rpc_$you_crashed	This error can occur if a server has crashed and restarted. A client RPC runtime library sends the error to the server if the client makes a remote procedure call before the server crashes, then receives a response after the server restarts.
rpc_$proto_error	An internal protocol error.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_shutdown(3ncs), rrpc_shutdown(3ncs)

rpc_bind(3ncs)

Name

rpc_bind – allocate an RPC handle and set its binding to a server (client only)

Syntax

#include <idl/c/rpc.h>

handle_t rpc_$bind(*object, sockaddr, slength, status)*
uuid_$t **object*;
socket_$addr_t **sockaddr*;
unsigned long *slength*;
status_$t **status*;

Arguments

object	The UUID of the object to be accessed. If there is no specific object, specify **uuid_$nil**.
sockaddr	The socket address of the server.
slength	The length, in bytes, of *sockaddr*.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$bind` routine creates a fully bound RPC handle that identifies a particular object and server. This routine is equivalent to an `rpc_$alloc_handle` routine followed by an `rpc_$set_binding` routine.

Examples

The following statement binds the binop client to the specified object and socket address. The **loc** parameter is the result of a previous call to `rpc_$name_to_sockaddr` which converted the host name and port number to a socket address.

```
rh = rpc_$bind (&uuid_$nil, &loc, llen, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$cant_bind_sock	The RPC runtime library created a socket but was unable to bind it to a socket address.
rpc_$not_in_call	An internal error.
rpc_$proto_error	An internal protocol error.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_clear_binding(3ncs), rpc_clear_server_binding(3ncs), rpc_set_binding(3ncs)

rpc_clear_binding (3ncs)

Name

rpc_clear_binding – unset the binding of an RPC handle to a host and server (client only)

Syntax

#include <idl/c/rpc.h>

void rpc_$clear_binding(*handle, status*)
handle_t *handle*;
status_$t **status*;

Arguments

handle The RPC handle whose binding is being cleared.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$clear_binding` routine removes any association between an RPC handle and a particular server and host, but it does not remove the association between the handle and an object. This routine saves the RPC handle so that it can be reused to access the same object, either by broadcasting or after resetting the binding to another server.

A remote procedure call made using an unbound handle is broadcast to all Local Location Brokers (LLBs) on the local network. If the call's interface and the object identified by the handle are both registered with any LLB, that LLB forwards the request to the registering server. The client RPC runtime library returns the first response that it receives and binds the handle to the first server that responded.

The `rpc_$clear_binding` routine is the inverse of the `rpc_$set_binding` routine.

Examples

Clear the binding represented in *handle*:

```
rpc_$clear_binding (handle, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$not_in_call An internal error.

rpc_$proto_error An internal protocol error.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_bind(3ncs), rpc_clear_server_binding(3ncs), rpc_set_binding(3ncs)

rpc_clear_server_binding (3ncs)

Name

rpc_clear_server_binding – unset the binding of an RPC handle to a server (client only)

Syntax

#include <idl/c/rpc.h>

void rpc_$clear_server_binding(*handle, status*)
handle_t *handle*;
status_$t **status*;

Arguments

handle The RPC handle whose binding is being cleared.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$clear_server_binding` routine removes the association between an RPC handle and a particular server (that is, a particular port number), but does not remove the associations with an object and with a host (that is, a network address). This call replaces a fully bound handle with a bound-to-host handle. A bound-to-host handle identifies an object located on a particular host but does not identify a server exporting an interface to the object.

If a client uses a bound-to-host handle to make a remote procedure call, the call is sent to the Local Location Broker (LLB) forwarding port at the host identified by the handle. If the call's interface and the object identified by the handle are both registered with the host's LLB, the LLB forwards the request to the registering server. When the client RPC runtime library receives a response, it binds the handle to the server. Subsequent remote procedure calls that use this handle are then sent directly to the bound server's port.

The `rpc_$clear_server_binding` routine is useful for client error recovery when a server dies. The port that a server uses when it restarts is not necessarily the same port that it used previously; therefore, the binding that the client was using may not be correct. This routine enables the client to unbind from the dead server while retaining the binding to the host. When the client sends a request, the binding is automatically set to the server's new port.

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$not_in_call An internal error.

rpc_$proto_error An internal protocol error.

Files

```
/usr/include/idl/rpc.idl
/usr/include/idl/c/rpc.h
```

See Also

intro(3ncs), rpc_bind(3ncs), rpc_clear_binding(3ncs), rpc_set_binding(3ncs)

rpc_dup_handle(3ncs)

Name

rpc_dup_handle – make a copy of an RPC handle (client only)

Syntax

#include <idl/c/rpc.h>

handle_t rpc_$dup_handle(*handle, status*)
handle_t *handle*;
status_$t **status*;

Arguments

handle The RPC handle to be copied.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$dup_handle` routine returns a copy of an existing RPC handle. Both handles can then be used in the client program for concurrent multiple accesses to a binding. Because all duplicates of a handle reference the same data, an `rpc_$set_binding`, `rpc_$clear_binding`, or `rpc_$clear_server_binding` routine made on any one duplicate affects all duplicates. However, an RPC handle is not freed until `rpc_$free_handle` is called on all copies of the handle.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_alloc_handle(3ncs), rpc_free_handle(3ncs)

Name

rpc_free_handle – free an RPC handle (client only)

Syntax

#include <idl/c/rpc.h>

void rpc_$free_handle(*handle, status*)
handle_t *handle*;
status_$t **status*;

Arguments

handle	The RPC handle to be freed.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$free_handle` routine frees an RPC handle. This routine clears any association between the handle and a server or an object and releases the resources identified by the RPC handle. The client program cannot use a handle after it is freed.

Examples

The following statement frees a handle:

```
rpc_$free_handle (handle, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$not_in_call	An internal error.
rpc_$proto_error	An internal protocol error.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_alloc_handle(3ncs), rpc_dup_handle(3ncs)

rpc_inq_binding(3ncs)

Name

rpc_inq_binding – return the socket address represented by an RPC handle (client or server)

Syntax

#include <idl/c/rpc.h>

void rpc_$inq_binding(*handle, sockaddr, slength, status*)
handle_t *handle*;
socket_$addr_t **sockaddr*;
unsigned long **slength*;
status_$t **status*;

Arguments

handle	An RPC handle.
sockaddr	The socket address represented by *handle*.
slength	The length, in bytes, of *sockaddr*.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$inq_binding` routine enables a client to determine the socket address, and therefore the server, identified by an RPC handle. It is useful when a client uses an unbound handle in a remote procedure call and wishes to determine the particular server that responded to the call.

Examples

The Location Broker administrative tool, `lb_admin`, uses the following statement to determine the GLB that last responded to a lookup request:

```
rpc_$inq_binding(lb_$handle, &global_broker_addr,
                    &global_broker_addr_len, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$not_in_call An internal error.

rpc_$proto_error An internal protocol error.

rpc_$unbound_handle
The handle is not bound and does not represent a particular host address. Returned by `rpc_$inq_binding`.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_bind(3ncs), rpc_set_binding(3ncs)

rpc_inq_object (3ncs)

Name

rpc_inq_object – return the object UUID represented by an RPC handle (client or server)

Syntax

#include <idl/c/rpc.h>

void rpc_$inq_object(*handle, object, status*)
handle_t *handle*;
uuid_$t **object*;
status_$t **status*;

Arguments

handle	An RPC handle.
object	The UUID of the object identified by *handle*.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$inq_object` routine enables a client or server to determine the particular object that a handle represents.

If a server exports an interface through which clients can access several objects, it can use `rpc_$inq_object` to determine the object requested in a call. This routine requires an RPC handle as input, so the server can make the call only if the interface uses explicit handles (that is, if each operation in the interface has a handle parameter). If the interface uses an implicit handle, the handle identifier is not passed to the server.

Examples

A database server that manages multiple databases must determine the particular database to be accessed whenever it receives a remote procedure call. Each manager routine makes the following call; the routine then uses the returned UUID to identify the database to be accessed:

```
rpc_$inq_object (handle, &db_uuid, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$unk_if	The requested interface is not known. It is not registered in the server, the version number of the registered interface is different from the version number specified in the request, or the UUID in the request does not match the UUID of the registered interface.

rpc_$not_in_call An internal error.

rpc_$proto_error An internal protocol error.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs)

rpc_listen (3ncs)

Name

rpc_listen – listen for and handle remote procedure call (RPC) packets (server only)

Syntax

```
#include <idl/c/rpc.h>

void rpc_$listen(max_calls, status)
unsigned long max_calls;
status_$t *status;
```

Arguments

max_calls	This value indicates the maximum number of calls that the server is allowed to process concurrently. On ULTRIX systems, this value should be 1; any other value is ignored and defaulted to one.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$listen` routine dispatches incoming remote procedure call requests to manager procedures and returns the responses to the client. You must issue `rpc_$use_family` or `rpc_$use_family_wk` before you use `rpc_$listen`. This routine normally does not return. A return from this routine indicates either an irrecoverable error, or that an `rpc_shutdown` call has been issued. If `status.all` is equal to **status_$ok** , the assumption is that `rpc_$shutdown` has occurred.

Examples

Listen for incoming remote procedure call requests.

```
rpc_$listen (1, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$not_in_call	An internal error.
rpc_$you_crashed	This error can occur if a server has crashed and restarted. A client RPC runtime library sends the error to the server if the client makes a remote procedure call before the server crashes, then receives a response after the server restarts.
rpc_$proto_error	An internal protocol error.
rpc_$bad_pkt	The server or client has received an ill-formed packet.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
/usr/include/idl/c/rpc.h
```

See Also

intro(3ncs), rpc_shutdown(3ncs)

rpc_name_to_sockaddr (3ncs)

Name

rpc_name_to_sockaddr – convert a host name and port number to a socket address (client or server)

Syntax

#include <idl/c/rpc.h>

void rpc_$name_to_sockaddr(*name, nlength, port, family, sockaddr, slength, status*)
unsigned char *name*;
unsigned long *nlength*;
unsigned long *port*;
unsigned long *family*;
socket_$addr_t **sockaddr*;
unsigned long **slength*;
status_$t **status*;

Arguments

name — A string that contains a host name and, optionally, a port and an address family. The format is *family*:*host* [*port*], where *family*: and [*port*] are optional. If you specify a *family* as part of the *name* parameter, you must specify **socket_$unspec** in the *family* parameter. The *family* part of the name parameter is **ip**; *host* is the host name; *port* is an integer port number.

nlength — The number of characters in *name*.

port — The socket port number. This parameter should have the value **rpc_$unbound_port** if you are not specifying a well-known port; in this case, the returned socket address will specify the Local Location Broker (LLB) forwarding port at *host*. If you specify the port number in the *name* parameter, this parameter is ignored.

family — The address family to use for the socket address. This value corresponds to the communications protocol used to access the socket and determines how the *sockaddr* is expressed. If you specify the address family in the *name* parameter, this parameter must have the value **socket_$unspec**.

sockaddr — The socket address corresponding to *name*, *port*, and *family*.

slength — The length, in bytes, of *sockaddr*.

status — The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$name_to_sockaddr` routine provides the socket address for a socket, given the host name, the port number, and the address family.

You can specify the socket address information either as one text string in the *name* parameter or by passing each of the three elements as separate parameters(*name, port,* and *family*); in the latter case, the *name* parameter should contain only the hostname.

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$not_in_call An internal error.

rpc_$proto_error An internal protocol error.

NOTE

This routine has been superseded by the `socket_$from_name` routine.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_sockaddr_to_name(3ncs), socket_from_name(3ncs)

rpc_register (3ncs)

Name

rpc_register – register an interface (server only)

Syntax

#include <idl/c/rpc.h>

void rpc_$register(*ifspec, epv, status*)
rpc_$if_spec_t **ifspec*;
rpc_$epv_t *epv*;
status_$t **status*;

Arguments

ifspec	The interface being registered.
epv	The entry point vector (EPV) for the operations in the interface. The EPV is always defined in the server stub that is generated by the NIDL compiler from an interface definition.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$register` routine registers an interface with the RPC runtime library. After an interface is registered, the RPC runtime library will pass requests for that interface to the server.

You can call `rpc_$register` several times with the same interface (for example, from various subroutines of the same server), but each call must specify the same EPV. Each registration increments a reference count for the registered interface; an equal number of `rpc_$unregister` routines are then required to unregister the interface.

Examples

The following statement registers the bank interface with the bank server host's RPC runtime library:

```
rpc_$register (&bank_$if_spec, bank_$server_epv, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$op_rng_error	The requested operation does not correspond to a valid operation in the requested interface.

rpc_$too_many_ifs	The maximum number of interfaces is already registered with the RPC runtime library; the server must unregister some interface before it registers an additional interface.
rpc_$not_in_call	An internal error.
rpc_$you_crashed	This error can occur if a server has crashed and restarted. A client RPC runtime library sends the error to the server if the client makes a remote procedure call before the server crashes, then receives a response after the server restarts.
rpc_$proto_error	An internal protocol error.
rpc_$illegal_register	You are trying to register an interface that is already registered and you are using an EPV different from the one used when the interface was first registered. An interface can be multiply registered, but you must use the same EPV in each `rpc_$register` routine.
rpc_$bad_pkt	The server or client has received an ill-formed packet.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_register_mgr(3ncs), rpc_register_object(3ncs), rpc_unregister(3ncs)

rpc_register_mgr (3ncs)

Name

rpc_register_mgr – register a manager (server only)

Syntax

#include <idl/c/rpc.h>

void rpc_$register_mgr(*type, ifspec, sepv, mepv, status*)
uuid_$t **type*;
rpc_$if_spec_t **ifspec*;
rpc_$generic_epv_t *sepv*;
rpc_$mgr_epv_t *mepv*;
status_$t **status*;

Arguments

type	The UUID of the type being registered.
ifspec	The interface being registered.
sepv	The generic EPV, a vector of pointers to server stub procedures.
mepv	The manager EPV, a vector of pointers to manager procedures.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$register_mgr` routine registers the set of manager procedures that implement a specified interface for a specified type.

Servers can invoke this routine several times with the same interface (*ifspec*) and generic EPV (*sepv*) but with a different object type (*type*) and manager EPV (*mepv*) on each invocation. This technique allows a server to export several implementations of the same interface.

Servers that export several versions of the same interface (but not different implementations for different types) must also use `rpc_$register_mgr`, not `rpc_$register`. Such servers should supply **uuid_$nil** as the *type* to `rpc_$register_mgr`.

If a server uses `rpc_$register_mgr` to register a manager for a specific interface and a specific type that is not nil, the server must use `rpc_$register_object` to register an object.

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$op_rng_error	The requested operation does not correspond to a valid operation in the requested interface.

rpc_$unk_if	The requested interface is not known. It is not registered in the server, the version number of the registered interface is different from the version number specified in the request, or the UUID in the request does not match the UUID of the registered interface.
rpc_$too_many_ifs	The maximum number of interfaces is already registered with the RPC runtime library; the server must unregister some interface before it registers an additional interface.
rpc_$not_in_call	An internal error.
rpc_$you_crashed	This error can occur if a server has crashed and restarted. A client RPC runtime library sends the error to the server if the client makes a remote procedure call before the server crashes, then receives a response after the server restarts.
rpc_$proto_error	An internal protocol error.
rpc_$illegal_register	You are trying to register an interface that is already registered and you are using an EPV different from the one used when the interface was first registered. An interface can be multiply registered, but you must use the same EPV in each `rpc_$register` routine.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_register(3ncs), rpc_register_object(3ncs), rpc_unregister(3ncs)

rpc_register_object (3ncs)

Name

rpc_register_object – register an object (server only)

Syntax

```
#include &lt;idl/c/rpc.h&gt;

void rpc_$register_object(object, type, status)
uuid_$t *object;
uuid_$t *type;
status_$t *status;
```

Arguments

object	The UUID of the object being registered.
type	The UUID of the type of the object.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$register_object` routine declares that a server supports operations on a particular object and declares the type of that object.

A server must register objects with `rpc_$register_object` only if it registers generic interfaces with `rpc_$register_mgr`. When a server receives a call, the RPC runtime library searches for the object identified in the call (that is the object that the client specified in the handle) among the objects registered by the server. If the object is found, the type of the object determines which of the manager EPVs should be used to operate on the object.

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$op_rng_error	The requested operation does not correspond to a valid operation in the requested interface.
rpc_$unk_if	The requested interface is not known. It is not registered in the server, the version number of the registered interface is different from the version number specified in the request, or the UUID in the request does not match the UUID of the registered interface.
rpc_$too_many_ifs	The maximum number of interfaces is already registered with the RPC runtime library; the server must unregister some interface before it registers an additional interface.
rpc_$not_in_call	An internal error.
rpc_$proto_error	An internal protocol error.

rpc_$illegal_register You are trying to register an interface that is already registered and you are using an EPV different from the one used when the interface was first registered. An interface can be multiply registered, but you must use the same EPV in each `rpc_$register` routine.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_register(3ncs), rpc_register_mgr(3ncs), rpc_unregister(3ncs)

rpc_set_async_ack (3ncs)

Name

rpc_set_async_ack – set or clear asynchronous-acknowledgement mode (client only)

Syntax

#include <idl/c/rpc.h>

void rpc_$set_async_ack (*state*)
unsigned long *state*;

Arguments

state — If "true" (nonzero), asynchronous-acknowledgement mode is set. If "false" (zero), synchronous-acknowledgement mode is set.

Description

The `rpc_$set_async_ack` call sets or clears asynchronous-acknowledgement mode in a client.

Synchronous-acknowledgement mode is the default. Calling `rpc_$set_async_ack` with a nonzero value for *state* sets asynchronous-acknowledgement mode. Calling it with a zero value for *state* sets synchronous-acknowledgement mode.

After a client makes a remote procedure call and receives a reply from a server, the RPC runtime library at the client acknowledges its receipt of the reply. This "reply acknowledgement" can occur either synchronously (before the runtime library returns to the caller) or asynchronously (after the runtime library returns to the caller).

It is generally good to allow asynchronous reply acknowledgements. Asynchronous-acknowledgement mode can save the client runtime library from making explicit reply acknowledgements, because after a client receives a reply, it may shortly issue another call that can act as an implicit acknowledgement.

Asynchronous-acknowledgement mode requires that an "alarm" be set to go off sometime after the remote procedure call returns. Unfortunately, setting the alarm can cause two problems:

1. There may be only one alarm that can be set, and the application itself may be trying to use it.

2. If, at the time the alarm goes off, the application is blocked in a system call that is doing I/O to a "slow device" (such as a terminal), the system call will return an error (with the EINTR errno); the application may not be coded to expect this error. If neither of these problems exists, the application should set asynchronous-acknowledgement mode to get greater efficiency.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs)

rpc_set_binding (3ncs)

Name

rpc_set_binding – bind an RPC handle to a server (client only)

Syntax

#include <idl/c/rpc.h>

void rpc_$set_binding(*handle, sockaddr, slength, status*)
handle_t *handle*;
socket_$addr_t **sockaddr*;
unsigned long *slength*;
status_$t **status*;

Arguments

handle	An RPC handle.
sockaddr	The socket address of the server with which the handle is being associated.
slength	The length, in bytes, of *sockaddr*.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$set_binding` routine sets the binding of an RPC handle to the specified server. The handle then identifies a specific object at a specific server. Any subsequent remote procedure calls that a client makes using the handle are sent to this destination.

You can use this routine either to set the binding in an unbound handle or to replace the existing binding in a fully bound or bound-to-host handle.

Examples

The following statement sets the binding on the handle `h` to the first server in the `lbresults` array, which was returned by a previous Location Broker lookup routine, `lb_lookup_interface`:

```
rpc_$set_binding (h, &lbresults[0].saddr, lbresults[0].saddr_len,
                            &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$cant_bind_sock	The RPC runtime library created a socket but was unable to bind it to a socket address.
rpc_$not_in_call	An internal error.
rpc_$proto_error	An internal protocol error.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_alloc_handle(3ncs), rpc_clear_binding(3ncs), rpc_clear_server_binding(3ncs)

rpc_set_fault_mode (3ncs)

Name

rpc_set_fault_mode – set the fault-handling mode for a server (server only)

Syntax

#include <idl/c/rpc.h>

unsigned long rpc_$set_fault_mode(*state*)
unsigned long *state*;

Arguments

state If 'true' (not zero), the server exits when a fault occurs. If 'false' (zero), the server reflects faults back to the client.

Description

The `rpc_$set_fault_mode` function controls the handling of faults that occur in user server routines.

In the default mode, the server reflects faults back to the client and continues processing. Calling `rpc_$set_fault_mode` with value other than zero for *state* sets the fault-handling mode so that the server sends an **rpc_$comm_failure** fault back to the client and exits. Calling `rpc_$set_fault_mode` with *state* equal to zero resets the fault-handling mode to the default.

This function returns the previous state of the fault-handling mode.

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine.

rpc_$not_in_call An internal error.

rpc_$proto_error An internal protocol error.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs)

Name

rpc_set_short_timeout – set or clear short-timeout mode (client only)

Syntax

#include <idl/c/rpc.h>

unsigned long rpc_$set_short_timeout(*handle, state, status*)
handle_t *handle*;
unsigned long *state*;
status_$t **status*;

Arguments

handle An RPC handle.

on If 'true' (not zero), short-timeout mode is set on *handle*. If 'false' (zero), standard timeouts are set.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$set_short_timeout` routine sets or clears short-timeout mode on a handle. If a client uses a handle in short-timeout mode to make a remote procedure call, but the server does not respond, the call fails quickly. As soon as the server responds, standard timeouts take effect and apply for the remainder of the call.

Calling `rpc_$set_short_timeout` with a value other than zero for *state* sets short-timeout mode. Calling it with *state* equal to zero, sets standard timeouts. Standard timeouts are the default.

This routine returns the previous setting of the timeout mode in `status.all`.

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$not_in_call An internal error.

rpc_$proto_error An internal protocol error.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs)

rpc_shutdown (3ncs)

Name

rpc_shutdown – shut down a server (server only)

Syntax

#include <idl/c/rpc.h>

void rpc_$shutdown(*status*)
status_$t **status*;

Arguments

status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$shutdown` routine shuts down a server. When this routine is executed, the server stops processing incoming calls and `rpc_$listen` returns.

If `rpc_$shutdown` is called from within a remote procedure, that procedure completes, and the server shuts down after replying to the caller.

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$comm_failure	The call could not be completed due to a communication problem.
rpc_$not_in_call	An internal error.
rpc_$proto_error	An internal protocol error.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_allow_remote_shutdown(3ncs), rpc_listen(3ncs), rrpc_shutdown(3ncs)

Name

rpc_sockaddr_to_name – convert a socket address to a host name and port number (client or server)

Syntax

#include <idl/c/rpc.h>

void rpc_$sockaddr_to_name(*sockaddr, slength, name, nlength, port, status*)
socket_$addr_t **sockaddr*;
unsigned long *slength*;
unsigned char *name*;
unsigned long **nlength*;
unsigned long **port*;
status_$t **status*;

Arguments

sockaddr	A socket address.
slength	The length, in bytes, of *sockaddr*.
name	A string that contains the host name and the address family. The format is *family:host [port]* where *family* is **ip**.
nlength	On input, *nlength* is the length of the *name* buffer. On output, *nlength* is the number of characters returned in the *name* parameter.
port	The socket port number.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$sockaddr_to_name` routine provides the address family, the host name, and the port number identified by the specified socket address.

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$not_in_call	An internal error.
rpc_$proto_error	An internal protocol error.

NOTE

This routine has been superseded by the `socket_$to_name` routine.

rpc_sockaddr_to_name (3ncs)

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_name_to_sockaddr(3ncs), socket_to_name(3ncs)

Name

rpc_unregister – unregister an interface (server only)

Syntax

#include <idl/c/rpc.h>

void rpc_$unregister(*ifspec, status*)
rpc_$if_spec_t **ifspec*;
status_$t **status*;

Arguments

ifspec	An **rpc_$if_spec_t**. An interface specifier obtained from a previous RPC register call. The interface being unregistered.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$unregister` routine unregisters an interface that the server previously registered with the RPC runtime library. After an interface is unregistered, the RPC runtime library will not pass requests for that interface to the server.

If a server uses several `rpc_$register` or `rpc_$register_mgr` routines to register an interface more than once, then it must call `rpc_$unregister` an equal number of times to unregister the interface.

Examples

The following statement unregisters a matrix arithmetic interface:

```
rpc_$unregister (&matrix_$if_spec, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$op_rng_error	The requested operation does not correspond to a valid operation in the requested interface.
rpc_$unk_if	The requested interface is not known. It is not registered in the server, the version number of the registered interface is different from the version number specified in the request, or the UUID in the request does not match the UUID of the registered interface.
rpc_$not_in_call	An internal error.
rpc_$proto_error	An internal protocol error.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_register(3ncs), rpc_register_mgr(3ncs), rpc_register_object(3ncs)

Name

rpc_use_family – create a socket of a specified address family for a remote procedure call (RPC) server (server only)

Syntax

#include <idl/c/rpc.h>

void rpc_$use_family(*family, sockaddr, slength, status*)
unsigned long *family*;
socket_$addr_t **sockaddr*;
unsigned long **slength*;
status_$t **status*;

Arguments

family	The address family of the socket to be created. The value must be one of **socket_$internet** or **socket_$unspec**.
sockaddr	The socket address of the socket on which the server will listen.
slength	The length, in bytes, of *sockaddr*.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$use_family` routine creates a socket for a server without specifying its port number. The RPC runtime software assigns a port number. If a server must listen on a particular well-known port, use `rpc_$use_family_wk` to create the socket.

A server listens on one socket per address family, regardless of how many interfaces that it exports. Therefore, servers should make this call once per supported address family.

Examples

The following statement creates a server's socket:

```
rpc_$use_family (family, &saddr, &slen, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$cant_create_sock
The RPC runtime library was unable to create a socket.

rpc_$not_in_call An internal error.

rpc_$proto_error An internal protocol error.

rpc_$too_many_sockets
The server is trying to use more than the maximum number of sockets that is allowed; it has called `rpc_$use_family` or `rpc_$use_family_wk` too many times.

rpc_$addr_in_use
The address and port specified in an `rpc_$use_family_wk` routine are already in use. This is caused by multiple calls to `rpc_$use_family_wk` with the same well-known port.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_use_family_wk(3ncs)

Name

rpc_use_family_wk – create a socket with a well-known port for a remote procedure call (RPC) server (server only)

Syntax

#include <idl/c/rpc.h>

void rpc_$use_family_wk(*family, ifspec, sockaddr, slength, status*)
unsigned long *family*;
rpc_$if_spec_t **ifspec*;
socket_$addr_t **sockaddr*;
unsigned long **slength*;
status_$t **status*;

Arguments

family — The address family of the socket to be created. This value corresponds to the communications protocol used to access the socket and determines how the sockaddr is expressed. The value must be one of **socket_$unspec** or **socket_$internet.**

ifspec — The interface that will be registered by the server. Typically, this parameter is the interface *if_spec* generated by the NIDL compiler from the interface definition; the well-known port is specified as an interface attribute.

sockaddr — The socket address of the socket on which the server will listen.

slength — The length, in bytes, of *sockaddr*.

status — The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rpc_$use_family_wk` routine creates a socket that uses the port specified through the *if_spec* parameter. Use this routine to create a socket only if a server must listen on a particular well-known port. Otherwise, use `rpc_$use_family`.

A server listens on one socket per address family, regardless of how many interfaces that it exports. Therefore, servers that use well-known ports should make this call once per supported address family.

Examples

The following statement creates the well-known socket identified by `sockaddr` for an array processor server:

```
rpc_$use_family_wk (socket_$internet, &matrix$if_spec,
                        &sockaddr, &slen, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$cant_create_sock
The RPC runtime library was unable to create a socket.

rpc_$not_in_call
An internal error.

rpc_$proto_error
An internal protocol error.

rpc_$too_many_sockets
The server is trying to use more than the maximum number of sockets that is allowed; it has called `rpc_$use_family` or `rpc_$use_family_wk` too many times.

rpc_$bad_pkt
The server or client has received an ill-formed packet.

rpc_$addr_in_use
The address and port specified in an `rpc_$use_family_wk` routine are already in use. This is caused by multiple calls to `rpc_$use_family_wk` with the same well-known port.

Files

```
/usr/include/idl/c/rpc.h
/usr/include/idl/rpc.idl
```

See Also

intro(3ncs), rpc_use_family(3ncs)

Name

rrpc_inq_interfaces – obtain a list of the interfaces that a server exports

Syntax

#include <idl/c/rrpc.h>

void rrpc_$inq_interfaces(*handle, max_ifs, ifs, l_if, status*)
handle_t *handle*;
unsigned long *max_ifs*;
rrpc_$interface_vec_t *ifs[]*;
unsigned long **l_if*;
status_$t **status*;

Arguments

handle	An RPC handle.
max_ifs	The maximum number of elements in the array of interface specifiers.
ifs	An array of **rpc_$if_spec_t**.
l_if	The index of the last element in the returned array.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rrpc_$inq_interfaces` routine returns an array of RPC interface specifiers.

Files

```
/usr/include/idl/c/rrpc.h
/usr/include/idl/rrpc.idl
```

See Also

intro(3ncs)

rrpc_inq_stats (3ncs)

Name

rrpc_inq_stats – obtain statistics about a server

Syntax

#include <idl/c/rrpc.h>

void rrpc_$inq_stats(*handle, max_stats, stats, l_stat, status*)
handle_t *handle*;
unsigned long *max_stats*;
rrpc_$stat_vec_t *stats*;
unsigned long **l_stat*;
status_$t **status*;

Arguments

handle A remote procedure call (RPC) *handle* .

max_stats The maximum number of elements in the array of statistics.

stats An array of 32-bit integers representing statistics about the server. A set of **rrpc_$sv** constants defines indices for the elements in this array. The following list describes the statistic indexed by each **rrpc_$sv** constant:

rrpc_$sv_calls_in
The number of calls processed by the server.

rrpc_$sv_rcvd
The number of packets received by the server.

rrpc_$sv_sent
The number of packets sent by the server.

rrpc_$sv_calls_out
The number of calls made by the server.

rrpc_$sv_frag_resends
The number of fragments sent by the server that duplicated previous sends.

rrpc_$sv_dup_frags_rcvd
The number of duplicate fragments received by the server.

l_stat The index of the last element in the returned array.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rrpc_$inq_stats` routine returns an array of integer statistics about a server.

Files

/usr/indlude/idl/c/rrpc.h
/usr/include/idl/rrpc.idl

See Also

intro(3ncs)

rrpc_shutdown (3ncs)

Name

rrpc_shutdown – shut down a server

Syntax

#include <idl/c/rrpc.h>

void rrpc_$shutdown(*handle, status*)
handle_t *handle*;
status_$t **status*;

Arguments

handle A remote procedure call (RPC) handle.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `rrpc_$shutdown` routine shuts down a server, if the server allows it. A server can use the `rpc_$allow_remote_shutdown` routine to allow or disallow remote shutdown.

Diagnostics

This section lists status codes for errors returned by this `rrpc_$` routine in `status.all`.

rrpc_$shutdown_not_allowd
You send an `rrpc_shutdown` request to a server that has not issued an `rpc_allow_remote_shutdown` call.

Files

```
/usr/include/idl/c/rrpc.h
/usr/include/idl/rrpc.idl
```

See Also

intro(3ncs), rpc_allow_remote_shutdown(3ncs), rpc_shutdown(3ncs)

Name

socket_equal – compare two socket addresses

Syntax

```
#include <idl/c/socket.h>

boolean socket_$equal(sockaddr1, s1length, sockaddr2, s2length, flags,
                      status)
socket_$addr_t *sockaddr1;
unsigned long s1length;
socket_$addr_t *sockaddr2;
unsigned long s2length;
unsigned long flags;
status_$t *status;
```

Arguments

sockaddr1
: A socket address. The socket address is the structure returned by either `rpc_use_family` or `rpc_use_family_wk`.

s1length
: The length, in bytes, of *sockaddr1*.

sockaddr2
: A socket address. The socket address is the structure returned by either `rpc_use_family` or `rpc_use_family_wk`.

s2length
: The length, in bytes, of *sockaddr2*.

flags
: The logical OR of values selected from the following:

socket_$eq_hostid	Indicates that the host IDs are to be compared.
socket_$eq_netaddr	Indicates that the network addresses are to be compared.
socket_$eq_port	Indicates that the port numbers are to be compared.
socket_$eq_network	Indicates that the network IDs are to be compared.

status
: The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `socket_$equal` routine compares two socket addresses. The *flags* parameter determines which fields of the socket addresses are compared. The call returns 'true' (not zero) if all of the fields compared are equal, 'false' (zero) if not.

Examples

The following routine compares the network and host IDs in the socket addresses *sockaddr1* and *sockaddr2:*

```
if (socket_$equal (&sockaddr1, s1length, &sockaddr2, s2length,
        socket_$eq_network | socket_$eq_hostid, &status))
printf ("sockaddrs have equal network and host IDs\n");
```

Files

/usr/include/idl/c/socket.h
/usr/include/idl/socket.idl

See Also

intro(3ncs)

Name

socket_family_from_name – convert an address family name to an integer

Syntax

#include <idl/c/socket.h>

unsigned long socket_$family_from_name(*name, nlength, status*)
socket_$string_t *name*;
unsigned long *nlength*;
status_$t **status*;

Arguments

name	The textual name of an address family. Currently, only **ip** is supported.
nlength	The length, in bytes, of *name*.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `socket_$family_from_name` routine returns the integer representation of the address family specified in the text string *name.*

Examples

The server program for the banks example, `/usr/examples/banks/bankd.c` accepts a textual family name as its first argument. The program uses the following `socket_$family_from_name` routine to convert this name to the corresponding integer representation:

```
family = socket_$family_from_name
              (argv[1], (long)strlen(argv[1]), &status);
```

Files

`/usr/include/idl/socket.idl`
`/usr/include/idl/c/socket.h`

See Also

intro(3ncs), socket_family_to_name(3ncs), socket_from_name(3ncs), socket_to_name(3ncs)

socket_family_to_name (3ncs)

Name

socket_family_to_name – convert an integer address family to a textual name

Syntax

#include <idl/c/socket.h>

void socket_$family_to_name(*family, name, nlength, status*)
unsigned long *family*;
socket_$string_t *name*;
unsigned long **nlength*;
status_$t **status*;

Arguments

family	The integer representation of an address family.
name	The textual name of *family*. Currently, only **ip** is supported.
nlength	On input, the maximum length, in bytes, of the name to be returned. On output, the actual length of the returned name.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `socket_$family_to_name` routine converts the integer representation of an address family to a textual name for the family.

Files

```
/usr/include/idl/socket.idl
/usr/include/idl/c/socket.h
```

See Also

intro(3ncs)

Name

socket_from_name – convert a name and port number to a socket address

Syntax

```
#include <idl/c/socket.h>

void socket_$from_name(family, name, nlength, port, sockaddr, slength,
                       status)
unsigned long family;
socket_$string_t name;
unsigned long nlength;
unsigned long port;
socket_$addr_t *sockaddr;
unsigned long *slength;
status_$t *status;
```

Arguments

family The integer representation of an address family. Value can be **socket_$internet** or **socket_$unspec** If the *family* parameter is **socket_$unspec**, then the *name* parameter is scanned for a prefix of *family***:** (for example, **ip:**).

name A string in the format *family***:***host* [*port*] , where *family***:**, *host*, and [*port*] are all optional.

The *family* is an address family. The only valid *family* is **ip**. If you specify a *family* as part of the *name* parameter, you must specify **socket_$unspec** in the *family* parameter.

The *host* is a host name. A leading number sign (#) can be used to indicate that the host name is in the standard numeric form (for example, #192.9.8.7). If *host* is omitted, the local host name is used.

The *port* is a port number. If you specify a *port* as part of the *name* parameter, the *port* parameter is ignored.

nlength The length, in bytes, of *name*.

port A port number. If you specify a port number in the *name* parameter, this parameter is ignored.

sockaddr A socket address.

slength The length, in bytes, of *sockaddr*.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `socket_$from_name` routine converts a textual address family, host name, and port number to a socket address. The address family and the port number can be either specified as separate parameters or included in the *name* parameter.

Files

```
/usr/include/idl/socket.idl
/usr/include/idl/c/socket.h
```

See Also

intro(3ncs), socket_family_from_name(3ncs), socket_to_name(3ncs)

Name

socket_to_name – convert a socket address to a name and port number

Syntax

#include <idl/c/socket.h>

void socket_$to_name(*sockaddr, slength, name, nlength, port, status*)
socket_$addr_t **sockaddr*;
unsigned long *slength*;
socket_$string_t *name*;
unsigned long **nlength*;
unsigned long **port*;
status_$t **status*;

Arguments

sockaddr	A socket address. The socket address is the structure returned by either `rpc_$use_family` or `rpc_$use_family_wk`.
slength	The length, in bytes, of *sockaddr*.
name	A string in the format *family:host[port]*, where *family* is the address family and *host* is the host name; *host* may be in the standard numeric form (for example, #192.1.2.3) if a textual host name cannot be obtained. Currently, only **ip** is supported for *family*.
nlength	On input, the maximum length, in bytes, of the name to be returned. On output, the actual length of the name returned.
port	The port number.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `socket_$to_name` routine converts a socket address to a textual address family, host name, and port number.

Files

```
/usr/include/idl/socket.idl
/usr/include/idl/c/socket.h
```

See Also

intro(3ncs), socket_family_to_name(3ncs), socket_from_name(3ncs), socket_to_numeric_name(3ncs)

socket_to_numeric_name (3ncs)

Name

socket_to_numeric_name – convert a socket address to a numeric name and port number

Syntax

#include <idl/c/socket.h>

void socket_$to_numeric_name(*sockaddr, slength, name, nlength, port, status*

socket_$addr_t **sockaddr*;
unsigned long *slength*;
socket_$string_t *name*;
unsigned long **nlength*;
unsigned long **port*;
status_$t **status*;

Arguments

sockaddr	A socket address. The socket address is the structure returned by either `rpc_$use_family` or `rpc_$use_family_wk`.
slength	The length, in bytes, of *sockaddr*.
name	A string in the format *family:host[port]*, where *family* is the address family and *host* is the host name in the standard numeric form (for example, #192.7.8.9 for an IP address). Currently only **ip** is supported for *family*.
nlength	On input, the maximum length, in bytes, of the name to be returned. (error if less than size of "nnnnn.nnnn"). On output, the actual length of the name returned.
port	The port number.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `socket_$to_numeric_name` routine converts a socket address to a textual address family, a numeric host name, and a port number.

Files

```
/usr/include/idl/socket.idl
/usr/include/idl/c/socket.h
```

See Also

intro(3ncs), socket_family_to_name(3ncs), socket_from_name(3ncs), socket_to_name(3ncs)

socket_valid_families (3ncs)

Name

socket_valid_families – obtain a list of valid address families

Syntax

#include <idl/c/socket.h>

void socket_$valid_families(*max_families, families, status*)
unsigned long **max_families*;
socket_$addr_family_t *families*[ ];
status_$t **status*;

Arguments

max_families	The maximum number of families that can be returned.
families[ ]	An array of **socket_$addr_family_t**. Possible values for this type are enumerated in `/usr/include/idl/nbase.idl`. Currently, only **ip** is supported for *family*.
status	The completion status. This variable is set if the *families[ ]* array is not long enough to hold all the valid families. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `socket_$valid_families` routine returns a list of the address families that are valid on the calling host.

Examples

The following routine returns the valid address family:

```
socket_$valid_families (1, &families, $status);
```

Files

```
/usr/include/idl/socket.idl
/usr/include/idl/c/socket.h
```

See Also

intro(3ncs), socket_valid_family(3ncs)

Name

socket_valid_family – check whether an address family is valid

Syntax

#include <idl/c/socket.h>

boolean socket_$valid_family(*family, status*)
unsigned long *family*;
fBstatus_$t **status*;

Arguments

family	The integer representation of an address family.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `socket_$valid_family` routine returns 'true' if the specified address family is valid for the calling host, 'false' if not valid.

Examples

The following routine checks whether **socket_$internet** is a valid address family:

```
internetvalid = socket_$valid_family(socket_$internet, &status);
```

Files

```
/usr/include/idl/socket.idl
/usr/include/idl/c/socket.h
```

See Also

intro(3ncs), socket_valid_families(3ncs)

uuid_decode(3ncs)

Name

uuid_decode – convert a character-string representation of a UUID into a UUID structure

Syntax

#include <idl/c/uuid.h>

void uuid_$decode(*s, uuid, status*)
uuid_$string_t *s*;
uuid_$t **uuid*;
status_$t **status*;

Arguments

s	The character-string representation of a UUID.
uuid	The UUID that corresponds to *s*.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok** , then the routine that supplied it was successful.

Description

The `uuid_$decode` routine returns the UUID corresponding to a valid character-string representation of a UUID.

Examples

The following routine returns as **foo_uuid** the UUID corresponding to the character-string representation in **foo_uuid_rep:**

```
uuid_$decode (foo_uuid_rep, &foo_uuid, &status);
```

Files

```
/usr/include/idl/uuid.idl
/usr/include/idl/c/uuid.h
```

See Also

intro(3ncs), uuid_encode(3ncs)

Name

uuid_encode – convert a UUID into its character-string representation

Syntax

#include <idl/c/uuid.h>

void uuid_$encode(*uuid, s*)
uuid_$t **uuid*;
uuid_$string_t *s*;

Arguments

uuid	A UUID.
s	The character-string representation of *uuid*.

Description

The `uuid_$encode` routine returns the character-string representation of a UUID.

Examples

The following routine returns as **foo_uuid_rep** the character-string representation for the UUID **foo_uuid:**

```
uuid_$encode (&foo_uuid, foo_uuid_rep);
```

Files

```
/usr/include/idl/uuid.idl
/usr/include/idl/c/uuid.h
```

See Also

intro(3ncs), uuid_decode(3ncs)

uuid_equal (3ncs)

Name

uuid_equal – compare two UUIDs

Syntax

#include <idl/c/uuid.h>

boolean uuid_$equal(*u1, u2*)
uuid_$t **u1*;
uuid_$t **u2*;

Arguments

u1	A UUID.
u2	Another UUID.

Description

The `uuid_$encode` routine compares the UUIDs *u1* and *u2*. It returns 'true' if they are equal, 'false' if they are not.

Examples

The following code compares the UUIDs **bar_uuid** and **foo_uuid:**

```
if (uuid_$equal (&bar_uuid, &foo_uuid))
    printf ("bar and foo UUIDs are equal\n");
else
    printf ("bar and foo UUIDs are not equal\n");
```

Files

`/usr/include/idl/uuid.idl`
`/usr/include/idl/c/uuid.h`

See Also

intro(3ncs)

Name

uuid_gen – generate a new UUID

Syntax

#include <idl/c/uuid.h>

void uuid_$gen(*uuid*)
uuid_$t **uuid*;

Arguments

uuid A pointer to a UUID structure to be filled in.

Description

The `uuid_$gen` routine returns a new UUID. Typically used when creating a new remote application.

Examples

The following routine returns as **new_uuid** a new UUID:

```
uuid_$gen (&new_uuid);
```

Files

```
/usr/include/idl/uuid.idl
/usr/include/idl/c/uuid.h
```

See Also

intro(3ncs)

Standard I/O Routines (3s)

Insert tabbed divider here.
Then discard this sheet.

Name

stdio – standard buffered input/output package

Syntax

#include <stdio.h>

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

Description

The functions described in section 3s constitute a user-level buffering scheme. The in-line macros `getc` and `putc`(3s) handle characters quickly. The higher level routines `gets, fgets, scanf, fscanf, fread, puts, fputs, printf, fprintf, fwrite` all use `getc` and `putc;` they can be freely intermixed.

A file with associated buffering is called a *stream*, and is declared to be a pointer to a defined type FILE. The `fopen`(3s) subroutine creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. There are three normally open streams with constant pointers declared in the include file and associated with the standard open files:

stdin	standard input file
stdout	standard output file
stderr	standard error file

A constant 'pointer' NULL (0) designates no stream at all.

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

Any routine that uses the standard input/output package must include the header file <stdio.h> of pertinent macro definitions. The functions and constants mentioned in sections labeled 3S are declared in the include file and need no further declaration. The constants, and the following 'functions' are implemented as macros; redeclaration of these names is perilous: `getc, getchar, putc, putchar, feof, ferror, fileno.`

VAX Only

On VAX machines, the GFLOAT version of *libc* is used when you use the `cc`(1) command with the **–Mg** option, or you use the `ld`(1) command with the **–lcg** option. The GFLOAT version of *libc* must be used with modules compiled with `cc`(1) using the **–Mg** option.

Also note that neither the compiler nor the linker `ld`(1) can detect when mixed double floating point types are used, and the program may produce erroneous results if this occurs on VAX machines.

System V Compatibility

This library contains System V compatibility features that are available to general ULTRIX programs. For a discussion of how these features are documented, and how to specify that the System V environment is to be used in compiling and linking your programs, see `intro`(3).

Diagnostics

The value EOF is returned uniformly to indicate that a FILE pointer has not been initialized with `fopen,` input (output) has been attempted on an output (input) stream, or a FILE pointer designates corrupt or otherwise unintelligible FILE data.

For purposes of efficiency, this implementation of the standard library has been changed to line buffer output to a terminal by default and attempts to do this transparently by flushing the output whenever a `read`(2) from the standard input is necessary. This is almost always transparent, but may cause confusion or malfunctioning of programs which use standard I/O routines but use `read`(2) themselves to read from the standard input.

In cases where a large amount of computation is done after printing part of a line on an output terminal, it is necessary to `fflush`(3s) the standard output before going off and computing so that the output will appear.

Files

/lib/libc.a
/usr/lib/libcg.a (VAX only)

See Also

open(2), close(2), read(2), write(2), fread(3s), fseek(3s), ferror(3s), fclose(3s), fopen(3s)

Name

ctermid – generate file name for terminal

Syntax

#include <stdio.h>

char *ctermid(s)
char *s;

Description

The `ctermid` subroutine generates the pathname of the controlling terminal for the current process, and stores it in a string.

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

NOTE

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

See Also

ttyname(3)

cuserid (3s)

Name

cuserid – get character login name of the user

Syntax

#include <stdio.h>

char *cuserid (s)
char *s;

Description

The `cuserid` subroutine generates a character-string representation of the login name of the owner of the current 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.

Return Value

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

In POSIX mode, if *s* is not a NULL pointer, *s* is the return value.

Environment

When your program is compiled using the POSIX environment, `cuserid` returns the name associated with the effective userid of the calling process. When compiled in the BSD or System V environments, it returns the name associated with the login activity on the controlling terminal, if any. Otherwise, it returns the same as in the POSIX environment.

See Also

getlogin(3), getpwent(3)

Name

fclose, fflush – close or flush a stream

Syntax

#include <stdio.h>

fclose(*stream*)
FILE **stream*;

fflush(*stream*)
FILE **stream*;

Description

The `fclose` routine causes any buffers for the named *stream* to be emptied, and the file to be closed. Buffers allocated by the standard input/output system are freed. The `fclose` routine is performed automatically upon calling `exit`.

The `fflush` routine causes any buffered data for the named output *stream* to be written to that file. If *stream* is NULL, all open output streams are flushed. The stream remains open.

Diagnostics

These functions return EOF if buffered data cannot be transferred to an output stream.

Environment

If not called in POSIX mode, these functions return EOF if *stream* is not associated with an output file. In POSIX mode, if *stream* is associated with an input file, the file pointer is positioned following the last byte read from that *stream.*

See Also

close(2), fopen(3s), setbuf(3s)

Name

ferror, feof, clearerr, fileno – stream status inquiries

Syntax

#include <stdio.h>

feof(stream)
FILE *stream;

ferror(stream)
FILE *stream

clearerr(stream)
FILE *stream

fileno(stream)
FILE *stream;

Description

The `ferror` function returns nonzero when an error has occurred reading or writing the named *stream*, otherwise zero. Unless cleared by `clearerr`, the error indication lasts until the stream is closed.

The `feof` function returns nonzero when end of file is read on the named input *stream*, otherwise zero.

The `clearerr` function resets both the error and EOF indicators on the named *stream*.

The `fileno` function returns the integer file descriptor associated with the *stream*, see `open`(2).

These functions are implemented as macros; they cannot be redeclared.

See Also

open(2), fopen(3s)

Name

fgetpos, fsetpos – save and restore stream position

Syntax

#include <stdio.h>

int fgetpos (*stream, pos*)
FILE **stream*;
fpos_t **pos*;

int fsetpos (*stream, pos*)
FILE **stream*;
fpos_t **pos*;

Description

The `fgetpos` function stores the current position of *stream* in *pos*.

The `fsetpos` function restores *stream* to the position returned by an earlier `fgetpos` call.

Return Value

If successful, the return value is zero; on failure, a nonzero value is returned and errno is set to the appropriate value.

See Also

fseek(3s)

fopen(3s)

Name

fopen, freopen, fdopen – open a stream

Syntax

#include <stdio.h>

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

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

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

Description

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

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

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

- "r" Open for reading
- "w" Truncate or create for writing
- "a" Append; open for writing at end of file, or create for writing
- "A" Append with no overwrite; open for writing at end-of-file, or create for writing
- "r+" Open for reading and writing
- "w+" Truncate or create for reading and writing
- "a+" Append; open or create for reading and writing at end-of-file
- "A+" Append with no overwrite, open or create for update at end-of-file

The letter "b" can also follow r, w, or a. In some C implementations, the "b" is needed to indicate a binary file, however, it is not needed in ULTRIX. If "+" is used, the "b" may occur on either side, as in "rb+" or "w+b".

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

The `freopen` routine is typically used to attach the preopened *streams* associated with **stdin**, **stdout** and **stderr** to other files.

The `fdopen` routine associates a *stream* with a file descriptor. File descriptors are obtained from `open`, `dup`, `creat`, or `pipe`(2), which open files but do not return pointers to a FILE structure *stream*. Streams are necessary input for many of the

Section 3s library routines. The *type* of *stream* must agree with the mode of the open file.

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

When a file is opened for append with no overwrite (that is when type is "A" or "A+"), it is impossible to overwrite information already in the file. The `fseek` routine may be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output. If two separate processes open the same file for append, each process may write freely to the file without fear of destroying output being written by the other. The output from the two processes will be intermixed in the file in the order in which it is written.

Return Value

The `fopen` and `freopen` routines return a NULL pointer on failure.

Environment

SYSTEM_V

When your program is compiled using the System V environment, append with no overwrite is specified by using the "a" or "a+" type string, and the "A" and "A+" type strings are not allowed.

POSIX

In the POSIX environment, the "a" and "a+" strings, and the "A" and "A+" strings specify append with no overwrite.

See Also

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

fread (3s)

Name

fread, fwrite – buffered binary input/output

Syntax

#include <stdio.h>

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

size_t fwrite(*ptr, size, nitems, stream*)
void **ptr*;
size_t *size, nitems*;
FILE **stream*;

Description

The `fread` function reads into a block beginning at *ptr*, *nitems* of data of the size *size* (usually sizeof **ptr*) from the named input *stream*. It returns the number of items actually read.

If *stream* is **stdin** and the standard output is line buffered, then any partial output line will be flushed before any call to `read`(2) to satisfy the `fread`.

The `fwrite` function appends, at most, *nitems* of data of the size *size* (usually sizeof **ptr*) beginning at *ptr* to the named output *stream*. It returns the number of items actually written.

Return Value

The `fread` and `fwrite` functions return 0 upon end of file or error.

See Also

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

Name

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

Syntax

#include <stdio.h>

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

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

void rewind(*stream*)
FILE **stream*;

Description

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

The `fseek` function undoes any effects of `ungetc`(3s).

The `ftell` function returns the current value of the offset relative to the beginning of the file associated with the named *stream*. It is measured in bytes and is the only foolproof way to obtain an *offset* for `fseek`.

The `rewind` (stream) function is equivalent to `fseek` (stream , 0L, 0, SEEK_SET), except that no value is returned.

Return Value

The `fseek` function returns –1 for improper seeks, otherwise 0.

See Also

lseek(2), fopen(3s)

getc(3s)

Name

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

Syntax

#include <stdio.h>

int getc(stream)
FILE *stream;

int getchar()

int fgetc(stream)
FILE *stream;

int getw(stream)
FILE *stream;

Description

The `getc` function returns the next character from the named input *stream*.

The `getchar` function is identical to `getc` (stdin).

The `fgetc` function behaves like `getc`, but is a genuine function, not a macro. It may be used to save object text.

The `getw` function returns the next word (in a 32-bit integer on a VAX-11 or MIPS machine) from the named input *stream*. It returns the constant EOF upon end of file or error, but since that is a good integer value, *feof* and `ferror`(3s) should be used to check the success of `getw`. The `getw` assumes no special alignment in the file.

Restrictions

Because it is implemented as a macro, `getc` treats a stream argument with side effects incorrectly. In particular, 'getc(*f++);' doesn't work as expected.

Diagnostics

These functions return the integer constant EOF at end of file or upon read error.

A stop with message, 'Reading bad file', means an attempt has been made to read from a stream that has not been opened for reading by `fopen`.

See Also

fopen(3s), fread(3s), gets(3s), putc(3s), scanf(3s), ungetc(3s)

Name

gets, fgets – get a string from a stream

Syntax

#include <stdio.h>

char *gets(s)
char *s;

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

Description

The `gets` routine reads a string into *s* from the standard input stream **stdin**. The string is terminated by a newline character, which is replaced in *s* by a null character. The `gets` routine returns its argument.

The `fgets` routine reads *n*–1 characters, or up to a newline character, whichever comes first, from the *stream* into the string *s*. The last character read into *s* is followed by a null character. The `fgets` routine returns its first argument.

Restrictions

The `gets` routine deletes a newline, while `fgets` keeps it.

Diagnostics

The `gets` and `fgets` routines return the constant pointer NULL upon end of file or error.

See Also

ferror(3s), fread(3s), getc(3s), puts(3s), scanf(3s)

Name

printf, fprintf, sprintf – formatted output conversion

Syntax

#include <stdio.h>

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

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

BSD Environment

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

System V and POSIX Environments

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

Description

The `printf` function places output on the standard output stream, `stdout`. The `fprintf` subroutine places output on the named output *stream*. The `sprintf` subroutine places output in the string *s*, and appends the null terminator '\0' to the end of the string.

The first argument controls how each of these functions converts, formats, and prints the other arguments. The first argument is a character string that contains two types of objects, characters and conversion specifications. These functions copy characters that appear in the first argument to the output stream. Conversion specifications cause these functions to convert the next succesive argument and send the formatted argument to the output stream.

You introduce conversion specifications using the percent sign (**%**). Following the **%**, you can include:

- Zero or more flags, which modify the meaning of the conversion specification.
- An optional minus sign (–), which specifies left adjustment of the converted value in the indicated field.
- An optional digit string that specifies a field width. If the converted value has fewer characters than the field width, `printf` pads the value with blanks. By default, `printf` pads the value on the left. If the conversion string specifies the value is left-justified, `printf` pads the value on the right. If the field width begins with a zero, `printf` pads the values with zeros, instead of blanks.
- An optional period (.), which separates the field width from the next digit string.
- An optional digit string specifying a precision. The precision controls the

number of digits that appear after the radix character, exponential and floating-point conversions. Precision also controls the maximum number of characters that are placed in the converted value for a string.

- The character **h** or **l** specifying that a following **d**, **i**, **o**, **u**, **x**, or **X** corresponds to an integer or longword integer argument. You can use an uppercase **L** or a lowercase **l.**
- A character that indicates the type of conversion to be applied.

A field width or precision can be an asterisk (*), instead of a digit string. If you use an asterisk, you can include an argument that supplies the field width or precision.

The flag characters and their meanings are as follows:

– The result of the conversion is left-justified within the field.

+ The result of a signed conversion always begins with a sign (+ or –).

blank
If the first character of a signed conversion is not a sign, `printf` pads the value on the left with a blank. If the blank and plus sign (+) flags both appear, `printf` ignores the blank flag.

The result has been converted to a different format. The value is to be converted to an alternative form.

For **c, d, s,** and **u** conversions, this flag has no effect.

For **o**
conversions, this flag increases the precision to force the first digit of the result to be a zero.

For **x** or **X** conversions, `printf` pads a non-zero result on the left with **0x** or **0X**.

For **e, E, f, g,** and **G** conversions, the result always contains a radix character, even if no digits follow that character. (A radix character usually appears in the result of these conversions only if a digit follows it.)

For **g** and **G** conversions, `printf` does not remove trailing zeros from the result.

The conversion characters and their meanings are as follows:

dox Convert the integer argument to decimal, octal, or hexadecimal notation, respectively.

f Convert the floating point or double precision argument to decimal notation in the style *[–]ddd.ddd*, where the number of *d*s following the radix character is equal to the precision for the argument. If the precision is missing, `printf` prints six digits. If the precision is explicitly zero, the function prints no digits and no radix characters.

e Convert the floating point or double precision argument in the style *[–]d. ddde±dd*, where one digit appears before the radix character and the number of digits that appear after the radix character is equal to the precision. When you omit the precision, `printf` prints six digits.

g Convert the floating point or double precision argument to style **d**, style **f**, or style **e**. The style `prinf` uses depends on the format of the converted value.

The function removes trailing zeros before evaluating the format of the converted value.

If a radix character appears in the converted value that is followed by a digit, `printf` uses style **d**. If the converted value contains an exponent that is is less than –4 or greater than the precision, the function uses style .BR e . Otherwise, the `printf` function uses style **f**.

c Print the character argument.

s Print the character argument. The `printf` function prints the argument until it encounters a null characters or has printed the number of characters specified by the precision. If the precision is zero or has not been specified, `printf` prints the character argument until it encounters a null character.

u Convert the unsigned integer argument to a decimal value. The result must be in the range of 0 through 4294967295, where the upper bound is defined by MAXUNIT.

i Convert the integer argument to decimal. (This conversion character is the same as **d**.)

n Store the number of characters formatted in the integer argument.

p Print the pointer to the argument. (This conversion character is the same as %08X).

% Print a percent sign (%). The function converts no argument.

A non-existent or small field width never causes truncation of a value. Padding takes place only if the specified field width exceeds the length of the value.

In all cases, the radix character `printf` uses is defined by the last successful call to `setlocale` category `LC_NUMERIC`. If `setlocale` category `LC_NUMERIC` has not been called successfully or if the radix character is undefined, the radix character defaults to a period (.).

International Environment

LC_NUMERIC If this environment is set and valid, `printf` uses the international language database named in the definition to determine radix character rules.

LANG If this environment variable is set and valid `printf` uses the international language database named in the definition to determine collation and character classification rules. If `LC_NUMERIC` is defined, its definition supercedes the definition of LANG.

Restrictions

The `printf` function cannot format values that exceed 128 characters.

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 use the following function call:

```
printf("%s, %s %d, %02d:%02d",
                    weekday, month, day, hour, min);
```

To print π to 5 decimal places use the following call:

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

Return Values

In the BSD environment, `printf` and `fprintf` return zero for success and EOF for failure. The `sprintf` subroutine returns its first argument for success and EOF for failure.

In the System V and POSIX environments, `printf`, `fprintf`, and `sprintf` return the number of characters transmitted for success. The `sprintf` function ignores the null terminator (\0) when calculating the number of characters transmitted. If an output error occurs, these routines return a negative value.

See Also

ecvt(3), nl_printf(3int), nl_scanf(3int), setlocale(3), putc(3s), scanf(3s), environ(5int)
Guide to Developing International Software

putc(3s)

Name

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

Syntax

#include <stdio.h>

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

putchar(*c*)

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

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

Description

The `putc` routine appends the character *c* to the named output *stream*. It returns the character written.

The `putchar(c)` routine is defined as `putc` (c, stdout).

The `fputc` routine behaves like `putc`, but is a genuine function rather than a macro.

The `putw` routine appends word (that is, **int**) *w* to the output *stream*. It returns zero. The `putw` routine neither assumes nor causes special alignment in the file.

Restrictions

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

Diagnostics

The `putc`, `putchar`, and `fputc` functions return the constant EOF upon error. The `putw` function returns a non-zero value on error.

See Also

fclose(3s), fopen(3s), fread(3s), getc(3s), printf(3s), puts(3s)

Name

puts, fputs – put a string on a stream

Syntax

#include <stdio.h>

puts(s)
char *s;

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

Description

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

The `fputs` subroutine copies the null-terminated string *s* to the named output *stream*.

Neither routine copies the terminal null character.

Restrictions

The `puts` subroutine appends a new line, while `fputs` does not.

See Also

fopen(3s), gets(3s), putc(3s), printf(3s), ferror(3s) fread(3s)

Name

scanf, fscanf, sscanf – convert formatted input

Syntax

#include <stdio.h>

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

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

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

Description

Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string, *format,* and a set of *pointer* arguments that indicate where to store the converted input. The `scanf` function reads from the standard input stream *stdin*. The `fscanf` function reads from the named input *stream*. The `sscanf` function reads from the character string *s*.

In the *format* string you specify how to convert the input stream. You may use one or more conversion specifications in a single format string, depending on the number of *pointer* arguments you specify. Conversion specifications are introduced by a percent sign and specify the format of one input field. You may also use spaces, tabs, form feeds, new-line characters, alphabetic characters, and numbers in the format string. The following list describes conversion specifications and the other components of a *format* string:

- Conversion specifications have the following format:

  `%[*][w][l][h][code]`

  Each conversion specification must be introduced by a percent sign. The rest of the conversion specification is optional and has the following purpose:

* Specifies that an input field in the input string is not read by `scanf;` that is, the function skips the field.

w Specifies the maximum field width.

l Specifies that the variable where the input value is stored is a longword integer or a double-precision variable. The `scanf` function ignores the **l** if the input field is a character string or a pointer.

h Specifies that the variable where the input value is stored is a short integer or floating-point variable. The `scanf` function ignores the **h** if the input field is a character string or a pointer.

type Specifies the conversion code. Possible values for the conversion code are described in the paragraphs that follow.

- Alphabetic characters and numbers that appear inside the *format* string, but not in a conversion specification, specify that `scanf` ignore those characters in the input string.
- The white-space characters in a *format* string that appear outside of a conversion specification normally have no effect on how `scanf` formats data. The exception is when the white space character precedes the **c** conversion code in the *format* string. In this case, the white space causes `scanf` to ignore leading white space in the input field. Normally, `scanf` treats leading white space as part of the input character string for the **c** conversion code.

Each conversion specification in the *format* string directs the conversion of the next input field. The `scanf` function stores the result of each conversion in the *pointer* that corresponds to the conversion specification. Thus, the conversion specification controls how `scanf` converts the first unread input field, and `scanf` stores the result in the first *pointer*. The second conversion specification controls how `scanf` converts the next input field. The `scanf` function stores the result of the second conversion in the second *pointer*, and so on.

You do not include *pointers* for conversion specifications that contain the asterisk character. These specifications cause `scanf` to ignore an input field, so no *pointer* storage is needed.

An input field is defined as a string of non-space characters; it begins at the first unread character and extends to the first inappropriate character or EOF. An inappropriate character is one that is not valid for the value `scanf` is reading. For example, the letter ''z'' is invalid for an integer value. If the `scanf` function does not reach EOF and encounters no inappropriate characters, the field width is the number of characters specified by *w*. For all conversion codes except left-bracket ([) and **c**, `scanf` ignores leading white space in an input field.

The conversion code controls how `scanf` converts an input field. The data type of a *pointer* that corresponds to a conversion specification must match the conversion code. For example, the *pointer* that corresponds to a **c** conversion code must point to a character variable. The *pointer* that corresponds to a **d** conversion code must point to an integer, and so on. The following list describes the valid conversion codes:

% The input field is a percent sign. The `scanf` function does not move any value to *pointer*.

d D The input field is a decimal integer; the corresponding *pointer* must point to an integer. If you specify **h** , *pointer* can point to a short integer.

u U The input field is an unsigned decimal integer; *pointer* must point to an unsigned integer.

o 0 The input field is octal integer is expected; the corresponding *pointer* must point to an integer. If you specify **h** , *pointer* can be a short integer.

x X The input field is a hexadecimal integer; the corresponding *pointer* must point to an integer pointer. If you specify **h**, *pointer* can be a short integer.

e,f,g The input field is an optionally signed string of digits. The field may contain a radix character and an exponent field begins with a letter **E** or **e**, followed by an optional sign or space and an integer. The *pointer* must point to a floating-point variable. If you specify **l**, *pointer* must point to a double-precision variable.

s The input field is a character string. The *pointer* must point to an array of characters large enough to contain the string and a termination character (\0). The `scanf` function adds the termination character automatically. A white-space character terminates the input field, so the input field cannot contain spaces.

c The input field is a character or character string. The *pointer* must point to either a character variable or a character array.

The `scanf` function reads white space in the input field, including leading white space. To cause `scanf` to ignore white space, you can include a space in front of the conversion specification that includes the **c**.

[The input field is a character string. The *pointer* must point to an array of characters large enough to contain the string and a termination character (\0). The `scanf` function adds the termination character automatically.

Following the left bracket, you specify a list of characters and a right bracket ( ]). The `scanf` function reads the input field until it encounters a character other than those listed between the brackets. The `scanf` function ignores white-space characters.

You can change the meaning of the characters within the brackets by including a circumflex (^) character before the list of characters. The circumflex causes `scanf` to read the input field until it encounters one of the characters in the list.

You can represent a range of characters by specifying the first character, a hyphen (-), and the last character. For example, you can express [0123456789] using [0–9]. When you use a hyphen to represent a range of characters, the first character you specify must precede or be equal to the last character you specify in the current collating sequence. If the last character sorts before the first character, the hyphen stands for itself. The hyphen also stands for itself when it is the first or the last character that appears within the brackets.

To include the right square bracket as a character within the list, put the right bracket first in the list. If the right bracket is preceded by any character other than the circumflex, `scanf` interprets it as a closing bracket.

At least one input character must be valid for this conversion to be considered successful.

i The input field is an integer. If the field begins with a zero, `scanf` interprets it as an octal value. If the field begins with ''0X'' or ''0x, `scanf` interprets it as a hexadecimal value. The *pointer* must point to an integer. If you specify **h**, *pointer* can point to a short integer.

n The `scanf` function maintains a running total of the number of input fields it has read so far. This conversion code causes `scanf` to store that total in the integer that corresponds to *pointer*.

p The input field is a pointer. The *pointer* must point to an integer variable.

In all cases, `scanf` uses the radix character and collating sequence that is defined by the last successful call to `setlocale` category `LC_NUMERIC` or `LC_COLLATE`. If the radix or collating sequence is undefined, the `scanf` function uses the C locale definitions.

International Environment

LC_NUMERIC If this environment is set and valid, `scanf` uses the international language database named in the definition to determine radix character rules.

LANG If this environment variable is set and valid `scanf` uses the international language database named in the definition to determine collation and character classification rules. If `LC_NUMERIC` is defined, its definition supersedes the definition of LANG.

Restrictions

You cannot directly determine whether conversion codes that cause `scanf` to ignore data (for example, brackets and asterisks) succeeded.

The `scanf` function ignores any trailing white-space characters, including a newline character. If you want `scanf` to read a trailing white-space character, include the character in the conversion code for the data item that contains it.

Examples

The following shows an example of calling the `scanf` function:

```
int i, n; float x; char name[50];

n = scanf("%d%f%s", &i, &x, name);
```

Suppose the input to the `scanf` function appear as follows:

```
25 54.32E-1 thompson
```

In this case, `scanf` assigns the value 25 to the *i* variable and the value 5.432 to the *x* variable. The character variable *name* receives the value thompson\0. The function returns the value 3 to the *n* variable because it read and assigned three input fields.

The following example demonstrates using the **d** conversion code to cause `scanf` to ignore characters:

```
int i; float x; char name[5];

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

Suppose the following shows the input to the function:

```
56789 0123 56a72
```

In this case, the `scanf` function assigns the value 56 to the *i* variable and the value

789.0 to the *x* variable. The function ignores the 0123 input field, because the **%*d** conversion specification causes `scanf` to skip one input field. The function assigns 56 to *name*; it reads the first two characters in the last input field and stops at the third character. The letter 'a' is not in the set of characters from 0 to 9.

Return Values

The `scanf` function returns the number of successfully matched and assigned input fields. This number can be zero if the `scanf` function encounters invalid input characters, as specified by the conversion specification, before it can assign input characters.

If the input ends before the first conflict or conversion, `scanf` returns EOF. These functions return EOF on end of input and a short count for missing or invalid data items.

Environment

In POSIX mode, the **E**, **F**, and **X** formats are treated the same as the **e**, **f**, and **x** formats, respectively; otherwise, the upper-case formats expect double, double, and long arguments, respectively.

See Also

atof(3), nl_scanf(3int), getc(3s), printf(3s), environ(5int)
Guide to Developing International Software

Name

setbuf, setbuffer, setlinebuf, setvbuf – assign buffering to a stream

Syntax

#include <stdio.h>

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

setbuffer(*stream, buf, size*)
FILE **stream*;
char **buf*;
int *size*;

setlinebuf(*stream*)
FILE **stream*;

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

Description

The three types of buffering available are unbuffered, block buffered, and line buffered. When an output stream is unbuffered, information appears on the destination file or terminal as soon as written; when it is block buffered many characters are saved up and written as a block; when it is line buffered characters are saved up until a new line is encountered or input is read from stdin. The routine `fflush,` may be used to force the block out early. Normally all files are block buffered. For further information, see `fclose`(3s). A buffer is obtained from `malloc`(3) upon the first `getc` or `putc` on the file. If the standard stream **stdout** refers to a terminal it is line buffered. The standard stream **stderr** is always unbuffered.

The `setbuf` routine is used after a stream has been opened but before it is read or written. The character array *buf* is used instead of an automatically allocated buffer. If *buf* is the constant pointer NULL, input/output will be completely unbuffered. A manifest constant BUFSIZ tells how big an array is needed:

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

The `setbuffer` routine, an alternate form of `setbuf,` is used after a stream has been opened but before it is read or written. The character array *buf* whose size is determined by the *size* argument is used instead of an automatically allocated buffer. If *buf* is the constant pointer NULL, input/output will be completely unbuffered.

The `setlinebuf` routine is used to change **stdout** or **stderr** from block buffered or unbuffered to line buffered. Unlike `setbuf` and `setbuffer` it can be used at any time that the file descriptor is active.

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

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

If *buf* is not the NULL pointer, the array it points to will be used for buffering, instead of an automatically allocated buffer. The *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.

A file can be changed from unbuffered or line buffered to block buffered by using `freopen`. For further information, see `fopen`(3s). A file can be changed from block buffered or line buffered to unbuffered by using `freopen` followed by `setbuf` with a buffer argument of NULL.

Restrictions

The standard error stream should be line buffered by default.

The `setbuffer` and `setlinebuf` functions are not portable to non 4.2 BSD versions of UNIX.

See Also

malloc(3), fclose(3s), fopen(3s), fread(3s), getc(3s), printf(3s), putc(3s), puts(3s).

Name

tmpfile – create a temporary file

Syntax

#include <stdio.h>

FILE ***tmpfile ( )**

Description

The `tmpfile` subroutine creates a temporary file and returns a corresponding FILE pointer. The file will automatically be deleted when all references to the file have been closed. The file is opened for update.

See Also

creat(2), unlink(2), fopen(3s), mktemp(3), tmpnam(3s)

Name

tmpnam, tempnam – create a name for a temporary file

Syntax

#include <stdio.h>

char *tmpnam (s)
char *s;

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

Description

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

The `tmpnam` subroutine always generates a file name using the path-name defined as `P_tmpdir` in the <stdio.h> header file. If *s* is NULL, `tmpnam` leaves its result in an internal static area and returns a pointer to that area. The next call to `tmpnam` will destroy the contents of the area. If *s* is not NULL, it is assumed to be the address of an array of at least `L_tmpnam` bytes, where `L_tmpnam` is a constant defined in <stdio.h>; `tmpnam` places its result in that array and returns *s*.

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

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

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

Notes

The `tmpnam` and `tempnam` routines generate a different file name each time they are called.

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

Restrictions

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

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

See Also

creat(2), unlink(2), fopen(3s), malloc(3), mktemp(3), tmpfile(3s)

ungetc (3s)

Name

ungetc – push character back into input stream

Syntax

#include <stdio.h>

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

Description

The `ungetc` routine pushes the character *c* back on an input stream. That character will be returned by the next `getc` call on that stream. The `ungetc` routine returns *c*. One character of pushback is guaranteed in all cases.

The `fseek`(3s) routine erases all memory of pushed back characters.

Diagnostics

The `ungetc` routine returns EOF if it cannot push a character back.

Environment

In POSIX mode, the file's EOF indicator is cleared.

See Also

fseek(3s), getc(3s), setbuf(3s)

Name

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

Syntax

#include <stdio.h>
#include <varargs.h>

int vprintf (format, ap)
char *format;
va_list ap;

int vfprintf (stream, format, ap)
FILE *stream;
char *format;
va_list ap;

int vsprintf (s, format, ap)
char *s, *format;
va_list ap;

Description

The `vprintf`, `vfprintf`, and `vsprintf` routines are the same as `printf`, `fprintf`, and `sprintf`, respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by `varargs`(3).

Examples

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

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

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

See Also

varargs(3)

Special Library Routines (3x)

Insert tabbed divider here.
Then discard this sheet.

Name

intro – introduction to miscellaneous library functions

Description

These functions constitute minor libraries and other miscellaneous runtime facilities. Most are available only when programming in C.

The list below includes libraries which provide device-independent plotting functions, terminal-independent screen management routines for two-dimensional nonbitmap display terminals, functions for managing data bases with inverted indexes, and sundry routines used in executing commands on remote machines. The routines `getdiskbyname`, `rcmd`, `rresvport`, `ruserok`, and `rexec` reside in the standard C runtime library ''–lc''. All other functions are located in separate libraries indicated in each manual entry.

Files

```
/lib/libc.a
/usr/lib/libdbm.a
/usr/lib/libtermcap.a
/usr/lib/libcurses.a
/usr/lib/lib2648.a
/usr/lib/libplot.a
```

Name

creatediskbyname – get the disk description associated with a file name

Syntax

#include <disktab.h>

struct disktab *
creatediskbyname(*name*)
char **name*;

Description

The `creatediskbyname` subroutine takes the name of the character device special file representing a disk device (for example, `/dev/rra0a)` and returns a structure pointer describing its geometry information and the default disk partition tables. It obtains this information by polling the controlling disk device driver. The `creatediskbyname` subroutine returns information only for MSCP and SCSI disks.

The `<disktab.h>` file has the following form:

```
#define DISKTAB        "/etc/disktab"

struct  disktab {
      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[8];
};

struct  disktab *getdiskbyname();
struct  disktab *creatediskbyname();
```

Diagnostics

Successful completion of the `creatediskbyname` subroutine returns a pointer to a valid disktab structure. Failure of this subroutine returns a null pointer. The subroutine fails if it cannot obtain the necessary information from the device driver or disktab file.

A check is done to ensure that the disktab file exists and is readable. This check ensures that the subroutine is not being called because the disktab file was accidentally removed. If there is no disktab file, the subroutine fails.

The `creatediskbyname` subroutine also fails if it cannot determine disk geometry attributes by polling the driver. This can occur if the disk is not an MSCP or SCSI disk. In some cases where the disk consists of removable media and the media is not loaded, the driver will be unable to determine disk attributes.

Restrictions

The `creatediskbyname` subroutine returns information only for MSCP and SCSI disks.

See Also

getdiskbyname(3x), ra(4), rz(4), disktab(5)

Name

curses – screen functions with optimal cursor motion

Syntax

cc [flags] files **–lcurses –ltermcap** [libraries]

Description

These routines give the user a method of updating screens with reasonable optimization. They keep an image of the current screen, and the user sets up an image of a new one. Then the `refresh` subroutine tells the routines to make the current screen look like the new one. To initialize the routines, the routine `initscr` must be called before any of the other routines that deal with windows and screens are used. The routine `endwin` should be called before exiting.

Functions

addch(ch)	add a character to *stdscr*
addstr(str)	add a string to *stdscr*
box(win,vert,hor)	draw a box around a window
clear()	clear *stdscr*
clearok(scr,boolf)	set clear flag for *scr*
clrtobot()	clear to bottom on *stdscr*
clrtoeol()	clear to end of line on *stdscr*
crmode()	set cbreak mode
delch()	delete a character
deleteln()	delete a line
delwin(win)	delete *win*
echo()	set echo mode
endwin()	end window modes
erase()	erase *stdscr*
getch()	get a char through *stdscr*
getcap(name)	get terminal capability *name*
getstr(str)	get a string through *stdscr*
gettmode()	get tty modes
getyx(win,y,x)	get (y,x) co-ordinates
inch()	get char at current (y,x) co-ordinates
initscr()	initialize screens
insch(c)	insert a char
insertln()	insert a line
leaveok(win,boolf)	set leave flag for *win*
longname(termbuf,name)	get long name from *termbuf*
move(y,x)	move to (y,x) on *stdscr*
mvcur(lasty,lastx,newy,newx)	actually move cursor
newwin(lines,cols,begin_y,begin_x)	create a new window
nl()	set newline mapping
nocrmode()	unset cbreak mode
noecho()	unset echo mode
nonl()	unset newline mapping
noraw()	unset raw mode
overlay(win1,win2)	overlay win1 on win2

overwrite(win1,win2)	overwrite win1 on top of win2
printw(fmt,arg1,arg2,...)	printf on *stdscr*
raw()	set raw mode
refresh()	make current screen look like *stdscr*
resetty()	reset tty flags to stored value
savetty()	stored current tty flags
scanw(fmt,arg1,arg2,...)	scanf through *stdscr*
scroll(win)	scroll *win* one line
scrollok(win,boolf)	set scroll flag
setterm(name)	set term variables for name
standend()	end standout mode
standout()	start standout mode
subwin(win,lines,cols,begin_y,begin_x)	create a subwindow
touchwin(win)	''change'' all of *win*
unctrl(ch)	printable version of *ch*
waddch(win,ch)	add char to *win*
waddstr(win,str)	add string to *win*
wclear(win)	clear *win*
wclrtobot(win)	clear to bottom of *win*
wclrtoeol(win)	clear to end of line on *win*
wdelch(win,c)	delete char from *win*
wdeleteln(win)	delete line from *win*
werase(win)	erase *win*
wgetch(win)	get a char through *win*
wgetstr(win,str)	get a string through *win*
winch(win)	get char at current (y,x) in *win*
winsch(win,c)	insert char into *win*
winsertln(win)	insert line into *win*
wmove(win,y,x)	set current (y,x) co-ordinates on *win*
wprintw(win,fmt,arg1,arg2,...)	printf on *win*
wrefresh(win)	make screen look like *win*
wscanw(win,fmt,arg1,arg2,...)	scanf through *win*
wstandend(win)	end standout mode on *win*
wstandout(win)	start standout mode on *win*

See Also

ioctl(2), getenv(3), tty(4), termcap(3x), termcap(5)
Screen Updating and Cursor Movement Optimization: A Library Package, *ULTRIX Supplementary Documents* Vol. II:Programmer

Name

dbminit, fetch, store, delete, firstkey, nextkey – data base subroutines

Syntax

```
typedef struct {
        char *dptr;
        int dsize;
} datum;

dbminit(file)
char *file;

datum fetch(key)
datum key;

store(key, content)
datum key, content;

delete(key)
datum key;

datum firstkey()

datum nextkey(key)
datum key;
```

Description

These functions maintain key/content pairs in a data base. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two file system accesses. The functions are obtained with the loader option **–ldbm**.

Keys and *contents* are described by the *datum* typedef. A *datum* specifies a string of *dsize* bytes pointed to by *dptr*. Arbitrary binary data, as well as normal ASCII strings, are allowed. The data base is stored in two files. One file is a directory containing a bit map 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, the data stored under a key is accessed by `fetch` and data is placed under a key by `store`. A key (and its associated contents) is deleted by `delete`. A linear pass through all keys in a database may be made, in an (apparently) random order, by use of `firstkey` and `nextkey`. The `firstkey` will return the first key in the database. With any key `nextkey` will return the next key in the database. This code will traverse the data base:

```
for (key = firstkey(); key.dptr != NULL; key = nextkey(key))
```

Restrictions

The `.pag`file four times its actual content. Older UNIX systems may create real file blocks for these holes when touched. These files cannot be copied by normal means (`cp, cat, tp, tar, ar`) without filling in the holes.

The *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. The `store` will return an error in the event that a disk block fills with inseparable data.

The `delete` does not physically reclaim file space, although it does make it available for reuse.

Return Value

Routines that return a *datum* indicate errors with a null (0) *dptr*. All functions that return an *int* indicate errors with negative values. A zero return indicates a successful completion.

disassembler (3x)

Name

disassembler – disassemble a MIPS instruction and print the results

Syntax

```
int disassembler (iadr, regstyle, get_symname, get_regvalue, get_bytes, print_header)
unsigned        iadr;
int             regstyle;
char            *(*get_symname)();
int             (*get_regvalue)();
long            (*get_bytes)();
void            (*print_header)();
```

Description

The **disassembler** function disassembles and prints a MIPS machine instruction on *stdout.*

The argument is the instruction address to be disassembled. The *regstyle* parameter specifies how registers are named in the disassembly. The value is 0 if compiler names are used; otherwise, hardware names are used.

The next four arguments are function pointers, most of which give the caller some flexibility in the appearance of the disassembly. The only function that must be provided is *get_bytes*. All other functions are optional. The *get_bytes* function is called without arguments and returns the next byte or bytes to disassemble.

The *get_symname* is passed an address, which is the target of a *jal* instruction. If null is returned or if *get_symname* is null the *disassembler* prints the address; otherwise, the string name is printed as returned from *get_symname*. If *get_regvalue* is not null, it is passed a register number and returns the current contents of the specified register. The **disassembler** function prints this information along with the instruction disassembly. If *print_header* is not null, it is passed the instruction address, *iadr,* and the current instruction to be disassembled, which is the return value from *get_bytes*. The *print_header* function can use these parameters to print any desired information before the actual instruction disassembly is printed.

If *get_bytes* is null, the **disassembler** returns -1 and errno is set to EINVAL; otherwise, the number of bytes that were disassembled is returned. If the disassembled word is a jump or branch instruction, the instruction in the delay slot is also disassembled.

See Also

ldfcn(5)

Name

getdiskbyname – get disk description by its name

Syntax

#include <disktab.h>

struct disktab *
getdiskbyname(name)
char *name;

Description

The `getdiskbyname` subroutine takes a disk name (for example, RM03) and returns a structure describing its geometry information and the standard disk partition tables. All information obtained from the `disktab`(5) file. A separate subroutine called `createdis kbyname` dynamically generates disktab entries by obtaining disk geometry information from the controlling device driver.

<disktab.h> has the following form:

```
#define DISKTAB        "/etc/disktab"

struct  disktab {
      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[8];
};

struct  disktab *getdiskbyname();
struct  disktab *creatediskbyname();
```

See Also

creatediskbyname(3x), disktab(5)

Name

getfsent, getfsspec, getfsfile, getfstype, setfsent, endfsent – get file system descriptor file entry

Syntax

#include <fstab.h>
#include /usr/include/sys/fs_types.h

struct fstab *getfsent()

struct fstab *getfsspec(*spec*)
char **spec*;

struct fstab *getfsfile(*file*)
char **file*;

struct fstab *getfstype(*type*)
char **type*;

int setfsent()

int endfsent()

Description

All routines operate on the file `/etc/fstab`, which contains descriptions of the known file systems. The routine `setfsent` opens this file. The routine `getfsent` reads the next file system description within `/etc/fstab` opening the file if necessary. The `endfsent` routine closes the file.

The `getfsspec`, `getfsfile`, and `getfstype` routines sequentially scan the file `/etc/fstab` for specific file system descriptions. The `getfsspec` routine searches for a description with a matching special file name field. The routine `getfsfile` searches for a description with a matching file system path prefix field. The routine `getfstype` searches for a description with a matching file system type field.

The `getfsent`, `getfsspec`, `getfstype`, and `getfsfile` each return a pointer to a representation of the description they have matched or read. Representations are in the format of the following structure:

```
#define      FSTAB_RW     "rw"   /* read-write device      */
#define      FSTAB_RO     "ro"   /* read-only device       */
#define      FSTAB_RQ     "rq"   /* read-write with quotas */
#define      FSTAB_SW     "sw"   /* swap device            */
#define      FSTAB_XX     "xx"   /* ignore totally         */

struct fstab {
       char   *fs_spec;     /* block special device name    */
       char   *fs_file;     /* file system path prefix      */
       char   *fs_type;     /* rw,ro,sw or xx               */
       int    fs_freq;      /* dump frequency, in days      */
       int    fs_passno;    /* pass number on parallel dump */
       char   *fs_name;     /* name of the file system type */
       char   *fs_opts      /* arbitrary options field      */
};
```

Return Value

A NULL or 0 is returned, but *errno* is not set on detection of errors.

Restrictions

All descriptions are contained in static areas, which should be copied.

Files

/etc/fstab	File system information file.

See Also

fstab(5)

Name

initgroups – initialize group access list

Syntax

initgroups(name, basegid)
char *name;
int basegid;

Description

The `initgroups` subroutine reads through the group file and sets up, using the `setgroups`(2) call, the group access list for the user specified in *name*. The *basegid* is automatically included in the groups list. Typically this value is given as the group number from the password file.

Restrictions

The `initgroups` subroutine uses the routines based on `getgrent`(3). If the invoking program uses any of these routines, the group structure will be overwritten in the call to `initgroups`.

Return Value

The `initgroups` returns –1 if it was not invoked by the superuser.

Files

/etc/group

See Also

setgroups(2)

Name

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

Syntax

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

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

Description

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

The `ldahread` function returns success or failure. If **TYPE(***ldptr***)** does not represent an archive file or if it cannot read the archive header, `ldahread` fails.

See Also

intro(3x), ldclose(3x), ldopen(3x), ar(5), ldfcn(5)

ldclose (3x)

Name

ldclose, ldaclose – close a common object file

Syntax

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

int ldclose (ldptr)
LDFILE *ldptr;

int ldaclose (ldptr)
LDFILE *ldptr;

Description

The `ldopen` and `ldclose` functions provide uniform access to simple object files and object files that are members of archive files. An archive of common object files can be processed as if it is a series of simple common object files.

If **TYPE**(*ldptr*) does not represent an archive file, `ldclose` closes the file and frees the memory allocated to the **LDFILE** structure associated with *ldptr*. If **TYPE**(*ldptr*) is the magic number for an archive file and if archive has more files, `ldclose` reinitializes **OFFSET**(*ldptr*) to the file address of the next archive member and returns failure. The **LDFILE** structure is prepared for a later `ldopen`(3x). In all other cases, `ldclose` returns success.

The `ldaclose` function closes the file and frees the memory allocated to the **LDFILE** structure associated with *ldptr* regardless of the value of **TYPE**(*ldptr*). The `ldaclose` function always returns success. This function is often used with `ldaopen`.

See Also

fclose(3s), intro(3x) ldopen(3x), ldfcn(5), paths.h(4)

Name

ldfhread – read the file header of a common object file

Syntax

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

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

Description

The `ldfhread` function reads the file header of the common object file currently associated with *ldptr* . It reads the file header into the area of memory beginning at *filehead*.

The `ldfhread` function returns **success** If *ldfhread* cannot read the file header, it fails.

Usually, `ldfhread` can be avoided by using the macro **HEADER**(*ldptr*) defined in **<ldfcn.h>** see `ldfcn`(5)). Note that the information in HEADER is swapped, if necessary. The information in any field, *fieldname*, of the file header can be accessed using **HEADER**(*ldptr*).*fieldname*.

See Also

intro(3x), ldclose(3x), ldopen(3x), ldfcn(5).

ldgetaux(3x)

Name

ldgetaux – retrieve an auxiliary entry, given an index

Syntax

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

pAUXU ldgetaux (ldptr, iaux)
LDFILE ldptr;
long iaux;

Description

The `ldgetaux` function returns a pointer to an auxiliary table entry associated with *iaux*. The AUXU is contained in a static buffer. Because the buffer can be overwritten by later calls to `ldgetaux`, it must be copied by the caller if the aux is to be saved or changed.

Note that auxiliary entries are not swapped as this routine cannot detect what manifestation of the AUXU union is retrieved. If LDAUXSWAP(ldptr, ldf) is non-zero, a further call to *swap_aux* is required. Before calling the *swap_aux routine, the caller should copy*

If the auxiliary cannot be retrieved, `ldgetaux` returns null (defined in <stdio.h>) for an object file. This occurs in the following instances:

- The auxiliary table cannot be found
- The *iaux* offset into the auxiliary table is beyond the end of the table

Typically, `ldgetaux` is called immediately after a successful call to `ldtbread` to retrieve the data type information associated with the symbol table entry filled by `ldtbread`. The index field of the symbol, pSYMR, is the *iaux* when data type information is required. If the data type information for a symbol is not present, the index field is *indexNi* and `ldgetaux` should not be called.

See Also

intro(3x), ldclose(3x), ldopen(3x), ldtbseek(3x), ldtbread(3x), ldfcn(5).

Name

ldgetname – retrieve symbol name for object file symbol table entry

Syntax

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

char *ldgetname (ldptr, symbol)
LDFILE * ldptr ;
pSYMR * symbol ;

Description

The `ldgetname` function returns a pointer to the name associated with *symbol* as a string. The string is contained in a static buffer. Because the buffer can be overwritten by later calls to `ldgetname`, the caller must copy the buffer if the name is to be saved.

If the name cannot be retrieved, `ldgetname` returns null (defined in **<stdio.h>**) for an object file. This occurs in the following instances:

- The string table cannot be found
- The name's offset into the string table is beyond the end of the string table

Typically, `ldgetname` is called immediately after a successful call to `ldtbread`.The `ldgetname` retrieves the name associated with the symbol table entry filled by the function, `ldtbread`.

See Also

intro(3x), ldclose(3x), ldopen(3x), ldtbseek(3x), ldtbread(3x), ldfcn(5).

ldgetpd (3x)

Name

ldgetpd – retrieve procedure descriptor given a procedure descriptor index

Syntax

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

long ldgetpd (ldptr, ipd, ppd)
LDFILE ldptr;
long ipd;
pPDR ipd;

Description

The `ldgetpd` function returns success or failure depending on whether the procedure descriptor with index *ipd* can be accessed. If it can be accessed, the structure pointed to by *ppd* is filled with the contents of the corresponding procedure descriptor. The *isym, iline,* and *iopt* fields of the procedure descriptor are updated to be used in further LD routine calls. The *adr* field is updated from the symbol referenced by the *isym field.*

The PDR cannot be retrieved when the following occurs:

- The procedure descriptor table cannot be found.
- The ipd offset into the procedure descriptor table is beyond the end of the table.
- The file descriptor that the ipd offset falls into cannot be found.

Typically, `ldgetpd` is called while traversing the table that runs from 0 to SYMHEADER(ldptr).ipdMax – 1.

See Also

ldclose(3x), ldopen(3x), ldtbseek(3x), ldtbread(3x), ldfcn(5)

Name

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

Syntax

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

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

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

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

Description

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

The `dlinit` and `ldlitem` functions provide the same behavior as `ldlread.` After an initial call to `ldlread` or `ldlinit, ldlitem` can be used to retrieve a series of line number entries associated with a single function. The `ldlinit` function simply finds the line number entries for the function identified by *fcnindx*. The `ldlitem` function finds and reads the entry with the smallest line number equal to or greater than *linenum* into *linent*.

The functions `ldlread, ldlinit,` and `ldlitem` each return either success or failure. The `ldlread` function fails if one of the following occurs:

- If line number entries do not exist in the object file.
- If *fcnindx* does not index a function entry in the symbol table.
- If it does not find a line number equal to or greater than *linenum*.

The `ldlitem` fails if it does not find a line number equal to or greater than *linenum*.

See Also

ldclose(3x), ldopen(3x), ldtbindex(3x), ldfcn(5)

Name

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

Syntax

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

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

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

Description

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

The `ldnlseek` function seeks to the line number entries of the section specified by *sectname*.

The `ldlseek` and `ldnlseek` functions return success or failure.

NOTE

Line numbers are not associated with sections in the MIPS symbol table; therefore, the second argument is ignored, but maintained for historical purposes.

If they cannot seek to the specified line number entries, both routines fail.

See Also

ldclose(3x), ldopen(3x), ldshread(3x), ldfcn(5)

Name

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

Syntax

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

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

Description

The `ldohseek` function seeks to the optional file header of the common object file currently associated with *ldptr*.

`ldohseek` function returns success or failure. If the object file does not have an optional header or if it cannot seek to the optional header, `ldohseek` fails.

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

See Also

ldclose(3x), ldopen(3x), ldfhread(3x), ldfcn(5)

Name

ldopen, ldaopen – open a common object file for reading

Syntax

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

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

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

ld readst (ldptr, flags)
LDFILE ∗ldptr;
intflags;

Description

The `ldopen` and `ldclose` functions provide uniform access to simple object files and to object files that are members of archive files. An archive of common object files can be processed as if it is a series of simple common object files.

If *ldptr* has the value null, `ldopen` opens *filename*, allocates and initializes the **LDFILE** structure, and returns a pointer to the structure to the calling program.

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

The `ldopen` and `ldclose` functions work in concert. The `ldclose` function returns failure only when only when **TYPE(***ldptr***)** is the archive magic number and there is another file in the archive to be processed. Only then should `ldopen` be called with the current value of *ldptr*. In all other cases, but especially when a new *filename* is opened, `ldopen` should be called with a null *ldptr* argument.

The following is a prototype for the use of `ldopen` and

```
/* for each filename to be processed*/

ldptr = NULL;
do
        if ( (ldptr = ldopen(filename, ldptr)) != NULL )

        {
                /* check magic number */
                /* process the file */
        }
} while (ldclose(ldptr) == FAILURE );
```

If the value of *oldptr* is not **NULL**, `ldaopen` opens *filename* anew and allocates and initializes a new **LDFILE** structure, copying the fields from *oldptr*. The `ldaopen` function returns a pointer to the new **LDFILE** structure. This new pointer is independent of the old pointer, *oldptr*. The two pointers can be used concurrently to read separate parts of the object file. For example, one pointer can be used to step sequentially through the relocation information while the other is used to read indexed symbol table entries.

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

The `ldopen` function causes the symbol table header and file descriptor table to be read. Further access, using *ldptr,* causes other appropriate sections of the symbol table to be read (for example, if you call `ldtbread,` the symbols or externals are read). To force sections for each symbol table in memory, call `ldreadst` with *ST_P** constants or'ed together from *st_support.h.*

See Also

fopen(3s), ldclose(3x), ldfcn(5)

Name

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

Syntax

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

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

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

Description

The `ldrseek` function seeks to the relocation entries of the section specified by *sectindx* of the common object file currently associated with *ldptr*.

The `ldnrseek` function seeks to the relocation entries of the section specified by *sectname*.

The functions `ldrseek` and `ldnrseek` returns success or failure. If *sectindx* is greater than the number of sections in the object file, `ldrseek` fails; if there is no section name corresponding with *sectname,* `ldnrseek` fails. If the specified section does not have relocation entries or if it cannot seek to the specified relocation entries, either function fails.

NOTE

The first section has an index of *one*.

See Also

ldclose(3x), ldopen(3x), ldshread(3x), ldfcn(5)

ldshread (3x)

Name

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

Syntax

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

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

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

Description

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

The `ldnshread` functions reads the section header specified by *sectname* into the area of memory beginning at *secthead*.

The `ldshread` and `ldnshread` functions return success or failure. If *sectindx* is greater than the number of sections in the object file, *ldshread* fails. If there is no section name corresponding with *sectname,* `ldnshread` fails. If it cannot read the specified section header, either function fails.

NOTE

> The first section header has an index of *one*.

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

See Also

ldclose(3x), ldopen(3x), ldfcn(5).

Name

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

Syntax

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

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

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

Description

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

The `ldnsseek` seeks to the section specified by *sectname*.

The `ldsseek` and `ldnsseek` return success or failure. If *sectindx* is greater than the number of sections in the object file, `ldsseek` fails; if there is no section name corresponding with *sectname,* `ldnsseek` fails. If a no section data for the specified section does not exist or if it cannot seek to the specified section, either function fails.

NOTE

The first section has an index of *one*.

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

See Also

ldclose(3x), ldopen(3x), ldshread(3x), ldfcn(5)

ldtbindex(3x)

Name

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

Syntax

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

long ldtbindex (ldptr)
LDFILE *ldptr;

Description

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

The index returned by `ldtbindex` can be used in later calls to `ldtbread`(3x). `ldtbindex` returns the index of the symbol table entry that begins at the current position of the object file; therefore, if `ldtbindex` is called immediately after a particular symbol table entry has been read, it returns the the index of the next entry.

If there are no symbols in the object file or if the object file is not positioned at the beginning of a symbol table entry, `ldtbindex` fails and returns BADINDEX (–1).

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

See Also

ldclose(3x), ldopen(3x), ldtbread(3x), ldtbseek(3x), ldfcn(5)

Name

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

Syntax

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

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

Description

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

`ldtbread` returns success or failure. If *symindex* is greater than the number of symbols in the object file or if it cannot read the specified symbol table entry, `ldtbread` fails.

The local and external symbols are concatenated into a linear list. Symbols are accessible from symnum zero to *SYMHEADER(ldptr).isymMax+SYMHEADER(ldptr).iextMax*. The index and iss fields of the SYMR are made absolute (rather than file relative) so that routines `ldgetname`(3x), `ldgetaux`(3x), and `ldtbread` proceed normally given those indices. Only the sym part of externals is returned.

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

See Also

ldclose(3x), ldgetname(3x), ldopen(3x), ldtbseek(3x), ldgetname(3x), ldfcn(5)

ldtbseek(3x)

Name

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

Syntax

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

int ldtbseek (ldptr)
LDFILE *ldptr;

Description

The `ldtbseek` function seeks to the symbol table of the object file currently associated with *ldptr*.

The `ldtbseek` function returns success or failure. If the symbol table has been stripped from the object file or if it cannot seek to the symbol table, *ldtbseek* fails.

See Also

ldclose(3x), ldopen(3x), ldtbread(3x), ldfcn(5)

Name

malloc, free, realloc, calloc, mallopt, mallinfo – fast main memory allocator

Syntax

#include <malloc.h>
char *malloc (size)
unsigned size;

void free (ptr)
char *ptr;

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

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

int mallopt (cmd, value)
int cmd, value;

struct mallinfo mallinfo (max)
int max;

Description

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

The `malloc` subroutine returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to `free` is a pointer to a block previously allocated by `malloc`. After `free` is performed, this space is made available for further allocation, and its contents have been destroyed. See `mallopt` below for a way to change this behavior.

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

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

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

The `mallopt` subroutine provides for control over the allocation algorithm. The available values for *cmd* are:

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

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

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

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

These values are defined in the malloc.h header file.

The `mallopt` subroutine may be called repeatedly, but may not be called after the first small block is allocated.

The `mallinfo` subroutine provides information describing space usage. It returns the following 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.

Restrictions

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

Return Value

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

See Also

brk(2), malloc(3)

Name

nlist – get entries from name list

Syntax

#include <nlist.h>

nlist(filename, nl)
char *filename;
struct nlist nl[];

cc ... -lmld

Description

The `nlist` subroutine examines the name list in the given executable output file and selectively extracts a list of values. The name list consists of an array of structures containing names, types and values. The list is terminated with a null name. Each name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the next two fields. If the name is not found, both entries are set to 0. For the structure declaration, see */usr/include/nlist.h*.

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

Diagnostics

If the file cannot be found or if it is not a valid namelist –1 is returned; otherwise, the number of unfound namelist entries is returned.

The type entry is set to 0 if the symbol is not found.

See Also

a.out(5)

Name

openpl, erase, label, line, circle, arc, move, cont, point, linemod, space, closepl, box, color, dot – graphics interface

Syntax

openpl()

erase()

label(*s***)**
char *s***[];**

line(*x1*, *y1*, *x2*, *y2***)**

circle(*x*, *y*, *r***)**

arc(*x*, *y*, *x0*, *y0*, *x1*, *y1***)**

move(*x*, *y***)**

cont(*x*, *y***)**

point(*x*, *y***)**

linemod(*s***)**
char *s***[];**

space(*x0*, *y0*, *x1*, *y1***)**

closepl()

box(*x0*, *x1*, *y0*, *y1***)**

color(*c***)**

dot()

Description

These subroutines generate graphic output in a device-independent manner. See `plot`(5) for a description of their effect. The `openpl` subroutine precedes the other subroutines as it opens the device for writing. The `closepl` subroutine flushes the output. The `box`, `color`, and `dot` routines are used by the lvp16 and hp7475a plotters only.

String arguments to `label` and `linemod` are null-terminated and do not contain newlines.

Many of these functions have additional options for different output devices. They are accessed by the `ld`(1) options as follows:

–lplot	device-independent graphics stream on standard output for `plot`(1g) filters
–lplotaed	AED 512 color graphics terminal
–lplotbg	BBN bitgraph graphics terminal
–lplotdumb	dumb terminals without cursor addressing or line printers
–lplotgigi	gigi graphics terminal

–lplotgrn	grn files
–lplot2648	HP 2648 graphics terminal
–lplot7221	HP 7221 graphics terminal
–lplotimagen	Imagen laser printer (default 240 DPI resolution)
–l300	GSI 300 terminal
–l300s	GSI 300S terminal
–l450	DASI 450 terminal
–l4013	Tektronix 4013 terminal
–l4014	Tektronix 4014 terminal
–llvp16	DEC LVP16 and HP7475A plotters

See Also

graph(1g), plot(1g), plot(5)

Name

ranhashinit, ranhash, ranlookup – access routine for the symbol table definition file in archives

Syntax

#include <ar.h>

int ranhashinit(pran, pstr, size)
struct ranlib *pran;
char *pstr;
int size;

ranhash(name)
char *name;

struct ranlib *ranhash(name)
char *name;

Description

The function `ranhashinit` initializes static information for future use by `ranhash` and `ranlookup`. The argument *pran* points to an array of ranlib structures. The argument *pstr* points to the corresponding ranlib string table (these are only used by `ranlookup`). The argument size *size* is the size of the hash table and should be a power of 2. If the size is not a power of 2, a 1 is returned; otherwise, a 0 is returned.

The function `ranhash` returns a hash number given a name. It uses a multiplicative hashing algorithm and the *size* argument to `ranhashinit.`

The `ranlookup` function looks up *name* in the ranlib table specified by *ranhashinit.* It uses the `ranhash` routine as a starting point. Then, it does a rehash from there. This routine returns a pointer to a valid ranlib entry on a match. If no matches are found (the "emptiness" can be inferred if the ran_off field is zero), the empty ranlib structure hash table should be sparse. This routine does not expect to run out of places to look in the table. For example, if you collide on all entries in the table, an error is printed tostderr and a zero is returned.

See Also

ar(1), ar(5)

Name

rcmd, rresvport, ruserok – routines for returning a stream to a remote command

Syntax

rem = rcmd(*ahost, inport, locuser, remuser, cmd, fd2p***);**
char ***ahost***;**
u_short *inport***;**
char **locuser*, **remuser*, **cmd***;**
int **fd2p***;**

s = rresvport(*port***);**
int **port***;**

ruserok(*rhost, superuser, ruser, luser***)**
char **rhost***;**
int *superuser***;**
char **ruser*, **luser***;**

Description

The `rcmd` subroutine is used by the superuser to execute a command on a remote machine using an authentication scheme based on reserved port numbers. The `rresvport` subroutine is a routine that returns a descriptor to a socket with an address in the privileged port space. The `ruserok` subroutine is a routine used by servers to authenticate clients requesting service with `rcmd`. All three functions are present in the same file and are used by the `rshd`(8c) server (among others).

The `rcmd` subroutine looks up the host **ahost* using `gethostbyname`(3n), returning –1 if the host does not exist. For further information, see `gethostent`(3n). Otherwise **ahost* is set to the standard name of the host and a connection is established to a server residing at the well-known Internet port *inport*.

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 nonzero, then an auxiliary channel to a control process will be set up, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as being UNIX signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the **stderr** (unit 2 of the remote command) will be made the same as the **stdout** and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

The protocol is described in detail in `rshd`(8c).

The `rresvport` subroutine is used to obtain a socket with a privileged address bound to it. This socket is suitable for use by `rcmd` and several other routines. Privileged addresses consist of a port in the range 0 to 1023. Only the superuser is allowed to bind an address of this sort to a socket.

The `ruserok` subroutine takes a remote host's name, as returned by a `gethostent`(3n) routine, two user names and a flag indicating if the local user's name is the superuser. It then checks the files `/etc/hosts.equiv` and `.rhosts` in the user's home directory to see if the request for service is allowed. A 1 is returned if the machine name is listed in the `hosts.equiv` file, or the host and remote user name are found in the `.rhosts` file. Otherwise `ruserok` returns –1. If the superuser flag is 1, the checking of the `hosts.equiv` file is bypassed.

See Also

rlogin(1c), rsh(1c), gethostent(3n), rexec(3x), rexecd(8c), rlogind(8c), rshd(8c)

Name

rexec – return stream to a remote command

Syntax

rem = rexec(ahost, inport, user, passwd, cmd, fd2p);
char **ahost;
u_short inport;
char *user, *passwd, *cmd;
int *fd2p;

Description

The `rexec` subroutine looks up the host **ahost* using `gethostbyname,` returning –1 if the host does not exist. For further information, see `gethostent`(3n). Otherwise **ahost* is set to the standard name of the host. If a username and password are both specified, then these are used to authenticate to the foreign host. If all this fails, the user is prompted for the information.

The port *inport* specifies which well-known DARPA Internet port to use for the connection; it will normally be the value returned from the call ''getservbyname("exec", "tcp")''. For further information, see `getservent`(3n). The protocol for connection is described in detail in `rexecd`(8c).

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 nonzero, then an auxiliary channel to a control process will be set up, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel and will also accept bytes on this channel as being UNIX signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the **stderr** (unit 2 of the remote command) will be made the same as the **stdout** and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

See Also

gethostent(3n), getservent(3n), rcmd(3x), rexecd(8c)

Name

tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs – terminal independent operation routines

Syntax

char PC;
char *BC;
char *UP;
short ospeed;

tgetent(bp, name)
char *bp, *name;

tgetnum(id)
char *id;

tgetflag(id)
char *id;

char *
tgetstr(id, area)
char *id, **area;

char *
tgoto(cm, destcol, destline)
char *cm;

tputs(cp, affcnt, outc)
register char *cp;
int affcnt;
int (*outc)();

Description

These functions extract and use capabilities from the terminal capability data base `termcap`(5). These are low level routines; see `curses`(3x) for a higher level package.

The `tgetent` function extracts the entry for terminal *name* into the buffer at *bp*. The *bp* should be a character buffer of size 1024 and must be retained through all subsequent calls to `tgetnum,` `tgetflag,` and `tgetstr.` The `tgetent` function returns –1 if it cannot open the `termcap` file, 0 if the terminal name given does not have an entry, and 1 if all goes well. It will look in the environment for a TERMCAP variable. If found, and the value does not begin with a slash, and the terminal type **name** is the same as the environment string TERM, the TERMCAP string is used instead of reading the termcap file. If it does begin with a slash, the string is used as a pathname rather than `/etc/termcap.` This can speed up entry into programs that call `tgetent,` as well as to help debug new terminal descriptions or to make one for your terminal if you cannot write the file `/etc/termcap.`

The `tgetnum` function gets the numeric value of capability *id,* returning –1 if is not given for the terminal. The `tgetflag` returns 1 if the specified capability is present in the terminal's entry, 0 if it is not. The `tgetstr` function gets the string value of capability *id,* placing it in the buffer at *area,* advancing the *area* pointer. It decodes the abbreviations for this field described in `termcap`(5), except for cursor addressing and padding information.

The `tgoto` function returns a cursor addressing string decoded from *cm* to go to column *destcol* in line *destline*. It uses the external variables UP (from the **up** capability) and BC (if **bc** is given rather than **bs**) if necessary to avoid placing **\n**, **^D** or **^@** in the returned string. Programs that call `tgoto` should be sure to turn off the XTABS bit(s), because `tgoto` may now output a tab. Note that programs using termcap should in general turn off XTABS anyway, because some terminals use control I for other functions, such as nondestructive space. If a **%** sequence is given that is not understood, then `tgoto` returns ''OOPS''.

The `tputs` function decodes the leading 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 external variable *ospeed* should contain the output speed of the terminal as encoded by `stty`(3). The external variable PC should contain a pad character to be used (from the **pc** capability) if a null (**^@**) is inappropriate.

Files

/usr/lib/libtermcap.a	–ltermcap library
/etc/termcap	data base

See Also

ex(1), curses(3x), termcap(5)

X/Open Transport Interface Routines (3xti)

Insert tabbed divider here.
Then discard this sheet.

Name

intro – introduction to the X/Open Transport Interface (XTI)

Description

The X/Open Transport Interface defines a transport service interface that is independent of any specific transport provider. The interface is provided by way of a set of library functions for the C programming language.

Transport Providers

The transport layer can comprise one or more transport providers at the same time. The transport provider identifier parameter passed to the `t_open()` function determines the required transport provider.

Transport Endpoints

A transport endpoint specifies a communication path between a transport user and a specific transport provider, which is identified by a local file descriptor *(fd)*. When a user opens a transport provider identifier, a local file descriptor *fd* is returned that identifies the transport endpoint.

Synchronizing Endpoints

One process can simultaneously open several *fd*s. In synchronous mode, however the process must manage the different actions of the associated transport connections sequentially. Conversely, several processes can share the same *fd* (by `fork()` or `dup()` operations) but they have to synchronize themselves so as not to issue a function that is unsuitable to the current state of the transport endpoint.

Modes Of Service

The transport service interface supports two modes of service: connection mode and connectionless mode. A single transport endpoint cannot support both modes of service simultaneously.

The connection-mode transport service is circuit-oriented and enables data to be transferred over an established connection in a reliable, sequential manner. In contrast, the connectionless-mode transport service is message-oriented and supports data transfer in self-contained units with no logical relationship required among multiple units.

Error Handling

Two levels of error are defined for the transport interface. The first is the library error level. Each library function has one or more error returns. A return of -1 indicates a failure. An external integer, **t_errno**, which is defined in the header file **<xti.h>**, holds the specific error number when such a failure occurs. This value is set when errors occur but is not cleared on successful library calls, so it should be tested only after an error has been indicated. If implemented, a diagnostic function, **t_error**, prints out information on the current transport error. The state of the transport provider may change if a transport error occurs.

The second level of error is the operating system service routine level. A special library level error number has been defined called [TSYSERR], which is generated by each library function when the operating system service routine fails or some general error occurs. When a function sets **t_errno** to [TSYSERR], the specific system error can be accessed through the external variable **errno**.

Key For Parameter Arrays

Each XTI function description, includes an array that summarizes the content of the input and output parameter. The key is as follows:

Key	Description
x	The parameter value is meaningful (input parameter must be set before the call and output parameter must be read after the call).
(x)	The content of the object pointed by the x pointer is meaningful.
?	The parameter value is meaningful, but the parameter is oprtional.
(?)	The content of the object pointed by the ? pointer is optional.
/	The parameter value is meaningless.
=	After the call, the parameter keeps the same value as before the call.

Name

t_accept – accept a connect request

Syntax

#include <xti.h>

int t_accept(*fd, resfd, call***)**
int *fd***;**
int *resfd***;**
struct t_*call* ******call***;**

Arguments

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 *Call* argument points to a **t_call** structure that contains the following members:

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

In *call*, the members have the following meanings:

addr Specifies the address of the caller.

opt Indicates any protocol-specific parameters associated with the connection.

udata Points to any user data to be returned to the caller.

sequence Is the value returned by `t_listen()` that uniquely associates the response with a previously received connect indication.

Description

A transport user issues this function to accept a connect request. A transport user can 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 by means of `t_accept()` or `t_snddis()`. Otherwise, `t_accept()` fails and sets **t_errno** to [TBADF].

If a different transport endpoint is specified (*resfd!=fd*), the endpoint must be bound to a protocol address (if it is the same, *qlen* must be set to 0) and must be in the T_IDLE state before the `t_accept()` is issued.

For both types of endpoints, `t_accept()` fails and sets **t_errno** to [TLOOK] if there are connection indications, (for example, connect or disconnect) waiting to be received on that endpoint.

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

All the *maxlen* fields are meaningless.

Parameters	Before Call	After Call
fd	x	/
resfd	x	/
call->addr.maxlen	/	/
call->addr.len	x	/
call->addr.buf	?(?)	/
call->opt.maxlen	/	/
call->opt.len	x	/
call->opt.buf	?(?)	/
call->udata.maxlen	/	/
call->udata.len	x	/
call->udata.buf	?(?)	/
call->sequence	x	/

Return Value

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

Diagnostics

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 on the same transport endpoint on which the connect indication arrived.

[TOUTSTATE] The function was issued in the wrong sequence on the transport endpoint referenced by *fd*, or the transport endpoint referred to by *resfd* is not in the appropriate state.

[TACCES] The user does not have permission to accept a connection on the responding transport endpoint or to use the specified options.

[TBADOPT] The specified options were in an incorrect format or contained illegal information.

[TBADDATA]	The specific amount of user data 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]	The specified sequence number was invalid.
[TLOOK]	An asynchronous event has occurred on the transport endpoint referenced by *fd* and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

See Also

t_connect(3xti), t_getstate(3xti), t_listen(3xti), t_open(3xti) , t_optmgmt(3xti), t_rcvconnect(3xti)

Name

t_alloc – allocate a library structure

Syntax

#include <xti.h>

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

Arguments

fd Refers to the transport endpoint through which the newly allocated structure is passed.

struct_type Specifies the allocated structure where each structure can subsequently be used as an argument to one or more transport functions.

The *struct_type* argument must specify one of the following:

T_BIND_STR	*struct*	t_bind
T_CALL_STR	*struct*	t_call
T_OPTMGMT_STR	*struct*	t_optmgmt
T_DIS_STR	*struct*	t_discon
T_UNITDATA_STR	*struct*	t_unitdata
T_UDERROR_STR	*struct*	t_uderr
T_INFO_STR	*struct*	t_info

fields Specifies which buffers 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 (size obtained from *info_addr*).

T_OPT The *opt* field of the **t_optmgmt**, **t_call**, **t_unitdata**, or **t_uderr** structures (size obtained from *info_options*).

T_UDATA The *udata* field of the **t_call**, **t_discon**, or **t_uderr** structures (for T_CALL_STR, size is the maximum value of *info_connect* and *info_discon*; for T_DIS_STR, size is the value of *info_discon*; for T_UNITDATA_STR, size is the value of *info_tsdu*).

T_ALL All relevant fields of the given structure.

Description

The `t_alloc()` function dynamically allocates memory for the various transport function argument structures as listed under the ARGUMENTS section. This function allocates memory for the specified structure and also allocates memory for buffers referenced by the structure.

Each of the accepted structures, except `t_info()`, contains at least one field of type *struct netbuf*. For each field of this type, the user can specify that the buffer for that field should be allocated as well. The length of the buffer allocated is based on the size information returned in the `t_open()` or `t_getinfo()`.

For each field specified in *fields*, `t_alloc()` allocates memory for the buffer associated with the field and initializes the *len* field to zero and the *buf* pointer and *maxlen* field accordingly. Because the length of the buffer allocated is based on the same size information that is returned to the user on `t_open()` and `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, `t_alloc()` will be unable to determine the size of the buffer to allocate and will fail, setting **t_errno** to [TSYSERR] and **errno** to [EINVAL]. For any field not specified in *fields*, *buf* will be set to NULL and *maxlen* will be set to zero.

Use of `t_alloc()` to allocate structures helps to ensure the compatibility of user programs with future releases of the transport interface functions.

Parameters	Before Call	After Call
fd	x	/
struct_type	x	/
fields	x	/

Return Value

Upon successful completion, `t_alloc()` returns a pointer to the newly allocated structure. On failure, NULL is returned.

Diagnostics

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

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

[TNOTSUPPORT] This function is not supported by the current implementation of XTI.

[TSYSERR] A system error has occurred during execution of this function.

[TNOSTRUCTYPE] An unsupported *struct_type* has been requested.

See Also

t_free(3xti), t_getinfo(3xti), t_open(3xti)

Name

t_bind – bind an address to a transport endpoint

Syntax

#include <xti.h>

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

Arguments

fd Refers to the transport endpoint which will be associated with a protocol address.

req Points to a **t_bind** structure containing the following members:

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

The *addr* field of the `t_bind()` structure specifies a protocol address, and the *qlen* field is used to indicate the maximum number of outstanding connect indications.

ret Points to a `t_bind()` structure. See the *req* argument.

Description

This function associates a protocol address with the transport endpoint specified by *fd* and activates the transport endpoint. In connection mode, the transport provider can begin enqueuing incoming connect indications or servicing a connection request on the transport endpoint. In connectionless mode, the transport user can send or receive data units through the transport endpoint.

Parameters	Before Call	After Call
fd	x	/
req->addr.maxlen	/	/
req->addr.len	x>=0	/
req->addr.buf	x(x)	/
req->qlen	x>=0	/
ret->addr.maxlen	x	/
ret->addr.len	/	x
ret->addr.buf	x	(x)
ret->qlen	/	x>=0

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

If the requested address is not available, or if no address is specified in *req* (the *len* field of *addr* in *req* is zero), the transport provider assigns an appropriate address to be bound only if automatic generation of an address is supported and returns that address in the *addr* field of *ret*. The user can compare the addresses in *req* and *ret* to determine whether the transport provider bound the transport endpoint to a different address than that requested. In any XTI implementation, if the `t_bind()` function does not allocate a local transport address, then the returned address is always the same as the input address and the structure *req->addr* must be filled by the user before the call. If the local address is not furnished for the call (*req->addr.len=0*), the `t_bind()` returns –1 with **t_errno** set to [TNOADDR].

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

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

This function allows more than one transport endpoint to be bound to the same protocol address. The transport provider, however, must support this capability also, it is not allowable to bind more than one protocol address to the same transport endpoint. If a user binds more than one transport endpoint to the same protocol address, only one endpoint can be used to listen for connect indications associated with the protocol address.

In other words, only one `t_bind()` for a given protocol address can specify a value of *qlen* greater than zero. In this way, the transport provider can identify which transport endpoint should be notified of an incoming connect indication. If a user attempts to bind a protocol address to a second transport endpoint with a value of *qlen* greater than zero, the transport provider assigns another address to be bound to that endpoint or, if automatic generation of addresses is not supported, returns –1 and sets **t_errno** to [TADDRBUSY].

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 that same protocol address while that initial listening endpoint is active (in the data transfer phase or in the T_IDLE state). This prevents more than one transport endpoint bound to the same protocol address from accepting connect indications.

Return Value

Upon successful completion, `t_bind()` returns 0 and –1 on failure, and **t_errno** is set to indicate the error.

Diagnostics

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 changes to T_IDLE and the information to be returned in *ret* is discarded.
[TSYSERR]	A system error has occurred during execution of this function.
[TADDRBUSY]	The address requested is in use and the transport provider cannot be allocate a new address.

See Also

t_alloc(3xti), t_close(3xti), t_open(3xti), t_optmgmt(3xti), t_unbind(3xti)

Name

t_close – close a transport endpoint

Syntax

#include <xti.h>

int t_close *fd*)
int *fd*;

Arguments

fd Identifies the local transport endpoint.

Description

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

The `t_close()` function should be called from the T_UNBND state. However, this function does not check state information, so it can be called from any state to close a transport endpoint. If this occurs, the local library resources associated with the endpoint are freed automatically. In addition, `close()` is issued for that file descriptor; the `t_close()` abortives if there are no other descriptors in this or in another process that references the transport endpoint and breaks the transport connection that may be associated with that endpoint.

Parameters	Before Call	After Call
fd	x	/

Return Value

The `t_close` returns 0 on success and –1 on failure, and **t_errno** is set to indicate the error.

Diagnostics

On failure, **t_errno** is set to the following:

The specified file descriptor does not refer to a transport endpoint.

See Also

t_getstate(3xti), t_open(3xti), t_unbind(3xti)

Name

t_connect – establish a connection with another transport user

Syntax

#include <xti.h>

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

Arguments

fd Identifies the local transport endpoint where communications is established.

sndcall Specifies information needed by the transport provider to establish a connection.

rcvcall Specifies information that is associated with the newly established connection.

The *sndcall* and *rcvcall* point to a **t_call** structure that contains the following members:

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

Description

This function enables a transport user to request a connection to the specified destination transport user. This function can be issued only in the T_IDLE state.

In *sndcall*, the argument *addr* specifies the protocol address of the destination transport user. The *opt* argument presents any protocol-specific information that might be needed by the transport provider. The *udata* argument points to optional user data that may be passed to the destination transport user during connection establishment. The *sequence* argument has no meaning for this function.

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

The *opt* argument permits users to define the options that can be passed to the transport provider. These options are specific to the underlying protocol of the transport provider. The user can choose not to negotiate protocol options by setting the *len* field of *opt* to zero. In this case, the provider may use default options.

Parameters	Before Call	After Call
resfd	x	/
sndcall->addr.maxlen	/	/
sndcall->addr.len	x	/
sndcall->addr.buf	x(x)	/
sndcall->opt.maxlen	/	/
sndcall->opt.len	x	/
sndcall->opt.buf	?(?)	/
sndcall->udata.maxlen	/	/
sndcall->udata.len	x	/
sndcall->udata.buf	?(?)	/
sndcall->sequence	/	/
rcvcall->addr.maxlen	x	/
rcvcall->addr.len	/	x
rcvcall->addr.buf	x	(x)
rcvcall->opt.maxlen	x	/
rcvcall->opt.len	/	x
rcvcall->opt.buf	x	(x)
rcvcall->udata.maxlen	x	/
rcvcall->udata.len	/	x
rcvcall->udata.buf	x	(?)
rcvcall->sequence	/	/

If used, **sndcall->opt.buf** structure must point to the corresponding options structures (**isoco_options**, **isocl_options** or **tcp_options**). The *maxlen* and *buf* fields of the **netbuf** structure pointed by *rcvcalladdr* and *rcvcall->opt* must be set before the call.

The *udata* argument enables the caller to pass user data to the destination transport 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()`. If the *len* of *udata* is zero in *sndcall*, no data are sent to the destination transport user.

On return, the *addr*, *opt*, and *udata* fields of *rcvcall* updates 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* can be NULL, in which case no information is given to the user on return from `t_connect()`.

By default, `t_connect()` executes in synchronous mode and waits for the destination user's response before returning control to the local user. A successful return (that is, a return value of zero) indicates that the requested connection has been established. However, if O_NONBLOCK is set by means of `t_open()` or `fcntl()`, `t_connect()` executes in asynchronous mode. In this case, the call waits for the remote user's response but returns control immediately to the local user and returns –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.

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.

Diagnostics

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.
[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.
[TACCES]	The user does not have permission to use the specified address or options.
[TBADOPT]	The specified protocol options were in an incorrect format or contained illegal information.
[TBADADDR]	The specified protocol address was 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.
[TBUFOVFLW]	The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument. If executed in synchronous mode, the provider's state, as seen by the user, changes to T_DATAXFER, and the connect indication information to be returned in *rcvcall* is discarded.
[TLOOK]	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

See Also

t_accept(3xti), t_alloc(3xti), t_getinfo(3xti), t_listen(3xti), t_open(3xti), t_optmgmt(3xti), t_rcvconnect(3xti)

Name

t_error – produces error message

Syntax

#include <xti.h>

int t_error(*errmsg***)**
char **errmsg*;
extern char **t_errlist[]*;
extern int *t_nerr*;

Arguments

errmsg Is a user-supplied error message that gives context to the error.

Description

The `t_error()` function produces a message on the standard error output that describes the last error encountered during a call to a transport function.

The `t_error()` function prints the user-supplied error message followed by a colon and a standard error message for the current error defined in **t_errno**. If **t_errno** is [TSYSERR], `t_error()` also prints a standard message for the current value contained in **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 *t_nerr* is the largest message number provided for in the *t_errlist* table.

The **t_errno** variable is set only when an error occurs and is not cleared on successful calls.

Parameters	Before Call	After Call
errmsg	x	/

Examples

If a `t_connect()` function fails on transport endpoint *fd2* because a bad address was given, the following call may follow the failure:

```
t_error ("t_connect failed on fd"):
```

The diagnostic message to be printed would look like:

```
t_connect failed on fd2: Incorrect transport address format
```

where "Incorrect transport address format" identifies the specific error that occurred, and "t_connect failed on fd2" tells the user which function failed on which transport endpoint.

Return Value

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

Diagnostics

On failure, **t_errno** is set to the following:

[TNOTSUPPORT]	This function is not supported by the current implementation of XTI.

Name

t_free – free a library structure

Syntax

#include <xti.h>

int t_free(*ptr, struct_type*)
char **ptr*;
int *struct_type*;

Arguments

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_STR`	*struct*	**t_bind;**
`T_CALL_STR`	*struct*	**t_call**
`T_OPTMGMT_STR`	*struct*	**t_optmgmt**
`T_DIS_STR`	*struct*	**t_discon**
`T_UNITDATA_STR`	*struct*	**t_unitdata**
`T_UDERROR_STR`	*struct*	**t_uderr**
`T_INFO_STR`	*struct*	**t_info**

Each of these structures is used as an argument to one or more transport functions.

Description

The `t_free()` function frees memory previously allocated by `t_alloc()`. This function frees memory for the specified structure and also frees memory for buffers referenced by the structure.

Parameters	Before Call	After Call
ptr	x	/
struct_type	x	/

The `t_free()` function checks 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 NULL, `t_free()` does not attempt to free memory. After all buffers are freed, `t_free()` frees the memory associated with the structure pointed to by *ptr*.

Results are undefined if *ptr* or any of the *buf* pointers points to a block of memory not previously allocated by `t_alloc()`.

Return Value

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

Diagnostics

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

[TNOTSUPPORT]	This function is not supported by the current implementation of XTI.
[TSYSERR]	A system error has occurred during execution of this function.

See Also

t_alloc(3xti)

Name

t_getinfo – get protocol-specific service information

Syntax

#include <xti.h>

int t_getinfo(*fd, info***)**
int *fd***;**
struct t_info **info***;**

Arguments

fd Identifies the file descriptor that is associated with the underlying transport protocol from which the current characteristics are to be returned.

info Specifies the structure that is used to return the same information returned by `t_open()`. Points to a **t_info** structure which contains the following members:

long *addr***;**	/* max size of the transport protocol address */
long *options***;**	/* max number of bytes of protocol-specific options */
long *tsdu***;**	/* max size of a transport service data unit (TSDU) */
long *etsdu***;**	/* max size of an expedited transport service data unit (ETSDU) */
long *connect***;**	/* max amount of data allowed on connection establishment functions */
long *discon***;**	/* max amount of data allowed on `t_snddis()` and `t_rcvdis()` functions */
long *servtype***;**	/* service type supported by the transport provider */

The values of the fields have the following meanings:

addr A value greater than or equal to zero indicates the maximum size of a transport protocol address; a value of –1 specifies that there is no limit on the address size; and a value of –2 specifies that the transport provider does not provide user access to transport protocol addresses.

options A value greater than or equal to zero indicates the maximum number of bytes of protocol-specific options supported by the provider; a value of –1 specifies that there is no limit on the option size and a value of –2 specifies that the transport provider does not support user-settable options.

tsdu A value greater than zero specifies the maximum size of a transport service data unit (TSDU); a value of zero specifies that the transport provider does not support the concept of TSDU, although it does support the sending of a data stream with no logical boundaries preserved across a connection; a value of –1 specifies that there is no limit on the size of a TSDU and a value of –2 specifies that the transfer of normal data is not supported by the transport provider.

etsdu A value greater than zero specifies the maximum size of an expedited transport service data unit (ETSDU); a value of zero specifies that the transport provider does not support the concept of ETSDU, although it 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 ETSDU; and a value of –2 specifies that the transfer of expedited data is not supported by the transport provider.

connect A value greater than or equal to zero specifies the maximum amount of data that may be associated with connection establishment functions; a value of –1 specifies that there is no limit on the amount of data sent during connection establishment; and a value of –2 specifies that the transport provider does not allow data to be sent with connection establishment functions.

discon A value greater than or equal to zero specifies the maximum amount of data that may be associated with the `t_snddis()` and `t_rcvdis()` functions; a value –1 specifies that there is no limit on the amount of data sent with these abortive release functions; and a value of –2 specifies that the transport provider does not allow data to be sent with the abortive release functions.

servtype This field specifies the service type supported by the transport provider, as described.

If a transport user is concerned with protocol independence, the sizes may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the `t_alloc()` function can be used to allocate these buffers. An error results 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()` returns –2 for ETSDU, `connect` and `discon`.

Description

This function returns the current characteristics of the underlying transport protocol associated with file descriptor *fd*. The *info* structure is used to return the same information returned by `t_open()`. This function enables a transport user to access this information during any phase of communications.

Parameters	Before Call	After Call
fd	x	/
info->addr	/	x
info->options	/	x
info->tsdu	/	x
info->etsdu	/	x
info->connect	/	x
info->discon	/	x
info->sertype	/	x

Return Value

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

Diagnostics

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

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

[TNOTSUPPORT] This function is not supported by the current implementation of XTI.

[TSYSERR] A system error has occurred during execution of this function.

See Also

t_alloc(3xti), t_open(3xti)

t_getstate (3xti)

Name

t_getstate – get the current state

Syntax

#include <xti.h>

int t_getstate(*fd*)
int *fd*;

Arguments

fd Identifies the local transport endpoint the current state is returned from.

Description

The `t_getstate()` function returns the current state of the transport provider associated with the transport endpoint specified by *fd*.

Parameters	Before Call	After Call
fd	x	/

Return Value

Upon successful completion, `t_getstate()` returns the current state. On failure, a value of –1 is 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 to send an orderly release request)

If the provider is undergoing a state transition when `t_getstate()` is called, the function fails.

Diagnostics

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 has been passed to the call.

[TSTATECHNG] The transport provider is undergoing a transient state change.

[TNOTSUPPORT]	This function is not supported by the current implementation of XTI.
[TSYSERR]	A system error has occurred during execution of this function.

See Also

t_open(3xti)

t_listen (3xti)

Name

t_listen – listen for a connect request

Syntax

#include <xti.h>

int t_listen(*fd, call***)**
int ***fd*****;**
struct t_call ****call*****;**

Arguments

fd Identifies the local transport endpoint where the connect indication arrived.

call Contains information describing the connect indication. The *call* points to a **t_call** structure which contains the following members:

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

The members of the **t_call** structure have the following meanings:

addr Returns the protocol address of the calling transport user.

udata Returns any user data sent by the caller on the connect request.

sequence Identifies the returned connect indication with a unique number. The value of *sequence* enables the user to listen for multiple connect indications before responding to any of them.

Because 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.

Description

This function listens for a connect request from a calling transport user. The *fd* identifies the local transport endpoint where connect indications arrive. On return, *call* contains information describing the connect indication.

By default, `t_listen` executes in synchronous mode and waits for a connect indiction to arrive before returning to the user. However, if O_NONBLOCK is set by means of `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].

Parameters	Before Call	After Call
fd	x	/
call->addr.maxlen	x	/
call->addr.len	/	x
call->addr.buf	x	(x)
call->opt.maxlen	x	/
call->opt.len	/	x
call->opt.buf	x	(x)
call->udata.maxlen	x	/
call->udata.len	/	x
call->udata.buf	x	(?)
call->sequence	/	x

Return Value

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

Diagnostics

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 on the transport endpoint referenced by *fd*.

[TBADQLEN] The *qlen* of the endpoint referenced by *fd* is zero.

[TBUFOVFLW] The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument. The provider's state, as seen by the user, changes to T_INCON, and the connect indication information to be returned in *call* is discarded. The value of *sequence* returned can be used to do a `t_snddis()`.

[TNODATA] O_NONBLOCK was set, but no connect indications had been queued.

[TLOOK] An asynchronous event has occurred on the transport endpoint and requires immediate attention.

[TNOTSUPPORT] This function is not supported by the underlying transport provider.

[TSYSERR] A system error has occurred during execution of this function.

See Also

fcntl(2), t_accept(3xti), t_alloc(3xti), t_bind(3xti), t_connect(3xti), t_open(3xti), t_optmgmt(3xti), t_rcvconnect(3xti)

Name

t_look – look at the current event on a transport endpoint

Syntax

#include <xti.h>

int t_look(*fd***)**
int *fd***;**

Arguments

fd Identifies the transport endpoint where the current event is returned.

Description

This function returns the current event on the transport endpoint specified by *fd*. This function enables a transport provider to notify a transport user of an asynchronous event when the user is issuing functions in synchronous mode. Certain events require immediate notification of the user and are indicated by a specific error, [TLOOK], on the current or next function to be executed.

This function also enables a transport user to poll a transport endpoint periodically for asynchronous events.

Parameters	Before Call	After Call
fd	x	/

Return Value

Upon successful completion, `t_look()` returns a value that indicates which of the allowable events has occurred or returns zero if no event exists. One of the following events is returned:

T_LISTEN Connection indication received

T_CONNECT Connect confirmation received

T_DATA Normal data received

T_EXDATA Expedited data received

T_DISCONNECT Disconnect received

T_UDERR Datagram error indication

T_ORDREL Orderly release indication

T_GODATA Flow control restrictions on normal data flow have been lifted. Normal data can be sent again.

T_GOEXDATA Flow control restrictions on expedited data flow have been lifted. Expedited data can be sent again.

On failure, –1 is returned, and **t_errno** is set to indicate the error.

Diagnostics

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

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

See Also

t_open(3xti), t_snd(3xti), t_sndudata(3xti)

Name

t_open – establish a transport endpoint

Syntax

#include <xti.h>

#include <fcntl.h>
int t_open(*name, oflag, info***)**
char **name***;**
int *oflag***;**
struct t_info **info***;**

Arguments

name Points to a transport provider identifier.

oflag Identifies any open flags as in `open()`. The *oflag* argument is constructed from O_RDWR optionally ORed with O_NONBLOCK. These flags are defined by the header file **<fcntl.h>**.

info Returns various default characteristics of the underlying transport protocol by setting fields in the *info* structure. This argument points to a `t_info()` structure that contains the following members:

long *addr*	/* max size of the transport protocol address */
long *options*	/* max number of bytes of protocol specific options */
long *tsdu*	/* max size of a transport service data unit (TSDU) */
long *etsdu*	/* max size of expedited transport service data unit (ETSDU) */
long *connect*	/* max amount of data allowed on connection establishment functions */
long *discon*	/* max amount of data allowed on `t_snddis()` and `t_rcvdis()` functions */
long *servtype*	/* service type supported by the transport provider */

The values of the fields have the following meanings:

addr A value greater than or equal to zero indicates the maximum size of a transport protocol address; a value of –1 specifies that there is no limit on the address size; and a value of –2 specifies that the transport provider does not provide user access to transport protocol addresses.

options A value greater than or equal to zero indicates the maximum number of bytes of protocol-specific options supported by the provider; a value of -1 specifies that there is no limit on the option size; and a value of -2 specifies that the transport provider does not support user-settable options.

tsdu A value greater than zero specifies the maximum size of a transport service data unit (TSDU); a value of zero specifies that the transport provider does not support the concept of TSDU; although it does support the sending of a data stream with no logical boundaries preserved across a connection; a value of –1 specifies that there is no limit on the size of an ETSDU; and a value of –2 specifies that the transfer of normal data is not supported by the transport provider.

etsdu A value greater than zero specifies the maximum size of an expedited transport service data unit (ETSDU); a value zero specifies that the transport provider does not support the concept of ETSDU, although it does support the sending of an expedited data stream with no logical boundaries preserved across a connection; a value of –1 specifies that there is no limit on the size of an ETSDU; and a value –2 specifies that the transfer of expedited data is not supported by the transport provider.

connect A value greater than or equal to zero specifies the maximum amount of data that may be associated with connection establishment functions; a value of –1 specifies that there is no limit on the amount of data sent during connection establishment; and a value of –2 specifies that the transport provider does not allow data to be sent with connection establishment functions.

discon A value greater than or equal to zero specifies the maximum amount of data that may be associated with the `t_snddis()` and `t_rcvdis()` functions; a value of –1 specifies that there is no limit on the amount of data sent with these abortive release functions; and a –2 specifies that the transport provider does not allow data to be sent with abortive release functions.

servtype This field specifies the service type supported by the transport provider, as described.

If a transport user is concerned with protocol independence, the sizes can be accessed to determine how large the buffers must be to hold each piece of information. Alternately, 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()` returns –2 for *etsdu*, *connect*, and *discon*.

A single transport endpoint may support only one of the above services at one time. If *info* is set to NULL by the transport user, no protocol information is returned by `t_open()`.

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 supplying a transport provider identifier that indicates a particular transport provider, that is a transport protocol, and returns a file descriptor that identifies that endpoint.

The `t_open()` function returns a file descriptor that is used by all subsequent functions to identify that particular local transport endpoint.

Parameters	Before Call	After Call
name	x	/
oflag	x	/
info->addr	/	x
info->options	/	x
info->tsdu	/	x
info->etsdu	/	x
info->connect	/	x
info->discon	/	x
info->servtype	/	x

Return Value

Upon successful completion, `t_open()` returns a file descriptor. On failure, -1 is returned, and **t_errno** is set to indicate the error.

Diagnostics

On failure, **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.

See Also

open(2)

Name

t_optmgmt – manage options for a transport endpoint

Syntax

#include <xti.h>

int t_optmgmt(*fd, req, ret***)**
int *fd***;**
struct t_optmgmt **req***;**
struct t_optmgmt **ret***;**

Arguments

fd Identifies a bound transport endpoint.

req Points to a **t_optmgmt** structure. See also *ret* argument.

ret Points to a **t_optmgmt** structure containing the following members:

```
struct netbuf opt;
long flags;
```

The meanings of the fields are as follows:

opt Identifies protocol options.

flags
Specifies the action to take with these options.

The options are represented by a **netbuf** structure in a manner similar to the address in `t_bind()`. The *req* argument is used to request a specific action of the provider and to send options to the provider. The *len* field specifies the number of bytes in the options. The *buf* field points to the options buffer, and the *maxlen* field has no meaning for the *req* argument. The transport provider can return options and flag values to the user through *ret*. For *ret*, *maxlen* specifies the maximum size of the options buffer, and *buf* points to the buffer where the options are to be placed. On return, *len* specifies the number of bytes of options returned. The *maxlen* field has no meaning for the *req* argument, but must be set in the *ret* argument to specify the maximum number of bytes the option buffer can hold. The actual structure and content of the options is imposed by the transport provider.

The *flags* field of *req* must specify one of the following actions:

T_NEGOTIATE This action enables the user to negotiate the values of the options specified in *req* with the transport provider. The transport provider evaluates the requested options and negotiates the values, returns the negotiated values through *ret*.

T_CHECK This action enables the user to verify whether the options specified in *req* are supported by the transport provider. On return, the *flags* field of *ret* has either T_SUCCESS or T_FAILURE set to indicate to the user whether options are supported. These *flags* are only meaningful for the T_CHECK request.

T_DEFAULT This action enables a user to retrieve the default options supported by the transport provider into the *opt* field of *ret*. In *req*, the *len* field of *opt* must be zero and the *buf* field may be NULL.

Description

The `t_optmgmt()` function enables a transport user to receive, verify, or negotiate protocol options with the transport provider.

If issued as part of the connectionless-mode service, `t_optmgmt()` may block due to flow control constraints. That is, the function does not complete until the transport provider has processed all previously sent data units.

Parameters	Before Call	After Call
fd	x	/
req->opt.maxlen	/	/
req->opt.len	x	/
req->opt.buf	x(x)	/
req->flags	x	/
ret->opt.maxlen	x	/
ret->opt.len	/	x
ret->opt.buf	x	(x)
ret->flags	/	x

Return Value

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

Diagnostics

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.

[TACCES] The user does not have permission to negotiate the specified options.

[TBADOPT] The specified protocol options were in an incorrect format or contained illegal information.

[TBADFLAG] An invalid flag was specified.

[TBUFOVFLW] The number of bytes allowed for an incoming argument is not sufficient to store the value of that argument. The information to be returned in *ret* is discarded.

[TNOTSUPPORT] This function is not supported by the current implementation of XTI.

[TSYSERR] A system error has occurred during execution of this function.

See Also

t_accept(3xti), t_alloc(3xti), t_connect(3xti), t_getinfo(3xti), t_listen(3xti), t_open(3xti), t_rcvconnect(3xti)

t_rcv (3xti)

Name

t_rcv – receive data or expedited data sent over a connection

Syntax

#include <xti.h>

int t_rcv(*fd, buf, nbytes, flags***)**
int *fd***;**
char **buf***;**
unsigned *nbytes***;**
int **flags***;**

Arguments

fd	Identifies the local transport endpoint through which data arrives.
buf	Points to a receive buffer where user data is placed.
nbytes	Specifies the size of the receive buffer.
flags	Specifies optional flags. Can be set on return from `t_rcv()`.

Description

This function receives either normal or expedited data.

By default, `t_rcv()` operates in synchronous mode and waits for data to arrive if none is currently available. However, if O_NONBLOCK is set (by means of `t_open()` or `fcntl()`, `t_rcv()` executes in asynchronous mode and fails if no data is available.

On return from the call, if T_MORE is set in *flags* this indicates that there is more data and the current transport service data unit (TSDU) or expedited transport service data (ETSDU) must be received in multiple `t_rcv()` calls. Each `t_rcv()` with the T_MORE flag set indicates that another `t_rcv()` must follow immediately to get more data from the current TSDU. The end of the TSDU is identified by the return of a `t_rcv()` call with the T_MORE flag not set. If the transport provider does not support the concept of a TSDU as indicated in the *info* argument on return from `t_open()` or `t_getinfo()`, the T_MORE flag is not meaningful and should be ignored.

On return, the data returned is expedited data if T_EXPEDITED is set in *flags*. If the number of bytes of expedited data exceeds *nbytes*, `t_rcv()` sets T_EXPEDITED and T_MORE on return from the initial call. Subsequent calls to retrieve the remaining ETSDU 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 is 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.

Parameters	Before Call	After Call
fd	x	/
buf	x	(x)
nbytes	x	/
flags	/	x

Return Value

Upon successful completion, `t_rcv()` returns the number of bytes received. On failure, a value of -1 is returned, and **t_errno** is set to indicate the error.

Diagnostics

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 on the transport endpoint referenced by *fd*.
[TNODATA]	O_NONBLOCK was set, but no data is currently available from the transport provider.
[TLOOK]	An asynchronous event has occurred on the transport endpoint and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

See Also

fcntl(2), t_getinfo(3xti), t_look(3xti), t_open(3xti), t_snd(3xti)

t_rcvconnect (3xti)

Name

t_rcvconnect – receive the confirmation from a connect request

Syntax

#include <xti.h>

int t_rcvconnect(*fd, call*)
int *fd*;
struct t_call **call*;

Arguments

fd Identifies the local transport endpoint where communications is established.

call Contains information associated with the newly established connection. *Call* points to a t_*call* structure that contains the following members:

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

The members of the *t_call* structure have the following meanings:

addr Returns the protocol address associated with the responding transport endpoint.

opt Presents any protocol-specific information associated with the transport endpoint.

udata Points to any optional user data that may be returned by the destination transport user during connection establishment.

sequence Has no meaning for this function.

Description

This function enables a calling transport user to determine the status of a previously sent connect request. Is used in conjunction with `t_connect()` to establish a connection in asynchronous mode. The connection is established on successful completion of 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* can be NULL, in which case no information is given to the user on return from `t_rcvconnect()`. By default, `t_rcvconnect()` executes in synchronous mode and waits for the connection to be established before returning. On return, the *addr*, *opt*, and *udata* fields reflect values associated with the connection.

Parameters	Before Call	After Call
fd	x	/
call->addr.maxlen	x	/
call->addr.len	/	x
call->addr.buf	x	(x)
call->opt.maxlen	x	/
call->opt.len	/	x
call->opt.buf	x	(x)
call->udata.maxlen	x	/
call->udata.len	/	x
call->udata.buf	x	(?)
call->sequence	/	/

If O_NONBLOCK is set by means of `t_open()` or `fcntl()`, `t_rcvconnect()` executes in asynchronous mode and reduces to a poll for existing connect confirmations. If none is available, `t_rcvconnect()` fails and returns immediately without waiting for the connection to be established. The `t_rcvconnect()` function must be reissued at a later time to complete the connection establishment phase and retrieve the information returned to *call*.

Return Value

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

Diagnostics

On failure, `t_errno()` is set to one of the following:

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

[TBUFOVFLW] The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument. The connect information to be returned in *call* is discarded. The provider's state, as seen by the user, is changed to DATAXFER.

[TNODATA] O_NONBLOCK was set, but a connect confirmation has not yet arrived.

[TLOOK] An asynchronous event has occurred on the transport connection 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 transport endpoint referenced by *fd*.

[TSYSERR] A system error has occurred during execution of this function.

See Also

t_accept(3xti), t_alloc(3xti), t_bind(3xti), t_connect(3xti), t_listen(3xti), t_open(3xti), t_optmgmt(3xti)

Name

t_rcvdis – retrieve information from disconnect

Syntax

#include <xti.h>

int t_rcvdis(*fd, discon***)**
int *fd***;**
struct t_discon **discon***;**

Arguments

fd Identifies the local transport endpoint.

discon Points to a **t_discon** structure containing the following members:

```
struct netbuf udata;
int reason;
int sequence:
```

The members of the *t_discon* struct have the following meanings:

udata Identifies any user data that was sent with the disconnect.

reason Specifies the reason for the disconnect through a protocol-dependent reason code.

sequence Identifies an outstanding connect indication with which the connection is associated. The *sequence* field 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.

Description

This function is used to identify the cause of a disconnect and to retrieve any user data sent with the disconnect.

If a user does not care if there is incoming data and does not need to know the value of *reason* or *sequence*, *discon* may be NULL and any user data associated with the disconnect is discarded. However, if a user has retrieved more than one outstanding connect indication, by means of `t_listen()` and *discon* is NULL, the user will be unable to identify with which connect indication the disconnect is associated.

Parameters	Before Call	After Call
fd	x	/
discon->udata.maxlen	x	/
discon->udata.len	/	x
discon->udata.buf	x	(?)
discon->reason	/	x
discon->sequence	/	?

Return Value

Upon successful completion, a value of 0 is returned. On failure, a value of -1 is returned, and `t_errno()` is set to indicate the error.

Diagnostics

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

[TBADF]	The specified file descriptor *fd* does not refer to a transport endpoint.
[TOUTSTATE]	The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.
[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* > 1, it remains in state T_INCON; otherwise, the endpoint state is set to T_IDLE. The disconnect indication information to be returned in *discon* will be discarded.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

See Also

t_alloc(3xti), t_connect(3xti), t_listen(3xti), t_open(3xti), t_snddis(3xti)

Name

t_rcvrel – acknowledge receipt of an orderly release indication

Syntax

#include <xti.h>

int t_rcvrel(*fd*)
int *fd*;

Arguments

fd Identifies the local transport endpoint.

Description

This function is used to acknowledge receipt of an orderly release indication. After receipt of this indication, the user cannot attempt to receive more data, because such an attempt will block forever. However, the user can continue to send data over the connection if `t_sndrel()` has not been issued by the user.

This function is an optional service of the transport provider, and is only supported if the transport provider returned service type T_COTS_ORD on `t_open()` or `t_getinfo()`.

Parameters	Before Call	After Call
fd	x	/

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.

Diagnostics

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 on the transport endpoint referenced by *fd*.

[TNOREL] No orderly release indication currently exists on the specified transport endpoint.

[TLOOK] An asynchronous event has occurred on the transport endpoint and requires immediate attention.

[TNOTSUPPORT] This function is not supported by the underlying transport provider.

[TSYSERR] A system error has occurred during execution of this function.

See Also

t_getinfo(3xti), t_open(3xti), t_sndrel(3xti)

Name

t_rcvudata – receive a data unit

Syntax

#include <xti.h>

int t_rcvudata(*fd, unitdata, flags***)**
int *fd*;
struct t_unitdata **unitdata*;
int **flags*:

Arguments

fd Identifies the local transport endpoint through which data is received.

unitdata Holds information associated with the received data unit. The *unitdata* argument points to a **t_unitdata** structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata
```

On return from this call, the members have the following meanings:

addr Specifies the protocol address of the sending unit.

opt Identifies protocol-specific options that were associated with this data unit.

udata Specifies the user data that was received.

flags Set on return to indicate that the complete data unit was not received.

Description

This function is used in connectionless mode to receive a data unit from another transport user.

By default, `t_rcvudata()` operates in synchronous mode waits for a data unit to arrive if none is currently available. However, if O_NONBLOCK is set by means of `t_open()` or `fcntl()`, *udata* executes in asynchronous mode and fails if no data units are available.

The *maxlen* field of *addr*, *opt*, and *udata* must be set before issuing this function to indicate the maximum size of the buffer for each.

If the buffer defined in the *udata* field of *unitdata* is not large enough to hold the current data unit, the buffer fills and T_MORE sets in *flags* on return to indicate that another `t_rcvudata()` should be issued to retrieve the rest of the data unit. Subsequent `t_rcvudata()` calls return zero for the length of the address and options until the full data unit has been received.

Parameters	Before Call	After Call
fd	x	/
unitdata->addr.maxlen	x	/
unitdata->addr.len	/	x
unitdata->addr.buf	x	(x)
unitdata->opt.maxlen	x	/
unitdata->opt.len	/	x
unitdata->opt.buf	x	(x)
unitdata->udata.maxlen	x	/
unitdata->udata.len	/	x
unitdata->udata.buf	x	(x)
flags	/	x

Return Value

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

Diagnostics

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 on the transport endpoint referenced by *fd*.

[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 options is not sufficient to store the information. The unit data information to be returned in *unitdata* is discarded.

[TLOOK] An asynchronous event has occurred on the transport endpoint and requires immediate attention.

[TNOTSUPPORT] This function is not supported by the underlying transport provider.

[TSYSERR] A system error has occurred during execution of this function.

See Also

fcntl(2), t_alloc(3xti), t_open(3xti), t_rcvuderr(3xti), t_sndudata(3xti)

Name

t_rcvuderr – receive a unit error indication

Syntax

#include <xti.h>

int t_rcvuderr(*fd, uderr***)**
int *fd***;**
struct t_uderr **uderr***;**

Arguments

fd Identifies the local transport endpoint through which the error report is received.

uderr Points to a **t_uderr** structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
long error;
```

On return from this call, the members have the following meanings:

addr Specifies the destination protocol address of the erroneous data unit.

opt Identifies protocol-specific options that were associated with the data unit.

error Specifies a protocol-dependent error code.

Description

This function is used in connectionless mode to receive information concerning an error on a previously sent data unit and should 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 *maxlen* field of *addr* and *opt* must be set before issuing this function to indicate the maximum size of the buffer for each.

If the user does not care to identify the data unit that produced an error, *uderr* may be set to NULL, and `t_rcvuderr()` simply clears the error indication without reporting any information to the user.

Parameters	Before Call	After Call
fd	x	/
uderr->addr.maxlen	x	/
uderr->addr.len	/	x
uderr->addr.buf	x	(x)
uderr->opt.maxlen	x	/
uderr->opt.len	/	x
uderr->opt.buf	x	(x)
uderr->error	/	x

Return Value

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

Diagnostics

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

[BADF] The specified file descriptor does not refer to a transport endpoint.

[TNOUDERR] No unit data error indication currently exists on the specified transport endpoint.

[TBUFOVFLW] The number of bytes allocated for the incoming protocol address or options is not sufficient to store the information. The unit data error information to be returned in *uderr* will be discarded.

[TNOTSUPPORT] This function is not supported by the underlying transport provider.

[TSYSERR] A system error has occurred during execution of this function.

See Also

t_rcvudata(3xti), t_sndudata(3xti)

Name

t_snd – send data or expedited data over a connection

Syntax

#include <xti.h>

int t_snd(*fd, buf, nbytes, flags***)**
int *fd***;**
char **buf***;**
unsigned *nbytes***;**
int *flags*;

Arguments

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.

flags Specifies any optional flags described below:

T_EXPEDITED
If set in *flags*, the data is sent as expedited data and is 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()` follows with more data for the current TSDU. The end of 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 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.

Description

This function is used to send either normal or expedited data.

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 by means of `t_open()` or `fcntl()`, `t_snd()` executes in asynchronous mode, and fails immediately, if there are flow control restrictions. The process can arrange to be informed when the flow control restrictions are cleared by means of `t_look()`.

On successful completion, `t_snd()` returns the number of bytes accepted by the transport provider. Normally, this equals the number of bytes specified in *nbytes*. However, if O_NONBLOCK is set, it is possible that only part of the data is accepted by the transport provider. In this case, `t_snd()` returns 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, the `t_snd()` returns –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 results in protocol error (see [TSYSERR] under the DIAGNOSTICS section).

The error [TLOOK] may be returned to inform the process that an event, such as a **disconnect**, has occurred.

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.

Parameters	Before Call	After Call
fd	x	/
buf	x(x)	/
nbytes	x	/
flags	x	/

Return Value

Upon successful completion, **t_errno** returns the number of bytes accepted by the transport provider. On failure, a value of –1 is returned, and **t_errno** is set to indicate the error.

In asynchronous mode, if the number of bytes accepted by the transport provider is less than the number of bytes requested, this may indicate that the transport provider is blocked due to flow control.

Diagnostics

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 on the transport endpoint referenced by *fd*.
[TBADFLAG]	An invalid flag was specified.
[TFLOW]	O_NONBLOCK was set, but the flow control mechanism prevented the transport provider from accepting any data at this time.
[TBADDATA]	Illegal amount of data: zero octets is not supported.

[TLOOK]	An asynchronous event has occurred on the transport endpoint.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function. A protocol error may not cause **t_errno** to fail until a subsequent access of the transport endpoint.

See Also

`t_getinfo`(3xti), `t_open`(3xti), `t_rcv`(3xti)

t_snddis (3xti)

Name

t_snddis – send user-initiated disconnect request

Syntax

#include <xti.h>

int t_snddis(*fd, call***)**
int *fd***;**
struct t_call**call***;**

Arguments

fd Identifies the local transport endpoint of the connection.

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;
```

Description

This function is used to initiate an abortive release on an already established connection or to reject a connect request.

Parameters	Before Call	After Call
fd	x	/
call->addr.maxlen	x	/
call->addr.len	x	/
call->addr.buf	/	/
call->opt.maxlen	/	/
call->opt.len	/	/
call->opt.buf	/	/
call->udata.maxlen	/	/
call->udata.len	x	/
call->udata.buf	?(?)	/
call->sequence	?	/

The values in *call* have different semantics, depending on the context of the call to `t_snddis()`. When rejecting a connect request, *call* must be non-NULL and contain a valid value of *sequence* to uniquely identify the rejected connect indication to the transport provider. The *sequence* parameter is only meaningful, if the transport connection is in the T_INCON state. The *addr* and *opt* fields of *call* are ignored. In all other cases, *call* needs be used only 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* can be NULL.

The *udata* field 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 the *udata* is zero, no data is sent to the remote user.

Return Value

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

Diagnostics

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 on the transport endpoint referenced by *fd*.
[TBADDATA]	The amount of user data specified was not within the bounds allowed by the transport provider. Some outbound data queued for this endpoint can be lost.
[TBADSEQ]	An invalid sequence number was specified, or a NULL call structure was specified when rejecting a connect request. Some outbound data queued for this endpoint can be lost.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

See Also

t_connect(3xti), t_getinfo(3xti), t_listen(3xti), t_open(3xti)

Name

t_sndrel – initiate an orderly release

Syntax

#include <xti.h>

int t_sndrel(*fd***)**
int *fd***;**

Arguments

fd Identifies the local transport endpoint where the connection exists.

Description

This function is used to initiate an orderly release of a transport connection and indicates to the transport provider that the transport user has no more data to send. After issuing `t_sndrel()`, the user can not send any more data over the connection. However, a user can continue to receive data if an orderly indication has not been received.

This function is an optional service of the transport provider and is only supported if the transport provider returned service type T_COTS_ORD on `t_open()` or `t_getinfo()`.

Parameters	Before Call	After Call
fd	x	/

Return Value

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

Diagnostics

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 on the transport endpoint referenced by *fd.*

[TFLOW] O_NONBLOCK was set, but the flow control mechanism prevented the transport provider from accepting the function at this time.

[TLOOK] An asynchronous event has occurred on the transport endpoint referenced by *fd* and requires immediate attention.

[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

See Also

t_getinfo(3xti), t_open(3xti), t_rcvrel(3xti)

Name

t_sndudata – send a data unit

Syntax

#include <xti.h>

int t_sndudata(*fd, unitdata***)**
int *fd***;**
struct t_unitdata **unitdata*;

Arguments

fd Identifies the local transport endpoint through which data will be sent.

unitdata Points to a **t_unitdata** structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

The members have the following meanings:

addr Specifies the protocol address of the destination user.

opt Identifies protocol-specific options that the user wants associated with the request.

udata Specifies the user data to be sent.

Description

This function is used in connectionless mode to send a data unit to another transport user.

Parameters	Before Call	After Call
fd	x	/
unitdata->addr.maxlen	/	/
unitdata->addr.len	x	/
unitdata->opt.maxlen	/	/
unitdata->opt.len	x	/
unitdata->opt.buf	?(?)	/
unitdata->udata.maxlen	/	/
unitdata->udata.len	x	/
unitdata->udata.buf	x(x)	/

If the *len* field of *udata* is zero, and sending of zero octets is not supported by the underlying transport service, the `t_sndudata()` returns –1 with **t_errno** set to [TBADDATA].

By default, `t_sndudata()` operates in synchronous mode and may wait if flow control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if O_NONBLOCK is set by means of `t_open()` or `fcntl()`, `t_sndudata()` executes in asynchronous mode and fails under such conditions. The process can arrange to be notified of the clearance of a flow control restriction by means of `t_look()`.

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()`, the provider generates a protocol error. See [TSYSERR] under the DIAGNOSTICS section. If `t_sndudata()` is issued before the destination user has activated its transport endpoint, the data unit can be discarded.

Return Value

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

Diagnostics

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 on the transport endpoint referenced by *fd*.
[TFLOW]	O_NONBLOCK was set, but the flow control mechanism prevented the transport provider from accepting any data at this time.
[TBADDATA]	Illegal amount of data; zero octets are not supported.
[TLOOK]	An asynchronous event has occurred on the transport endpoint.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function. A protocol error cannot cause `t_sndudata()` to fail until a subsequent access of the transport endpoint.

See Also

fcntl(2), t_alloc(3xti), t_open(3xti), t_rcvudata(3xti), t_rcvuderr(3xti)

Name

t_sync – synchronize transport library

Syntax

#include <xti.h>

int t_sync(*fd*)
int *fd*;

Arguments

fd Identifies the local transport endpoint.

Description

For the transport endpoint specified by *fd*, `t_sync()` synchronizes the data structures managed by the transport library with information from the underlying transport provider. In doing so, `t_sync()` can convert an uninitialized file descriptor to an initialized transport endpoint, by updating and allocating the necessary library data structures. The file descriptor, which is assumed to have referenced a transport endpoint, has to be obtained by means of an `open()`, `dup()`, or be the result of a fork and `exec()`. The 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 activates so as not to violate the state of the transport endpoint. The `t_sync()` function 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 valid only 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.

Parameters	Before Call	After Call
fd	x	/

Return Value

Upon successful completion, `t_sync` returns the state of the transport endpoint. On failure, a value of –1 is returned, and **t_errno** is set to indicate the error. The state returned is one of the following:

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)

Diagnostics

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.

See Also

dup(2), exec(2), fork(2), open(2)

Name

t_unbind – disable a transport endpoint.

Syntax

#include <xti.h>

int t_unbind(*fd*)
int *fd*;

Arguments

fd Identifies the transport endpoint that the `t_unbind()` function disables.

Description

The `t_unbind()` function disables the transport endpoint specified by *fd* that was previously bound by `t_bind()`. On completion of this call, no futher data or events destined for this transport endpoint are accepted by the transport provider.

Parameters	Before Call	After Call
fd	x	/

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.

Diagnostics

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.

[TLOOK] An asynchronous event has occurred on the transport endpoint.

[TSYSERR] A system error has occurred during execution of this function.

See Also

t_bind(3xti)

Yellow Pages Routines (3yp)

Insert tabbed divider here.
Then discard this sheet.

Name

intro – introduction to Yellow Pages (YP) library functions

Description

This section describes those functions that are in the Yellow Pages library.

getnetgrent (3yp)

Name

getnetgrent, setnetgrent, endnetgrent, innetgr – get network group entry

Syntax

innetgr(netgroup, machine, user, domain)
char *netgroup, *machine, *user, *domain;

setnetgrent(netgroup)
char *netgroup

endnetgrent()

getnetgrent(machinep, userp, domainp)
char **machinep, **userp, **domainp;

Description

The `innetgr` routine accesses the `netgroup` file and checks to see if the specified input parameters match an entry in the file. The routine returns 1 if it matches an entry, or 0 if it does not. Any of the three strings; **machine, user,** or **domain** can be NULL, which signifies any string in that position is valid.

The `getnetgrent` routine returns the next member of a network group. After the call, `machinep` will contain a pointer to a string containing the name of the machine part of the network group member, and similarly for **userp** and **domainp.** If **machinep, userp** or **domainp** is returned as a NULL pointer, it signifies any string is valid. The `getnetgrent` routine allocates space for the name by using the `malloc` routine. This space is released when an `endnetgrent` call is made. The `getnetgrent` routine returns 1 if it succeeds in obtaining another member of the network group, or 0 if it reaches the end of the group.

The `setnetgrent` routine establishes the network group from which `getnetgrent` will obtain members, and also restarts calls to `getnetgrent` from the beginning of the list. If the previous `setnetgrent` call was to a different network group, an `endnetgrent` call is implied.

The `endnetgrent` routine releases the space allocated during the `getnetgrent` calls.

Files

/etc/netgroup
/etc/yp/*domain*/netgroup
/etc/yp/*domain*/netgroup.byuser
/etc/yp/*domain*/netgroup.byhost

Name

yp_get_default_domain, yp_bind, yp_unbind, yp_match, yp_first, yp_next, yp_all, yp_order, yp_master, yperr_string, ypprot_err – Yellow Pages client package

Syntax

#include <rpcsvc/ypclnt.h>

yp_get_default_domain(*outdomain*)
char ***outdomain*;

yp_bind(*indomain*)
char **indomain*;

void yp_unbind(*indomain*)
char **indomain*;

yp_match(*indomain, inmap, inkey, inkeylen, outval, outvallen*)
char **indomain*;
char **inmap*;
char **inkey*;
int *inkeylen*;
char ***outval*;
int **outvallen*;

yp_first(*indomain, inmap, outkey, outkeylen, outval, outvallen*)
char **indomain*;
char **inmap*;
char ***outkey*;
int **outkeylen*;
char ***outval*;
int **outvallen*;

yp_next(*indomain, inmap, inkey, inkeylen, outkey, outkeylen,*
char **indomain*;
char **inmap*;
char **inkey*;
int *inkeylen*;
char ***outkey*;
int **outkeylen*;
char ***outval*;
int **outvallen*;

yp_all(*indomain, inmap, incallback*)
char **indomain*;
char **inmap*;
struct ypall_callback *incallback*;

yp_order(*indomain, inmap, outorder*)
char **indomain*;
char **inmap*;
int **outorder*;

yp_master(*indomain, inmap, outname*)
char **indomain*;
char **inmap*;

char ***outname*;

char *yperr_string(*incode***)**
int *incode*;

ypprot_err(*incode***)**
unsigned int *incode*;

Description

This package of functions provides an interface to the Yellow Pages (YP) data base lookup service. The package can be loaded from the standard library, `/lib/libc.a`. Refer to `ypfiles`(5yp) and `ypserv`(8yp) for an overview of the Yellow Pages, including the definitions of **map** and **domain,** and for a description of the servers, data bases, and commands that constitute the YP application.

All input parameters names begin with **in.** Output parameters begin with **out.** Output parameters of type **char **** should be addresses of uninitialized character pointers. The YP client package allocates memory using `malloc`(3). This memory can be freed if the user code has no continuing need for it. For each **outkey** and **outval,** two extra bytes of memory are allocated at the end that contain NEWLINE and NULL, respectively, but these two bytes are not reflected in **outkeylen** or **outvallen.** The **indomain** and **inmap** strings must be non-null and null-terminated. String parameters that are accompanied by a count parameter cannot be null, but can point to null strings, with the count parameter indicating this. Counted strings need not be null-terminated.

All functions of type **int** return 0 if they succeed, or a failure code (YPERR_ xxxx) if they do not succeed. Failure codes are described under **Diagnostics.**

The YP lookup calls require a map name and a domain name. It is assumed that the client process knows the name of the map of interest. Client processes fetch the node's default domain by calling `yp_get_default_domain`, and use the returned **outdomain** as the **indomain** parameter to successive YP calls.

To use YP services, the client process must be bound to a YP server that serves the appropriate domain. The binding is accomplished with `yp_bind`. Binding need not be done explicitly by user code; it is done automatically whenever a YP lookup function is called. The `yp_bind` function can be called directly for processes that make use of a backup strategy in cases when YP services are not available.

Each binding allocates one client process socket descriptor; each bound domain requires one socket descriptor. Multiple requests to the same domain use that same descriptor. The `yp_unbind` function is available at the client interface for processes that explicitly manage their socket descriptors while accessing multiple domains. The call to `yp_unbind` makes the domain unbound, and frees all per-process and per-node resources used to bind it.

If an RPC failure results upon use of a binding, that domain will be unbound automatically. At that point, the ypclnt layer will retry forever or until the operation succeeds. This action occurs provided that `ypbind` is running, and either the client process cannot bind a server for the proper domain, or RPC requests to the server fail.

The `ypbind` **–s** option allows the system administrator to lock `ypbind` to a particular domain and set of servers. Up to four servers can be specified. An example of the **–s** option follows:

```
/etc/ypbind -s domain,server1[,server2,server3,server4]
```

The ypclnt layer will return control to the user code, either with an error code, or with a success code and any results under certain circumstances. For example, control will be returned to the user code when an error is not RPC-related and also when the `ypbind` function is not running. An additional situation that will cause the return of control is when a bound ypserv process returns any answer (success or failure).

The `yp_match` function returns the value associated with a passed key. This key must be exact; no pattern matching is available.

The `yp_first` function returns the first key-value pair from the named map in the named domain.

The `yp_next` function returns the next key-value pair in a named map. The **inkey** parameter should be the **outkey** returned from an initial call to `yp_first` (to get the second key-value pair) or the one returned from the nth call to `yp_next` (to get the nth + second key-value pair).

The concept of first and of next is particular to the structure of the YP map being processed; there is no relation in retrieval order to either the lexical order within any original (non-YP) data base, or to any obvious numerical sorting order on the keys, values, or key-value pairs. The only ordering guarantee made is that if the `yp_first` function is called on a particular map, and then the `yp_next` function is repeatedly called on the same map at the same server until the call fails with a reason of YPERR_NOMORE, every entry in the data base will be seen exactly once. Further, if the same sequence of operations is performed on the same map at the same server, the entries will be seen in the same order.

Under conditions of heavy server load or server failure, it is possible for the domain to become unbound, then bound once again (perhaps to a different server) while a client is running. This can cause a break in one of the enumeration 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. Enumerating all entries in a map is accomplished with the `yp_all` function.

The `yp_all` function 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 take place as a single RPC request and response. The `yp_all` function can be used like any other YP procedure, to identify the map in the normal manner, and to supply the name of a function that will be called to process each key-value pair within the map. Returns from the call to `yp_all` occur only when the transaction is completed (successfully or unsuccessfully), or when the `foreach` function decides that it does not want to see 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

```
foreach(instatus, inkey, inkeylen, inval, invallen, indata);
int instatus;
char *inkey;
int inkeylen;
char *inval;
int invallen;
char *indata;
```

The **instatus** parameter will hold 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 YP protocol error code to a ypclnt layer error code.)

The key and value parameters are somewhat different than defined in the syntax section above. First, the memory pointed to by the **inkey** and **inval** parameters is private to the yp_all function and is overwritten with the arrival of each new key-value pair. It is the responsibility of the foreach function to do something useful with the contents of that memory, but it does not own the memory itself. Key and value objects presented to the foreach function look exactly as they do in the server's map — if they were not newline-terminated or null-terminated in the map, they will not be here either.

The **indata** parameter is the contents of the **incallback->data** element passed to yp_all. The **data** element of the callback structure may be used to share state information between the foreach function and the mainline code. Its use is optional, and no part of the YP client package inspects its contents.

The foreach function returns a Boolean value. It should return zero to indicate that it wants to be called again for further received key-value pairs, or nonzero to stop the flow of key-value pairs. If foreach returns a nonzero value, it is not called again; the functional value of yp_all is then 0.

The yp_order function returns the order number for a map.

The yp_master function returns the machine name of the master YP server for a map.

The yperr_string function returns a pointer to an error message string that is null-terminated but contains no period or new line.

The ypprot_err function takes a YP protocol error code as input and returns a ypclnt layer error code, which may be used in turn as an input to yperr_string.

Diagnostics

All integer functions return 0 if the requested operation is successful, or one of the following errors if the operation fails.

```
#define YPERR_BADARGS   1  /* args to function are bad */
#define YPERR_RPC       2  /* RPC failure - domain has been unbound */
#define YPERR_DOMAIN    3  /* can't bind to server on this domain */
#define YPERR_MAP       4  /* no such map in server's domain */
#define YPERR_KEY       5  /* no such key in map */
#define YPERR_YPERR     6  /* internal yp server or client error */
#define YPERR_RESRC     7  /* resource allocation failure */
#define YPERR_NOMORE    8  /* no more records in map database */
#define YPERR_PMAP      9  /* can't communicate with portmapper */
```

```
#define YPERR_YPBIND     10 /* can't communicate with ypbind */
#define YPERR_YPSERV     11 /* can't communicate with ypserv */
#define YPERR_NODOM      12 /* local domain name not set */
```

Files

/usr/include/rpcsvc/ypclnt.h
/usr/include/rpcsvc/yp_prot.h

See Also

ypfiles(5yp), ypserv(8yp)

Name

yppasswd – update user password in yellow pages password map.

Syntax

#include <rpcsvc/yppasswd.h>

yppasswd(*oldpass, newpw*)
char **oldpass*;
struct passwd **newpw*;

Description

The `yppasswd` routine uses Remote Procedure Call (RPC) and External Data Representation (XDR) routines to update a user password in a Yellow Pages password map. The RPC and XDR elements that are used are listed below under the RPC INFO heading.

If *oldpass* is indeed the old user password, this routine replaces the password entry with *newpw*. It returns 0 if successful.

RPC Information

```
program number:
        YPPASSWDPROG
xdr routines:
        xdr_ppasswd(xdrs, yp)
                XDR *xdrs;
                struct yppasswd *yp;
        xdr_yppasswd(xdrs, pw)
                XDR *xdrs;
                struct passwd *pw;
procs:
        YPPASSWDPROC_UPDATE
                Takes struct yppasswd as argument, returns integer.
                Same behavior as yppasswd() wrapper.
                Uses UNIX authentication.
versions:
        YPPASSWDVERS_ORIG
structures:
        struct yppasswd {
                char *oldpass;  /* old (unencrypted) password */
                struct passwd newpw;   /* new pw structure */
        };
```

See Also

yppasswd(1yp), yppasswdd(8yp)

Index

A

B

E

G

M

Q

R

S

T

U

V

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-baud modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital Subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal*	————	SSB Order Processing - WMO/E15 *or* Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

* For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

ULTRIX
Reference Pages Section 3: Library Routines
AA-LY16B-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? ______________________________

What do you like best about this manual? ______________________________

What do you like least about this manual? ______________________________

Please list errors you have found in this manual:

Page ______ Description ______________________________

Additional comments or suggestions to improve this manual:

____

What version of the software described by this manual are you using? __________

Name/Title ______________________ Dept. __________

Company ______________________ Date __________

Mailing Address ______________________________

______________________ Email ______________________ Phone __________

Do Not Tear – Fold Here and Tape

digital™

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3-2/Z04
110 SPIT BROOK ROAD
NASHUA NH 03062-9987

Do Not Tear – Fold Here

Cut
Along
Dotted
Line

Reader's Comments

ULTRIX
Reference Pages Section 3: Library Routines
AA-LY16B-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? ______________________________

What do you like best about this manual? ______________________________

What do you like least about this manual? ______________________________

Please list errors you have found in this manual:

Page	Description

Additional comments or suggestions to improve this manual:

____

What version of the software described by this manual are you using? ____________

Name/Title ______________________________ Dept. ____________

Company ______________________________ Date ____________

Mailing Address ______________________________

______________________ Email ______________________ Phone ____________

Do Not Tear – Fold Here and Tape

digital™

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3-2/Z04
110 SPIT BROOK ROAD
NASHUA NH 03062-9987

Do Not Tear – Fold Here

Cut Along Dotted Line