NAME

`fedisabletrap`, `feenabletrap`, `fegettrapflag`, `fesettrapflag`, `fetesttrap` – Manages trap flags

SYNOPSIS

```
#include <fenv.h>

void fedisabletrap(int traps);
void feenabletrap(int traps);
void fegettrapflag(fetrap_t *flagp, int traps);
void fesettrapflag(const fetrap_t *flagp, int traps);
int fetesttrap(int traps);
```

IMPLEMENTATION

CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

Cray Research extensions to IEEE Std 754-1985

DESCRIPTION

These functions provide access to the trap flags. Each trap flag indicates whether a trap will be taken if the corresponding exception is raised. These traps are not *exact* traps, as specified in the IEEE standard. When an exception causes a trap to be taken, the floating-point exception signal (`SIGFPE`, see `signal.h`(3C)) is raised.

The `int` input argument for the functions represents a subset of floating-point exception traps, and it can be constructed by bitwise ORs of the trap macros (for example, `FE_TRAP_OVERFLOW | FE_TRAP_INEXACT`). The behavior of these functions is undefined for other argument values.

The `fedisabletrap` function disables the traps represented by its argument. The argument *traps* represents traps as a bitwise OR of trap macros.

The `feenabletrap` function enables the traps represented by its argument. The parameter *traps* represents traps as a bitwise OR of trap macros.

The `fegettrapflag` function stores the representation of the trap flags indicated by the parameter *traps* through the pointer parameter *flagp*.

The `fesettrapflag` function enables or disables the set of floating-point exception traps indicated by the argument *traps*, according to the representation in the object pointed to by *flagp*. The value of `*`*flagp* must have been set by a previous call to `fegettrapflag`; if it has not been, the effect on the indicated traps is undefined.

The `fetesttrap` function determines which of a specified subset of the floating-point exception traps are currently enabled. The *traps* argument specifies as a bitwise OR the traps to be queried.

RETURN VALUES

The `fetesttrap` function returns the bitwise OR of the trap macros corresponding to the currently enabled traps included in the *traps* argument.

SEE ALSO

Migrating to the CRAY T90 Series IEEE Floating Point, Cray Research publication SN–2194

NAME

`fegetenv`, `feholdexcept`, `fesetenv`, `feupdateenv` – Manages the entire floating-point environment

SYNOPSIS

```
#include <fenv.h>

void fegetenv(fenv_t *envp);
int feholdexcept(fenv_t *envp);
void fesetenv(const fenv_t *envp);
void feupdateenv(const fenv_t *envp);
```

IMPLEMENTATION

CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

ANSI/IEEE Std 754-1985
X3/TR-17:199x

DESCRIPTION

These functions manage the floating-point environment (that is, the status flags and control modes) as one entity.

The `fegetenv` function stores the current floating-point environment in the object pointed to by *envp*.

The `feholdexcept` function saves the current environment in the object pointed to by *envp*, clears the exception flags, and disables all floating-point exception traps. (`feholdexcept` does not disable all floating-point exception traps on CRAY T3E systems.) It can be used in conjunction with the `feupdateenv` function to write functions that hide spurious exceptions from their callers.

The `fesetenv` function establishes the floating-point environment represented by the object pointed to by *envp*. The argument *envp* must point to an object set by a call to `fegetenv`, or equal the macro `FE_DFL_ENV`. The `fesetenv` function just installs the state of the exception flags represented by its argument; it does not raise these exceptions.

The `feupdateenv` function saves the current exceptions in its automatic storage, installs the environment represented by *envp*, and raises the saved exceptions.

RETURN VALUES

The `feholdexcept` function returns a nonzero value only if all traps were successfully disabled.

EXAMPLES

The following example hides spurious underflow exceptions:

```
#include <fenv.h>
double f(double x)
{
        double result;
        fenv_t save_env;
        feholdexcept(&save_env);
        /*compute result*/
        if (/*test spurious underflow*/) feclearexcept(FE_UNDERFLOW);
        feupdateenv(&save_env);
        return result;
}
```

SEE ALSO

Migrating to the CRAY T90 Series IEEE Floating Point, Cray Research publication SN–2194

NAME

fegetround, fesetround – Manage the rounding direction modes

SYNOPSIS

```
#include <fenv.h>
```

```
int  fegetround(void);
int  fesetround(int round);
```

IMPLEMENTATION

CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

ANSI/IEEE Std 754-1985
X3/TR-17:199x

DESCRIPTION

The fegetround function gets the current rounding direction.

The fesetround function establishes the rounding direction represented by its argument *round*. If the argument does not match a rounding direction macro, the rounding direction is not changed.

EXAMPLES

The following example saves, sets, and restores the rounding direction. If setting the rounding direction fails, it reports an error and aborts.

```
#include <fenv.h>
#include <assert.h>
int save_round;
int setround_ok;
save_round = fegetround();
setround_ok = fesetround(FE_UPWARD);
assert(setround_ok);
/*...*/
fesetround(save_round);
```

RETURN VALUES

The fegetround function returns the value of the rounding direction macro that represents the current rounding direction.

The `fesetround` function returns a nonzero value if the argument matches a rounding direction macro (that is, if the requested rounding direction can be established).

SEE ALSO

`fenv.h`(3C) for the macros that can be used as arguments

Migrating to the CRAY T90 Series IEEE Floating Point, Cray Research publication SN–2194

NAME

fenv.h – Library header for the IEEE floating-point environment

IMPLEMENTATION

CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

ANSI/IEEE Std 754-1985
X3/TR-17:199x
The fetrap_t type and the trap macros and functions are Cray Research extensions to ANSI/IEEE Std 754-1985.

DESCRIPTION

The header fenv.h declares types, macros, and functions that provide access to the IEEE floating-point environment. The *IEEE floating-point environment* refers collectively to any floating-point status flags and control modes supported by an IEEE-compliant Cray Research system. A *floating-point status flag* is a system variable signifying some occurrence in the floating-point arithmetic. A *floating-point control mode* is a system variable that affects floating-point arithmetic.

The following conventions are followed by the Cray Research implementation:

- A function call does not alter its caller's modes, clear its caller's flags, or depend on the state of its caller's flags unless the function is so documented.
- A function call has default modes, unless either its documentation promises otherwise or the function is known not to use floating-point values.
- A function call has the potential for raising floating-point exceptions, unless either its documentation promises otherwise or the function is known not to use floating-point values.

Because these conventions are followed, programmers can safely assume that default modes are in effect and ignore them if they wish. Programmers must also accept any consequences, including a usually modest performance overhead, associated with explicitly accessing the IEEE floating-point environment.

TYPES

The fenv.h header file defines the following types:

Type	**Description**
fenv_t	Represents the entire floating-point environment.
fexcept_t	Represents the floating-point exception flags collectively, including any status associated with the flags.
fetrap_t	Represents the floating-point exception trap flags collectively.

MACROS

Each of the following macros represents one of the floating-point exception flags. They expand to `int` constant expressions whose values are distinct powers of 2. These macros are used as arguments to the exception functions described in the `feclearexcept`(3C) man page.

Macro	Description
`FE_INEXACT`	Represents the inexact exception flag
`FE_DIVBYZERO`	Represents the divide by zero exception flag
`FE_UNDERFLOW`	Represents the underflow exception flag
`FE_OVERFLOW`	Represents the overflow exception flag
`FE_INVALID`	Represents the invalid operation exception flag
`FE_EXCEPTINPUT`	Represents the exceptional input exception flag (This macro is valid only on CRAY T90 systems with IEEE arithmetic.)

The `FE_ALL_EXCEPT` macro is the bitwise OR of all exception macros.

Each of the following macros represent one of the trap flags. The defined macros expand to `int` constant expressions, the values of which are distinct powers of 2. These macros are used as arguments to the trap flag functions described in the `fedisabletrap`(3C) man page.

Macro	Description
`FE_TRAP_INVALID`	Represents the invalid operation trap flag
`FE_TRAP_DIVBYZERO`	Represents the divide-by-zero trap flag
`FE_TRAP_OVERFLOW`	Represents the overflow trap flag
`FE_TRAP_UNDERFLOW`	Represents the underflow trap flag
`FE_TRAP_INEXACT`	Represents the inexact trap flag
`FE_ALL_TRAPS`	Represents all of the trap flags

Each of the following macros represents a rounding direction. They expand to `int` constant expressions, the values of which are distinct nonnegative values. These macros are used as arguments to the rounding functions described in the `fegetround`(3C) man page.

Macro	Description
`FE_TONEAREST`	Round toward nearest
`FE_UPWARD`	Round toward positive infinity
`FE_DOWNWARD`	Round toward negative infinity
`FE_TOWARDZERO`	Round toward zero

The following macro represents the default floating-point environment, the one installed at program startup, and has type pointer to `fenv_t`. It can be used as an argument to `fenv.h` functions that manage the floating-point environment.

Macro	**Description**
`FE_DFL_ENV`	Represents the default floating-point environment

On the CRAY T90 series with IEEE floating-point hardware, the default rounding mode and trap modes can be specified at program startup by using the `cpu`(8) command.

FUNCTIONS

The following functions are described on separate man pages:

`feclearexcept`(3C)
`fegetexcept`, see `feclearexcept`(3C)
`feraiseexcept`, see `feclearexcept`(3C)
`fesetexcept`, see `feclearexcept`(3C)
`fetestexcept`, see `feclearexcept`(3C)
`fegetround`(3C)
`fesetround`, see `fegetround`(3C)
`fegetenv`(3C)
`feholdexcept`, see `fegetenv`(3C)
`fesetenv`, see `fegetenv`(3C)
`feupdateenv`, see `fegetenv`(3C)
`fedisabletrap`(3C)
`feenabletrap`, see `fedisabletrap`(3C)
`fegettrapflag`, see `fedisabletrap`(3C)
`fesettrapflag`, see `fedisabletrap`(3C)
`fetesttrap`, see `fedisabletrap`(3C)

SEE ALSO

`feclearexcept`(3C), `fegetround`(3C), `fegetenv`(3C), `fedisabletrap`(3C)

`cpu`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

Migrating to the CRAY T90 Series IEEE Floating Point, Cray Research publication SN–2194

NAME

`ferror`, `feof`, `clearerr` – Returns indication of stream status

SYNOPSIS

```
#include <stdio.h>
int ferror (FILE *stream);
int feof (FILE *stream);
void clearerr (FILE *stream);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The `ferror` function tests the error indicator for *stream*.

The `feof` function tests the end-of-file (EOF) indicator for *stream*.

The `clearerr` function clears the error indicator and EOF indicator on the specified stream.

NOTES

In extended mode, these functions are implemented as macros; they cannot be declared or redeclared. The macro versions of these functions are not multitask protected. To obtain versions that are multitask protected, compile your code by using `-D_MULTIP_` and link by using `/lib/libcm.a`.

RETURN VALUES

The `ferror` function returns nonzero when an I/O error has previously occurred reading from or writing to the specified stream; otherwise, it returns 0.

The `feof` function returns nonzero when EOF has previously been detected while reading the named input stream; otherwise, it returns 0.

SEE ALSO

`fopen`(3C)

`open`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

fgetpos, fsetpos – Stores or sets the value of the file position indicator

SYNOPSIS

```
#include <stdio.h>
int fgetpos (FILE *stream, fpos_t *pos);
int fsetpos (FILE *stream, const fpos_t *pos);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The fgetpos function stores the current value of the file position indicator for the stream to which *stream* points in the object to which *pos* points. The value stored contains unspecified information that the fsetpos function uses for repositioning the stream to its position at the time of the call to fgetpos.

The fsetpos function sets the file position indicator for the stream to which *stream* points, according to the value of the object to which *pos* points; *pos* is a value obtained from an earlier call to fgetpos on the same stream.

A successful call to the fsetpos function clears the end-of-file indicator (EOF) for *stream* and undoes any effects of ungetc(3C) on the same stream. After an fsetpos call, the next operation on an update stream can be either input or output.

RETURN VALUES

If successful, these functions return 0; on failure, they return a nonzero value and store a positive value in errno.

SEE ALSO

errno.h(3C), ungetc(3C)

NAME

file – Introduction to file system and directory functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

These functions provide means for accessing basic system resources affecting file systems and directories.

ASSOCIATED HEADERS

The following header files are documented in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014:

<dirent.h>
<fcntl.h>
<fstab.h>
<utmp.h>

The following header files are documented in separate entries in this manual:

<stdio.h>
<sys/types.h>

ASSOCIATED FUNCTIONS

Function	**Description**
alphasort	Sorts alphabetically an array of pointers to directory entries (see scandir(3C))
closedir	Closes directory stream (see directory(3C)
endfsent	Gets file system descriptor file entry (see getfsent(3C))
endmntent	Gets file system descriptor file entry (see getmntent(3))
endutent	Accesses utmp file entry (see getut(3C))
flock(3C)	Applies or removes an advisory lock on an open file
ftw(3C)	Walks a file tree
getcwd(3C)	Gets path name of current directory
getfsent(3C)	Gets file system descriptor file entry
getfsfile	Gets file system descriptor file entry (see getfsent(3C))
getfsspec	Gets file system descriptor file entry (see getfsent(3C))
getfstype	Gets file system descriptor file entry (see getfsent(3C))
getmntent(3C)	Gets file system descriptor file entry
getutent	Accesses utmp file entry (see getut(3C))
getutid	Accesses utmp file entry (see getut(3C))
getutline	Accesses utmp file entry (see getut(3C))
getwd(3C)	Gets current directory path name

`hasmntopt`	Gets file system descriptor file entry (see `getmntent(3)`)
`lockf`(3C)	Provides record locking on files
`nftw`	Walks a file tree (see `ftw`(3C))
`opendir`	Opens directory and associates stream with it (see `directory`(3C))
`pututline`	Accesses `utmp` file entry (see `getut`(3C))
`readdir`	Returns a pointer to the next active directory entry (see `directory`(3C))
`readdir`	Returns a pointer to the next active directory entry (see `directory`(3C))
`readdir_r`	Returns a pointer to the next active directory entry (see `directory`(3C))
`rewinddir`	Resets position of directory stream to beginning of directory (see `directory`(3C))
`scandir`(3C)	Scans a directory
`seekdir`	Sets up next `readdir` operation (see `directory`(3C))
`setfsent`	Gets file system descriptor file entry (see `getfsent`(3C))
`setmntent`	Gets file system descriptor file entry (see `getmntent`(3C))
`setutent`	Accesses `utmp` file entry (see `getut`(3C))
`telldir`	Returns current location of directory stream (see `directory`(3C))
`utimes`(3C)	Sets file times
`utmpname`	Accesses `utmp` file entry (see `getut`(3C))

SEE ALSO

`message`(3C), `multic`(3C), `password`(3C), `terminal`(3C) (all introductory pages to other operating system service functions)

See the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014, for more complete descriptions of UNICOS header files.

NAME

`fileno` – Returns integer file descriptor associated with stream

SYNOPSIS

```
#include <stdio.h>
int fileno (FILE *stream);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX

DESCRIPTION

The `fileno` function returns the integer file descriptor associated with the named stream; see `open`(2).

NOTES

In extended mode, the `fileno` function is implemented as a macro; it cannot be declared or redeclared.

FORTRAN EXTENSION

You can also call the `fileno` function from Fortran programs, as follows:

```
INTEGER FILENO, stream
I = FILENO(stream)
```

SEE ALSO

`fopen`(3C)

`open`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

float.h – Library header for floating-point number limits

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

TYPES

None

MACROS

The header float.h defines macros that determine the characteristics of floating-point numbers, which, in turn, define the floating-point arithmetic available on Cray Research systems. Variables used in the explanations of the macros are as follows:

s	Sign (± 1)
b	Base or radix of exponent representation (an integer > 1)
e	Exponent (an integer between a minimum $e_{\min}$ and a maximum $e_{\max}$)
p	Precision (the number of base-b digits in the significand)
f_k	Nonnegative integers less than b (the significand)

A normalized floating-point number x ($f_1 > 0$ if $x \neq 0$) is defined by the following model:

$$x = s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k} \;,\quad e_{\min} \leq e \leq e_{\max}$$

The macros and definitions are shown in the following table:

Macro	Definition
FLT_RADIX	A constant expression suitable for use in #if preprocessing directives, which is the radix of exponent representation, b.
FLT_ROUNDS	The rounding modes for floating-point arithmetic, as follows: −1, indeterminate; 0, toward zero; 1, to nearest; 2, toward positive infinity; or 3, toward negative infinity. All other values for FLT_ROUNDS are used for implementation-defined rounding behavior.

Macro	Definition
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG	float, double, or long double value that is the number of base-FLT_RADIX digits in the floating-point significand, *p*.
FLT_DIG DBL_DIG LDBL_DIG	float, double, or long double value that is the number of decimal digits, *q*, such that any floating-point number with *q* decimal digits can be rounded into a floating-point number with *p* radix *b* digits and back again without change to the *q* decimal digits. $\lfloor (p - 1) \times \log_{10} b \rfloor + \begin{cases} 1 & \text{if } b \text{ is a power of 10} \\ 0 & \text{otherwise} \end{cases}$
FLT_MIN_EXP DBL_MIN_EXP LDBL_MIN_EXP	float, double, or long double value that is the minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized floating-point number, $e_{\min}$
FLT_MIN_10_EXP DBL_MIN_10_EXP LDBL_MIN_10_EXP	float, double, or long double value that is the minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers. $\lfloor \log_{10} b^{e_{\min}-1} \rceil$
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	float, double, or long double value that is the maximum integer such that FLT_RADIX raised to that power minus 1 is a representable finite floating-point number, $e_{\max}$
FLT_MAX_10_EXP DBL_MAX_10_EXP LDBL_MAX_10_EXP	float, double, or long double value that is the maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers. $\lfloor \log_{10}((1 - b^{-p}) \times b^{e_{\max}}) \rfloor$
FLT_MAX DBL_MAX LDBL_MAX	float, double, or long double value that is the maximum representable finite floating-point number. $(1 - b^{-p}) \times b^{e_{\max}}$
FLT_EPSILON DBL_EPSILON LDBL_EPSILON	float, double, or long double value that is the difference between 1.0 and the least value greater than 1.0 that is representable in the given floating point type, b^{1-p}.
FLT_MIN DBL_MIN LDBL_MIN	float, double, or long double value that is the minimum normalized positive floating-point number. $b^{e_{\min}-1}$

The values in `float.h` are shown in the following table comparing the values for Cray format floating point and the values required by the ISO/ANSI and IEEE standards. Information in the left column applies to all Cray Research PVP machines, such as CRAY C90 series and CRAY Y-MP series systems. Information in the right column applies to Cray MPP systems and some CRAY T90 series systems. Some early CRAY T90 series systems use Cray format floating point. Where values for Cray MPP systems and CRAY T90 series differ, both values are shown.

Macro	CRI format values	IEEE format values (Cray MPP/CRAY T90)
`FLT_ROUNDS`	0	1
`FLT_RADIX`	2	2
`FLT_MANT_DIG`	47	24 / 53
`DBL_MANT_DIG`	47	53
`LDBL_MANT_DIG`	94	53 / 113
`FLT_DIG`	13	6 / 15
`DBL_DIG`	13	15
`LDBL_DIG`	27	15 / 33
`FLT_MIN_EXP`	– 8189	– 125 / – 1021
`DBL_MIN_EXP`	– 8189	– 1021
`LDBL_MIN_EXP`	– 8189	– 1021 / – 16381
`FLT_MIN_10_EXP`	– 2465	– 37 / – 307
`DBL_MIN_10_EXP`	– 2465	– 307
`LDBL_MIN_10_EXP`	– 2465	– 307 / – 4931
`FLT_MAX_EXP`	8190	128 / 1024
`DBL_MAX_EXP`	8190	1024
`LDBL_MAX_EXP`	8190	1024 / 16384
`FLT_MAX_10_EXP`	2465	38 / 308
`DBL_MAX_10_EXP`	2465	308
`LDBL_MAX_10_EXP`	2465	308 / 4932
`FLT_MAX`	2.726870339048517e2465	3.4028234663852886e+38F / 1.7976931348623158e308
`DBL_MAX`	2.726870339048517e2465	1.7976931348623158e+308
`LDBL_MAX`	2.726870339048517e2465L	1.7976931348623158e308 / 1.189731495357231765085759326628007016E+4932L
`FLT_EPSILON`	7.10542735760100e– 15	1.1920928955078125e– 7F / 2.2204460492503131e– 16
`DBL_EPSILON`	7.10542735760100e– 15	2.2204460492503131e– 16
`LDBL_EPSILON`	2.5243548967072377773175314089e– 29L	2.2204460492503131e– 16 / 1.925929944387235853055977942584927319E-34L
`FLT_MIN`	1/2.726870339048517e2465	1.1754943508222875e– 38F / 2.2250738585072014e– 308

		IEEE format values
Macro	CRI format values	(Cray MPP/CRAY T90)
DBL_MIN	1/2.726870339048517e2465	2.2250738585072014e–308
LDBL_MIN	1/2.726870339048517e2465L	2.2250738585072014e–308 / 3.362103143112093506262677817321752603E–4932L

Because of the dynamic characteristics of Cray format (not IEEE format) floating-point operations, some of the floating-point values defined by this header are not exactly the same as the limiting values of the actual hardware. (This does not apply to to Cray systems that comply with the IEEE floating-point standard, such as CRAY T90 systems with floating-point hardware and Cray MPP systems systems.) The values are those closest to the actual hardware, meeting the following criteria:

- The reciprocal of the maximum floating-point value does not cause underflow.
- The reciprocal of the minimum positive floating-point value does not cause overflow.
- Correct results are obtained for the tests described by the paper "A Test of a Computer's Floating-Point Arithmetic Unit," Computing Science Technical Report No. 89, Bell Laboratories, by N. L. Schryer, February 4, 1981.

In other words, the model must allow all arithmetic operations on operands that are within the range of minimum to maximum, inclusive, and for which results are, mathematically, also within the range; and further, the results must be accurate to within the ability of the hardware to represent the mathematical value.

These values for floating-point maximum and minimum values define a range that is slightly smaller than the value that can be represented by Cray hardware, but use of numbers outside this range may not yield predictable results. When the defined values are used, the following relationships or expressions can be handled without the occurrence of floating-point exceptions:

max/max	min/min	1/max
1/min	max = 1/min	min = 1/max
(max/2)*2		

This model is defined by 47 bits in the mantissa, 94 bits for long double, a minimum biased exponent of 020003 (octal), and a maximum biased exponent of 057776 (octal).

Decimal representation of all numbers in this range are guaranteed to be accurate to 13 significant digits, 27 significant digits for long double. That is, any decimal string within the range of minimum to maximum with this or fewer significant digits are guaranteed to be convertible from decimal string to internal representation and back to the same decimal string.

FUNCTION DECLARATIONS

None

SEE ALSO

fenv.h(3C), fp.h(3C), limits.h(3C), values.h(3C)

NAME

flock – Applies or removes an advisory lock on an open file

SYNOPSIS

```
#include <sys/file.h>
int flock (int fd, int operation);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The flock function is a compatibility function, provided to aid in the porting of code from other systems. It is built on top of record locking. Shared locks are implemented as read record locks, and exclusive locks are implemented as write record locks. See fcntl(2) for more information.

The flock function applies or removes an advisory lock on the file associated with the file descriptor *fd*. A lock is applied by specifying an *operation* parameter that is the inclusive OR of LOCK_SH or LOCK_EX and, possibly, LOCK_NB. To unlock an existing lock, *operation* should be LOCK_UN.

The header file sys/file.h contains the following declarations:

```
#define  LOCK_SH  1  /* shared lock */
#define  LOCK_EX  2  /* exclusive lock */
#define  LOCK_NB  4  /* don’t block when locking */
#define  LOCK_UN  8  /* unlock */
```

Advisory locks allow cooperating processes to perform consistent operations on files, but do not guarantee consistency (that is, processes can still access files without using advisory locks, which could possibly result in inconsistencies).

The locking mechanism allows two types of locks: *shared* locks and *exclusive* locks. At any time, multiple shared locks can be applied to a file, but at no time are multiple exclusive, or both shared and exclusive, locks allowed simultaneously on a file.

A shared lock can be upgraded to an exclusive lock, and vice versa, simply by specifying the appropriate lock type; this results in the previous lock being released and the new lock applied (possibly after other processes have gained and released the lock).

Requesting a lock on an object that is already locked usually causes the caller to be blocked until the lock can be acquired. If `LOCK_NB` is included in *operation*, this does not happen; instead, if the object is already locked, the call fails, and the error `EWOULDBLOCK` is returned.

NOTES

Locks are on files, not file descriptors. That is, file descriptors duplicated through `dup`(2) or `fork`(2) do not result in multiple instances of a lock, but rather multiple references to a single lock. If a process holding a lock on a file forks and the child explicitly unlocks the file, the parent loses its lock.

Processes that have been blocked while awaiting a lock can be awakened by signals.

RETURN VALUES

If the operation was successful, 0 is returned; otherwise, – 1 is returned and an error code is stored in `errno.`

See `fcntl`(2) for a list of possible error values.

FILES

`/usr/include/sys/file.h`

SEE ALSO

`close` (2), `dup` (2), `execve` (2), `fcntl` (2), `fork` (2), `open` (2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR– 2012

NAME

flockfile, ftrylockfile, funlockfile – Locks file stream

SYNOPSIS

```
#include <stdio.h>
void flockfile(FILE *file);
int ftrylockfile(FILE *file);
void funlockfile(FILE *file);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

PThreads

DESCRIPTION

The flockfile, ftrylockfile, and funlockfile functions provide for explicit application-level locking of stdio (FILE *) objects. A thread can use these functions to delineate a sequence of I/O statements that are to be executed as a unit.

The flockfile function is used by a thread to acquire ownership of a (FILE *) object.

The ftrylockfile function is used by a thread to acquire ownership of a (FILE *) object if the object is available; ftrylockfile is a nonblocking version of flockfile.

The funlockfile junction is used to relinquish the ownership granted to the thread. The caller must be the current owner of the (FILE *) object.

Logically, each (FILE *) object is associated with a lock count. This count is implicitly initialized to zero when the (FILE *) object is created. The (FILE *) object is unlocked when the count is zero. When the count is positive, a single thread owns the (FILE *) object.

When the flockfile function is called, if the count is zero or if the count is positive and the caller owns the (FILE *) object, the count is incremented. Otherwise, the calling thread is suspended, waiting for the count to return to zero. Each call to funlockfile decrements the count. This allows matching calls to flockfile (or successful calls to ftrylockfile) and funlockfile to be nested.

You must link your program with lib/libcm.a to use these functions.

When linked with libcm.a, the basic I/O functions that reference (FILE *) objects behave as if they use flockfile and funlockfile internally to obtain ownership of these (FILE *) objects.

RETURN VALUES

There are no return values for `flockfile` and `funlockfile`. The function `ftrylockfile` returns 0 for success and a nonzero value to indicate that the lock cannot be acquired.

NAME

`floor`, `floorf`, `floorl`, `ceil`, `ceilf`, `ceill`, `fmod`, `fmodf`, `fmodl`, `fabs`, `fabsf`, `fabsl`, `cabs` – Provides math function for floor, ceiling, remainder, and absolute value

SYNOPSIS

```
#include <math.h>
#include <complex.h> (for function cabs only)
double floor (double x);
float floorf (float x);
long double floorl (long double x);
double ceil (double x);
float ceilf (float x);
long double ceill (long double x);
double fmod (double x, double y);
float fmodf (float x, float y);
long double fmodl (long double x, long double y);
double fabs (double x);
float fabsf (float x);
long double fabsl (long double x);
double cabs (double complex x);
```

IMPLEMENTATION

All Cray Research systems (`floor`, `ceil`, `fmod`, `fabs`, `cabs` only)
Cray MPP systems (`floorf`, `ceilf`, `fmodf`, `fabsf` only)
Cray PVP systems (`floorl`, `ceill`, `fmodl`, `fabsl` only)

STANDARDS

ISO/ANSI (functions `floor`, `fabs`, `ceil` only)
CRI extension (all others)

DESCRIPTION

The `floor`, `floorf`, and `floorl` functions compute, respectively, the largest integral value not greater than *x* for `double`, `float`, and `long double` numbers.

The `ceil`, `ceilf`, and `ceill` functions compute, respectively, the smallest integral value not less than *x* for `double`, `float`, and `long double` numbers.

The `fmod`, `fmodf`, and `fmodl` functions compute, respectively, the floating-point remainder of *x/y* for `double`, `float`, and `long double` numbers.

The `fabs`, `fabsf`, `fabsl`, and `cabs` functions compute, respectively, the absolute value of a floating-point number *x* for `double`, `float`, `long double`, and `double complex` numbers. For `cabs`, this is computed as follows:

$$\mathit{sqrtcreal}\,(x\,)^2 + \mathit{cimag}\,(x\,)^2$$

Vectorization is inhibited for loops containing calls to all functions except `fabs`, `fabsf`, `fabsl`, and `cabs`. In strict conformance mode, vectorization is inhibited for loops containing calls to functions `fabs`, `fabsf`, `fabsl`, and `cabs`.

RETURN VALUES

The `floor`, `floorf`, and `floorl` functions return the largest integral value not greater than *x*, expressed as a `double`, `float`, or `long double`, respectively.

The `ceil`, `ceilf`, and `ceill` functions return the smallest integral value not less than *x*, expressed as a `double`, `float`, or `long double`, respectively.

The `fmod`, `fmodf`, and `fmodl` functions return the value $x - i * y$, for some integer *i* such that, if *y* is not 0, the result has the same sign as *x* and a magnitude less than the magnitude of *y*. If *y* is 0, these functions return 0. Under some implementations, this may cause a domain error to occur.

The `fabs`, `fabsf`, `fabsl`, and `cabs` functions return the absolute value of *x*, expressed as a `double`, `float`, `long double`, or `double complex` number, respectively.

NAME

`FLOWMARK` – Allows timing of a section of code

SYNOPSIS

```
#include <stdlib.h>
int FLOWMARK (long *name);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

CRI extension

DESCRIPTION

The `FLOWMARK` function subdivides a large function into several smaller logical functions for Flowtrace. *name* points to a null-terminated ASCII character string that starts on a word boundary; it identifies the mark within the function. *name* should be 8 characters or less and should not contain blanks or nonprinting characters. The library can handle any pointer, no matter what its definition. To terminate a mark, call `FLOWMARK` again with a 0 argument.

For more information about tracing a program, see the description of the `-F` command-line option in the *Cray Standard C Reference Manual*, Cray Research publication SR–2074, and the *Guide to Parallel Vector Applications*, Cray Research publication SG–2182.

EXAMPLES

In Standard C, a mark is used as follows:

```
#include <stdlib.h>

main()
{
        long     zero = 0;
        .
        .
        .
        FLOWMARK("phantom");
        for (i = 0; i<10; i++) {
                .
                .
                .
                /* loop performs desired work */
                .
                .
                .
        }
        FLOWMARK(&zero);
}
```

FORTRAN EXTENSIONS

In Fortran, the calls would be:

```
      .
      .
      .
      CALL FLOWMARK('newname')
      .
      .
      .
      CALL FLOWMARK(0)
      .
      .
      .
```

SEE ALSO

`flowtrace`(7), `performance`(7), `perftrace`(7) (all online only)

Cray Standard C Reference Manual, Cray Research publication SR–2074

Guide to Parallel Vector Applications, Cray Research publication SG–2182

NAME

`fnmatch` – Matches file name or path name

SYNOPSIS

```
#include <fnmatch.h>
int fnmatch (const char *pattern, const char *string, int flags);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX

DESCRIPTION

The `fnmatch` function matches patterns according to the rules used by the shell. It checks the string specified by the *string* argument to determine whether it matches the pattern specified by the *pattern* argument.

The *flags* argument modifies the interpretation of *pattern* and *string*. The value of *flags* is the bitwise inclusive OR of any of the following constants, which are defined in the include file `<fnmatch.h>`:

Constant	**Description**
`FNM_NOESCAPE`	Causes the backslash character (`\`) to be treated as an ordinary character negating any special meaning for the character. Normally, every occurrence of a backslash followed by a character in *pattern* is replaced by that character.
`FNM_PATHNAME`	Causes the slash character (`/`) to be treated as an ordinary character. Slash characters in *string* must be explicitly matched by slashes in *pattern*.
`FNM_PERIOD`	Causes a leading period (`.`) in *string* to be matched as it would be matched by the shell, where the definition of *leading* is determined by the value of `FNM_PATHNAME`. If `FNM_PATHNAME` is set, a period is "leading" if it is the first character in *string* or if it immediately follows a slash; if `FNM_PATHNAME` is not set, a period is "leading" only if it is the first character of *string*.

NOTES

The pattern `*` matches the empty string, even if `FNM_PATHNAME` is specified.

RETURN VALUES

The `fnmatch` function returns zero if *string* matches the pattern specified by *pattern*; otherwise, it returns the value `FNM_NOMATCH`.

SEE ALSO

`glob`(3C), `wordexp`(3C), `regexp(3)`

`sh`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

`fopen`, `freopen`, `fdopen` – Opens a stream

SYNOPSIS

```
#include <stdio.h>
FILE *fopen (const char *file, const char *type);
FILE *freopen (const char *file, const char *type, FILE *stream);
FILE *fdopen (int fildes, const char *type);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (except `fdopen`)
POSIX (`fdopen` only)

DESCRIPTION

The `fopen` function opens the file specified by *file* and associates a stream (*stream*) with it. It returns a pointer to the `FILE` structure associated with the *stream*.

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

The *type* argument is a character string that has one of the following values:

Value	**Description**
`"r"`	Opens text file for reading.
`"w"`	Creates text file for writing or truncates it to 0 length.
`"a"`	Appends; opens or creates text file for writing at end-of-file.
`"r+"`	Opens text file for update (reading and writing).
`"w+"`	Creates text file for update or truncates it to 0 length.
`"a+"`	Appends; opens or creates text file for update; writing at end-of-file.
`"rb"`	Opens binary file for reading.
`"wb"`	Creates binary file for writing or truncates it to 0 length.
`"ab"`	Appends; opens or creates binary file for writing at end-of-file.
`"r+b"` or `"rb+"`	Opens binary file for update (reading and writing).
`"w+b"` or `"wb+"`	Creates binary file for update or truncates it to 0 length.
`"a+b"` or `"ab+"`	Appends; opens or creates binary file for updating, and writing at end-of-file.

You must specify the two-letter *type* values in the order shown; that is, `"rb"` is valid, but `"br"` is not. The same is true for the *type* arguments that contain the plus signs; `"r+b"` is valid, but `"b+r"` is not. (Under UNICOS, binary and text streams are implemented identically; specifications of `"b"` are ignored.)

The `freopen` function opens the file whose name is the string to which *file* points and associates the stream to which *stream* points with it. The *type* argument is used just as in the `fopen` function.

The `freopen` function first tries to close any file that is associated with the specified stream. Failure to close the file successfully is ignored. The error and end-of-file indicators for the stream are cleared. Typically the `freopen` function is used to attach the preopened *streams* associated with `stdin`, `stdout`, and `stderr` to other files.

The `fdopen` function associates *stream* with a file descriptor obtained from `open`(2), `dup`(2), `creat`(2), or `pipe`(2), which opens files but does not return pointers to a `FILE` structure *stream* that are necessary input for many of the C library functions. The *type* of *stream* must agree with the mode of the open file.

If the file does not exist or cannot be read, opening a file with read mode (`r` as the first character in the *type* argument) fails.

Opening a file for writing causes the file to be truncated to a length of 0 if it exists; otherwise, the file is created.

When a file is opened for append, you cannot overwrite information already in the file. You can use `fseek`(3C) 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 the file pointer is 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 is intermixed in the file.

When a file is opened for update, both input and output can be done on the resulting *stream*. Output may not be directly followed by input, however, without an intervening `fflush`(3C), `fsetpos`(3C), `fseek`(3C), or `rewind`(3C) function/operation, and input cannot be directly followed by output without an intervening `fsetpos`(3C), `fseek`(3C), or `rewind`(3C) function/operation, or an input operation that encounters end-of-file.

When opened, a stream is fully buffered if, and only if, it can be determined not to refer to an interactive device. The error and end-of-file indicators for the stream are cleared.

By default, `fopen` and the associated functions that perform I/O on streams are not multitask-protected. To obtain a multitask-protected version, link with `lib/libm.a`.

RETURN VALUES

Function `fopen` returns a pointer to an object controlling *stream*. Function `freopen` returns the value of *stream*. Both `fopen` and `freopen` return a null pointer on failure.

FORTRAN EXTENSIONS

You can also call the `fopen` function from Fortran programs, as follows:

```
CHARACTER file *m, type *n
INTEGER*8 FOPEN, stream
stream = FOPEN(file, type)
```

You can also call the `freopen` function from Fortran programs, as follows:

```
CHARACTER file *m, type *n
INTEGER*8 FREOPEN, stream
stream = FREOPEN(file, type, stream)
```

You can also call the `fdopen` function from Fortran programs, as follows:

```
CHARACTER type *n
INTEGER*8 FDOPEN, stream, fildes
stream = FDOPEN(fildes, type)
```

On systems other than Cray MPP systems and the CRAY T90 series, for any of these functions, arguments *file* or *type* may be integer variables. These integer variables must be packed 8 characters per word and terminated with a null (0) byte.

SEE ALSO

`fclose`(3C), `fseek`(3C)

`creat`(2), `dup`(2), `open`(2), `pipe`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

fortran.h – Library header for interlanguage communication functions

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The header file fortran.h defines the functions for interlanguage communication.

Type

The header file fortran.h defines the following type:

Type	Standards	Description
_fcd	CRI	C representation of the Fortran character descriptor

Function Declarations

Functions declared in the header file fortran.h are as follows:

_btol _cptofcd _fcdtocp _fcdlen _isfcd _ltob

SEE ALSO

Cray Standard C Reference Manual, Cray Research publication SR–2074

Interlanguage Programming Conventions, Cray Research publication SN–3009

NAME

`fpclassify`, `isfinite`, `isnormal`, `isnan` – Identifies its argument as NaN, infinite, normal, subnormal, or zero

SYNOPSIS

```
#include <fp.h>
```

```
int fpclassify (floating-type x);
int isfinite (floating-type x);
int isnormal (floating-type x);
int isnan (floating-type x);
```

IMPLEMENTATION

Cray MPP systems (type `double` versions only)
CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

ANSI/IEEE Std 754-1985
X3/TR-17:199x

DESCRIPTION

The `fpclassify` macro returns the value of the number classification macro appropriate to the value of its argument.

The `isfinite` macro determines if the argument*x* has a finite value. Values that are zero, subnormal, or normal are considered finite, but values that are infinite or NaN are not.

The `isnormal` macro determines if its argument value is normal, meaning it is neither zero, subnormal, infinite, nor NaN.

The `isnan` macro determines whether its argument value is a NaN. On CRAY T3D systems, *floating-point* must be `double`. On CRAY T90 systems with IEEE hardware, the *floating-type x* argument indicates a parameter of any floating type. If the argument is not a floating type, the behavior is undefined.

If any of the macro definitions are suppressed in order to access an actual function, or if a program defines an external identifier with the name of one of the macros, the behavior is undefined.

RETURN VALUES

The `isfinite` macro returns a nonzero value if its argument has a finite value.

The `isnormal` macro returns a nonzero value if its argument has a normal value.

The `isnan` macro returns a nonzero value if its argument has a NaN value.

The number classification macros returned by `fpclassify` are as follows:

`FP_NAN` The argument is a NaN.

`FP_INFINITE` The argument is either positive or negative infinity.

`FP_NORMAL` The argument is a normal floating-point number (neither zero, subnormal, infinite, nor NaN).

`FP_SUBNORMAL` The argument is a denormalized floating-point number.

`FP_ZERO` The argument is positive or negative zero.

NOTES

A version of `isnan` that is implemented as a function is also available on all Cray Research systems. That version is defined in the `<math.h>` header file. It offers the advantage of being compliant with the XPG4 standard but accepts only `double` arguments.

SEE ALSO

`isnan`(3C), for the `<math.h>` version of `isnan`
Migrating to the CRAY T90 Series IEEE Floating Point, Cray Research publication SN–2194

NAME

`fp.h` – Library header for IEEE floating-point functions and macros

IMPLEMENTATION

Cray MPP systems (see individual man pages for restrictions)
CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

ANSI/IEEE Std 754-1985
X3/TR-17:199x

DESCRIPTION

The header `fp.h` declares macros and functions to support general IEEE floating-point programming.

MACROS

The following macros defined in the `fp.h` header file return predefined values:

Macro **Description**

`HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL`

On CRAY T90 systems with IEEE arithmetic, expands to positive infinity for their corresponding data types. They expand to `double`, `float`, and `long double` expressions, respectively. `HUGE_VAL` is also defined, with the same value, in the `math.h` header file. On CRAY T3D systems, `HUGE_VAL` expands to a positive expression that is the largest representable `double` value. `HUGE_VALF` and `HUGE_VALL` are not defined.

`INFINITY` Expands to a floating expression of type `double`, representing positive infinity. The `INFINITY` macro is not suitable for static and aggregate initialization.

`NAN` Expands to a floating-point expression of type `double`, representing a quiet NaN. The `NAN` macro is not suitable for static and aggregate initialization.

`FP_NAN`, `FP_INFINITE`, `FP_NORMAL`, `FP_SUBNORMAL`, and `FP_ZERO`

Represent the mutually exclusive kinds of floating-point values. They expand to `int` constant expressions with distinct values. They are for number classification. See the `fpclassify`(3C) man page for individual descriptions of these macros.

DECIMAL_DIG Expands to an int constant expression representing the number of decimal digits supported by conversion between decimal and all internal floating-point formats. (DECIMAL_DIG is intended to give an appropriate number of digits to carry in canonical decimal representations.) Conversion from any floating-point value to decimal with DECIMAL_DIG digits and back is the identity function. DECIMAL_DIG is distinct from DBL_DIG, which is defined in terms of conversion from decimal to double and back.

The following function-like macros are described on their own man pages:

fpclassify(3C)
isfinite, see fpclassify(3C)
isnan, see fpclassify(3C)
isnormal, see fpclassify(3C)
isgreater(3C)
isgreaterequal, see isgreater(3C)
isless, see isgreater(3C)
islessequal, see isgreater(3C)
islessgreater, see isgreater(3C)
isunordered, see isgreater(3C)
signbit(3C)

FUNCTION DECLARATIONS

The following functions are described on their own man pages:

copysign(3C)
copysignf, see copysign(3C)
copysignl, see copysign(3C)
logb(3C)
logbf, see logb(3C)
logbl, see logb(3C)
nextafter(3C)
nextafterf, see nextafter(3C)
nextafterl, see nextafter(3C)
remainder(3C)
remainderf, see remainder(3C)
remainderl, see remainder(3C)
rint(3C)
rintf, see rint(3C)
rintl, see rint(3C)
rinttol(3C)
scalb(3C)
scalbf, see scalb(3C)
scalbl, see scalb(3C)

SEE ALSO

`copysign`(3C), `fpclassify`(3C), `isgreater`(3C), `logb`(3C), `nextafter`(3C), `remainder`(3C), `rint`(3C), `rinttol`(3C), `scalb`(3C), `signbit`(3C)

Migrating to the CRAY T90 Series IEEE Floating Point, Cray Research publication SN–2194

NAME

`fread`, `fwrite` – Reads or writes input or output

SYNOPSIS

```
#include <stdio.h>
extern size_t fread (void *ptr, size_t size, size_t nitems, FILE *stream);
extern size_t fwrite (const void *ptr, size_t size, size_t nitems, FILE
*stream);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The `fread` function copies, into an array to which *ptr* points, *nitems* items of data from the specified input *stream*, in which an item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. (`size_t` is defined in `stdio.h` to be `unsigned`.) The `fread` function stops appending bytes if an end-of-file or error condition is encountered while reading *stream*, or if *nitems* items have been read. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate. If a partial element is read, its value is indeterminate. The `fread` function does not change the contents of *stream*. The file position indicator for the stream (if defined) is advanced by the number of characters successfully read.

The `fwrite` function appends at most *nitems* items of data of size *size* from the array to *ptr* points to the specified output *stream*. The `fwrite` function stops appending when it has appended *nitems* items of data or if an error condition is encountered on *stream*. The file position indicator for the stream (if defined) is advanced by the number of characters successfully written. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate. The `fwrite` function does not change the contents of the array to which *ptr* points.

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

RETURN VALUES

Function `fread` returns the number of items successfully read. If *size* or *nitems* is 0 or nonpositive, 0 is returned, and the contents of the array and the state of the stream remain unchanged.

Function `fwrite` returns the number of items written. If *size* or *nitems* is 0, no characters are read or written, and 0 is returned.

FORTRAN EXTENSIONS

You also can call the `fread` and `fwrite` functions from Fortran programs. The following shows their use as Fortran functions:

```
INTEGER*8 ptr, size, nitems, stream, FREAD, I
I = FREAD(ptr, size, nitems, stream)

INTEGER*8 ptr, size, nitems, stream, FWRITE, I
I = FWRITE(ptr, size, nitems, stream)
```

The following shows the use of `fwrite` or `fread` as Fortran subroutines.

```
INTEGER*8 ptr, size, nitems, stream
CALL FREAD(ptr, size, nitems, stream)

INTEGER*8 ptr, size, nitems, stream
CALL FWRITE(ptr, size, nitems, stream)
```

In this case, the library function’s return value is unavailable.

The Fortran program cannot specify both the subroutine call and the function reference to `fwrite` or `fread` from the same procedure.

On all systems except Cray MPP systems and CRAY T90 series systems, argument *ptr* may be a Fortran character variable.

On all Cray Research systems, `FREADC` and `FWRITEC` are also available as Fortran interfaces to `fread` and `fwrite`. These two functions require that *ptr* be a Fortran character variable.

```
CHARACTER *n ptr
INTEGER*8 size, nitems, stream, FREADC, I
I = FREADC(ptr, size, nitems, stream)

INTEGER*8 size, nitems, stream, FWRITEC, I
I = FWRITEC(ptr, size, nitems, stream)
```

SEE ALSO

`fopen`(3C), `getc`(3C), `gets`(3C), `printf`(3C), `putc`(3C), `puts`(3C), `scanf`(3C)

`read`(2), `write`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

frexp, frexpf, frexpl, ldexp, ldexpf, ldexpl, modf, modff, modfl – Manipulates parts of floating-point numbers

SYNOPSIS

```
#include <math.h>
double frexp (double value, int *ptr);
float frexpf (float value, int *ptr);
long double frexpl (long double value, int *ptr);
double ldexp (double value, int exp);
float ldexpf (float value, int exp);
long double ldexpl (long double value, int exp);
double modf (double value, double *iptr);
float modff (float value, float *iptr);
long double modfl (long double value, long double *iptr);
```

IMPLEMENTATION

All Cray Research systems (frexp, ldexp, modf only)
Cray MPP systems (frexpf, ldexpf, modff only)
Cray PVP systems (frexpl, ldexpl, modfl only)

STANDARDS

ISO/ANSI (frexp, ldexp, modf only)
CRI extension (all others)

DESCRIPTION

The frexp, frexpf, and frexpl functions break a floating-point number into a normalized fraction and an integral power of 2. They store the integer in the int object to which *ptr* points.

The ldexp, ldexpf, and ldexpl functions multiply a floating-point number by an integral power of 2. A range error may occur. If the result underflows, the functions return zero. If the result overflows, the function ldexp returns HUGE_VAL (defined in both the math.h and fp.h header files), and ldexpl returns LDBL_MAX (defined in the float.h header file), with the same sign as the correct value of the function. Both functions set errno to ERANGE on overflow.

The `modf`, `modff`, and `modfl` functions break the argument *value* into integral and fractional parts, each of which has the same sign as the argument and store the integral part as a `double`, `float`, or `long double`, respectively, in the object to which *iptr* points.

Vectorization is inhibited for loops containing calls to any of these functions.

RETURN VALUES

The `frexp`, `frexpf`, and `frexpl` functions return the value *x*, such that *x* is a `double`, `float`, or `long double` with a magnitude in the interval [1/2, 1] or 0, and *value* equals *x* multiplied by 2 raised to the power **ptr*. If *value* is 0, both parts of the result are 0.

The `ldexp`, `ldexpf`, and `ldexpl` functions return the value of *value* multiplied by 2 raised to the power *exp*.

The `modf`, `modff`, and `modfl` functions return the signed fractional part of the *value* argument.

On Cray MPP systems and CRAY T90 systems with IEEE floating-point arithmetic:

- `frexp(`*NaN*`, *iptr)` returns NaN.
- `frexpl(`*NaN*`, *iptr)` returns NaN.
- `frexp(`*x*`, *iptr)`, where *x* is +/- infinity, returns *x*.
- `frexpl(`*x*`, *iptr)`, where *x* is +/- infinity, returns *x*.
- `ldexp(`*NaN*`)` returns NaN.
- `ldexpl(`*NaN*`)` returns NaN.
- `modf(`*NaN*`, *iptr)` returns NaN and `errno` is set to `EDOM`.
- `modfl(`*NaN*`, *iptr)` returns NaN and `errno` is set to `EDOM`.

NAME

`fseek`, `rewind`, `ftell` – Repositions a file pointer in a stream

SYNOPSIS

```
#include <stdio.h>
int fseek (FILE *stream, long int offset, int whence);
void rewind (FILE *stream);
long int ftell (FILE *stream);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The `fseek` function sets the file position indicator for *stream*. The new position is at the signed distance *offset* bytes from the beginning of the file, from the current position of the file pointer, or from the end of the file, according to the value of *whence*, as follows:

Name	**Description**
`SEEK_SET`	Sets position equal to *offset* bytes
`SEEK_CUR`	Sets position to current location of file position indicator plus *offset*
`SEEK_END`	Sets position to EOF plus *offset*

If the stream will be used with wide character input/output functions, *offset* must either be 0 or a value returned by an earlier call to the `ftell` function on the same stream and *whence* must be `SEEK_SET`.

A successful call to the `fseek` function clears the end-of-file (EOF) indicator for the stream and undoes any effects of the `ungetc`(3C) function on the same stream. After `fseek`, the next operation on a file opened for update can be either input or output.

The `rewind` function sets the file position indicator for the stream to which *stream* points to the beginning of the file. Calling `rewind(`*stream*`)` is equivalent to calling `fseek(`*stream*`, 0L, SEEK_SET)`, except that no value is returned, and the error indicator for the stream is cleared.

The `ftell` function obtains the current value of the file position indicator for the stream to which *stream* points. For a text or binary stream, the value is the number of characters from the beginning of the file.

Functions `fseek` and `rewind` undo any effects of `ungetc`(3C) on the same stream.

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

NOTES

The functions fseek and ftell, as well as fsetpos(3C) and fgetpos(3C), exist because on some computer systems, the size of int is too small to contain the position for large files. Thus, for functions fsetpos(3C) or fgetpos(3C), positions may be contained in structures.

A file position of 0 is ambiguous. It can mean "at beginning of file" or "at beginning of file after calling the ungetc(3C) function once."

RETURN VALUES

The fseek function returns nonzero for improper seeks, or seeks that could not be honored; otherwise, it returns 0. An improper seek can be, for example, an fseek done on a file that has not been opened using fopen(3C); in particular, you cannot use fseek on a terminal, or on a file opened using popen(3C).

If successful, the ftell function returns the current value of the file position indicator for the stream. On failure, the ftell function returns -1L and stores a positive value in errno: EBADF if *stream* does not point to an opened stream; otherwise, ftell can return the same errnos as lseek(2).

FORTRAN EXTENSIONS

You can also call the fseek and ftell functions from Fortran programs, as follows:

```
INTEGER*8 FSEEK, whence, stream, offset, I
I = FSEEK(stream, offset, whence)

INTEGER*8 FTELL, stream, I
I = FTELL(stream)
```

SEE ALSO

fgetpos(3C), fopen(3C), getc(3C), popen(3C), ungetc(3C)

unget(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

lseek(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

`truncate`, `ftruncate` – Truncates a file to a specified length

SYNOPSIS

```
#include <unistd.h>

int truncate (const char *path, off_t length);

int ftruncate (int fd, off_t length);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The `truncate` function causes the file to which *path* refers (or for `ftruncate`, the file to which *fd* refers) to be truncated to a maximum of *length* bytes. If the file was previously larger than this size, the extra data is lost. With `ftruncate`, the file must be open for writing.

These are compatibility functions, which are provided to aid in the porting of code from other systems. They are implemented through a combination of `open`(2), `lseek`(2), and `trunc`(2) system calls.

RETURN VALUES

If the call succeeds, a value of 0 is returned. If the call fails, –1 is returned, and `errno` is set to the error.

SEE ALSO

`lseek`(2), `open`(2), `trunc`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012, for the specific error values.

NAME

ftw, nftw – Walks a file tree

SYNOPSIS

```
#include <ftw.h>
```

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

```
int nftw (const char *path, int (*fn) (const char *, const struct stat
*, int, struct FTW *), int depth, int flags);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4 (ftw only)
AT&T extension (nftw only)

DESCRIPTION

The ftw and nftw functions recursively descend the directory hierarchy rooted in *path*. For each object in the hierarchy, ftw or nftw calls *fn*, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a stat structure (see stat(2)) containing information about the object, and an integer. Possible values of the integer, defined in the header file ftw.h are as follows:

Flag Value	**Description**
FTW_F	The object is a file.
FTW_D	The object is a directory.
FTW_DP	(nftw only) The object is a directory and subdirectories have been visited.
FTW_SL	(nftw only) The object is a symbolic link.
FTW_DNR	The object is a directory that cannot be read.
FTW_NS	Object for which stat could not be successfully executed.

If the integer is FTW_DNR, descendants of that directory are not processed. If the integer is FTW_NS, the stat structure contains nothing meaningful. For example, a file in a directory with read permission but without execute (search) permission would cause FTW_NS to be passed to *fn*. For nftw, stat failure for any reason is considered an error and nftw will return –1.

The nftw function works similarly to ftw except that it takes an additional argument *flags*. The possible values for *flag* are specified in the header file ftw.h and are as follows:

Flag Value	**Description**
FTW_PHYS	A physical walk; it does not follow symbolic links.
FTW_MOUNT	The walk does not cross a mount point.

`FTW_DEPTH` All subdirectories are visited before the directory itself.
`FTW_CHDIR` The walk changes to each directory before reading it.

Also, the `nftw` function calls *fn* with four (instead of three) arguments at each file and directory. This fourth argument is a pointer to a `struct FTW` that contains the following members:

```
int base;
int level;
```

The value of `base` is the offset to the base of the path name of the object; this path name is passed as the first argument to *fn*. The value of `level` indicates depth relative to the root of the walk, where the root level has a value of zero.

The `ftw` function 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, such as an I/O error, is detected within the function. If the tree is exhausted, `ftw` or `nftw` returns 0. If *fn* returns a nonzero value, `ftw` or `nftw` stops its tree traversal and returns whatever value was returned by *fn*. If either function detects an error, it returns −1 and sets the error type in `errno`.

The `ftw` and `nftw` functions use 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* argument must not be greater than the number of file descriptors currently available for use; however, these functions run more quickly if *depth* is at least as large as the number of levels in the tree.

NOTES

Because `ftw` and `nftw` are recursive, it is possible for them to terminate with a memory fault when applied to very deep file structures.

Functions `ftw`/`nftw` use `malloc` to allocate dynamic storage during its operation. If they are forcibly terminated (for example, if `longjmp` is executed by *fn* or an interrupt function) they do not have a chance to free that storage, so it remains 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.

SEE ALSO

`malloc`(3C)

`stat`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR−2012

NAME

getc, getchar, fgetc, getc_unlocked, getchar_unlocked, getw, fgetwc, getwchar, getwc – Gets a character or word from a stream

SYNOPSIS

```
#include <stdio.h>
int fgetc (FILE *stream);
int getc (FILE *stream);
int getchar (void);
int getc_unlocked (FILE *stream);
int getchar_unlocked (void);
int getw (FILE *stream);

#include <wchar.h>
wint_t fgetwc(FILE *stream);
wint_t getwc(FILE *stream);
wint_t getwchar(void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (fgetc, getc, and getchar only)
POSIX (getc_unlocked and getchar_unlocked only)
XPG4 (getw, fgetwc, getwc, and getwchar only)

DESCRIPTION

The fgetc function obtains the next character (if present) as an unsigned char converted to an int, from the input stream to which *stream* points, and it advances the associated file position indicator for the *stream* (if defined).

The getc function is equivalent to fgetc, except that it is implemented as a macro, and it may evaluate *stream* more than once; therefore, the argument should never be an expression with side effects. In particular, getc(*f++) does not work sensibly; use fgetc instead.

The getchar function is equivalent to getc with the argument of stdin.

The getc_unlocked and getchar_unlocked functions provide functionality equivalent to the getc and getchar functions, respectively. However, these interfaces are not guaranteed to be locked with respect to concurrent standard I/O operations in a multitasked application. Thus, you should use these functions only within a scope protected by the flockfile(3C) or ftrylockfile(3C) functions.

The getw function returns the next word (that is, type int) from the specified input stream. Function getw increments the associated file position indicator, if defined, to point to the next word. The size of a word is the size of a type int (64 bits). The getw function assumes no special alignment in the file.

The fgetwc function obtains the next character (if present) from the input stream to which *stream* points, converts that to the corresponding wide-character code, and advances the associated file position indicator for the *stream* (if defined). The st_ctime and st_mtime fields of the file are marked for update between the successful execution of fputwc(3C) and the next successful completion of a call to fflush(3C) or fclose(3C) on the same stream or a call to exit(3C) or abort(3C).

The getwc function is equivalent to fgetwc, except that, if it is implemented as a macro, it may evaluate *stream* more than once; therefore, the argument should never be an expression with side effects. Because it can be implemented as a macro, getwc can incorrectly treat a *stream* argument with side effects. In particular, getwc(*f++) might not work as expected. Therefore, this function is not recommended; you should use the fgetwc function instead.

The getwchar function is equivalent to getwc(*stdin*). If the value returned by getwchar is stored into a variable of type wchar_t and then compared against the wint_t macro WEOF, the comparison may never succeed.

NOTES

The fgetc function runs more slowly than getc, but it takes less space per invocation, and its name can be passed as an argument to a function.

The macro version of the getc function is not multitask protected. To obtain a multitask protected version, compile your code by using -D_MULTIP_ and link by using /lib/libcm.a.

WARNINGS

If the integer value returned by getc, getchar, or fgetc is stored into a character variable and then compared against the integer constant EOF, the comparison may never succeed, because sign-extension of a character when it is widened to an integer is not done by the Cray Standard C implementation.

If you use getw to read a file whose size is not a multiple of a word, the last partial word will not be read; getw will return EOF after the last full word is read from the file.

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

RETURN VALUES

The `fgetc`, `getc`, and `getchar` functions return the next character from the input stream to which *stream* points (or `stdin` for `getchar`). If the stream is at end-of-file (EOF), the EOF indicator for the stream is set and the functions return `EOF`. If a read error occurs, the error indicator for the stream is set and the functions return `EOF`.

The `getw` function returns the constant `EOF` at end-of-file or on an error. Because `EOF` is a valid integer, you should use `feof` or `ferror` to detect `getw` errors.

The return values for the `fgetwc`, `getwc`, and `getwchar` functions behave similarly to those returned for `fgetc`, `getc`, and `getchar`. They differ in that they return wide characters or `WEOF` as their values.

SEE ALSO

`fclose`(3C), `ferror`(3C), `fopen`(3C), `fread`(3C), `gets`(3C), `putc`(3C), `scanf`(3C)

NAME

`getconfval`, `getconfvals`, `freeconfval` – Gets configuration values

SYNOPSIS

```
#include <stdlib.h>
char *getconfval (char *product, char *field);
char **getconfvals (char *product, char *field);
void freeconfval (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

Function `getconfval` searches through the configuration file looking for the specified *product* and *field*, and returns a character pointer to the string defined in the configuration file. If no item is found for the requested program or field, a null pointer is returned. If more than one item is found for the requested program or field, the first item in the list is returned. To obtain the entire list, use function `getconfvals`.

Function `getconfvals` performs the same search as `getconfval`; however, it returns a pointer to a list of character pointers, which point to the configuration data. If no item is found for the requested program or field, the return value is null. A successful call returns a pointer to an array of character pointers associated with the desired *product* and *field*, which can be processed in a similar fashion as `argv` (for example, `getopt`(3C)).

The return strings can be interpreted as the calling program desires (for example, passing the resultant string into `atoi` (see `strtol`(3C)) as shown in example 3, following).

Function `freeconfval` frees up all memory allocated during previous calls to `getconfval` and/or `getconfvals`. This function should not be called unless all needed configuration information has been obtained. The `getconfval` and `getconfvals` functions buffer the configuration information internally to allow for faster access to same-product information, thereby reducing the number of disk requests.

Calling function `freeconfval` deallocates these buffers, which forces any subsequent `getconfval` call to reprocess and buffer the relevent portions of the configuration file. Function `freeconfval` also closes the `/etc/config/confval` file.

For more information about the structure of the configuration file, see `confval`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014.

EXAMPLES

For all of the following examples, assume that the configuration file contains the following information:

```
login.root_console:     "console" "weather"
login.debug:            "1"
gated.debug:            "0"
```

Example 1:

```
char *val;
val = getconfval("login", "root_console");
freeconfval();
```

The call in example 1 would result in:

```
      *val     = "console"
```

Example 2:

```
char *vals;
vals = getconfvals("login", "root_console");
```

The call in example 2 would result in:

```
      *(vals)     = "console"
      *(vals + 1) = "weather"
```

Example 3:

```
#include <stdlib.h>

int debug;
debug = atoi(getconfval("gated", "debug"));
```

The call in example 3 would result in:

```
      debug    = 0
```

Example 4:

```
char **vals;
vals = getconfvals("gated", "debug");
```

The call in example 4 would result in:

```
      *vals = "0"
```

Example 5:

```
char *val;
val = getconfval("junk", "junk");
```

The call in example 5 would result in:

```
        val      = NULL
```

Example 6:

```
char **vals;
vals = getconfvals("junk", "junk");
```

The call in example 6 would result in:

```
        vals     = NULL
```

CAUTIONS

If a program uses `getconfval` or `getconfvals`, and it `exec`s another image without an intervening call to `freeconfval`, the original file is not released, although the memory is released.

NOTES

The `getconfval` and `getconfvals` functions buffer information on a product basis. During the first call to either function, the configuration file is opened and all of the consecutive entries defined for the given product are buffered into memory. This improves the performance of any subsequent calls that attempt to reference information about the same product. If the program no longer needs any information from the configuration files, function `freeconfval` can be called to free up all memory allocated during previous `getconfval` and `getconfvals` calls.

Also, if the `confval` configuration file is modified between calls to these functions, the executing binary may not detect the changes due to the buffering scheme. For best results, the binary should be restarted (if possible).

RETURN VALUES

The `getconfval` call returns a character pointer to the first configuration value for the given program/field found in the configuration file. If no item is found, a null value is returned.

The `getconfvals` call returns a pointer to a list of character pointers that point to the value information for the specified product/field strings. If no item is found, a null value is returned.

FILES

`/etc/config/confval`	Contains configuration information

SEE ALSO

`atoi` (see `strtol`(3C)), `getopt`(3C)

`confval`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

getcwd – Gets path name of current directory

SYNOPSIS

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX

DESCRIPTION

The getcwd function copies the absolute path name of the current working directory to the character array pointed to by the argument *buf* and returns a pointer to the result. The *size* argument is the size in bytes of the character array pointed to by the *buf* argument. The value of *size* must be at least two greater than the length of the path name to be returned.

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

RETURN VALUES

The getcwd function returns null with errno set if *size* is not large enough, or if an error occurs in a lower-level function.

EXAMPLES

The following source code fragment prints to standard output the name of the current directory:

```
#include <stdlib.h>    /* exit */
#include <stdio.h>     /* perror, printf */
#inlcude <unistd.h>    /* getcwd */
...

char *cwd;

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

SEE ALSO

errno.h(3C), getwd(3C), malloc(3C)

NAME

`getdomainname`, `setdomainname` – Gets or sets name of current domain

SYNOPSIS

```
#include <unistd.h>
int getdomainname (char *name, int namelen);
int setdomainname (char *name, int namelen);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The purpose of domains is to enable two distinct networks that may have common host names to merge. Each network would be distinguished by a different domain name. Currently, only the network information service (formerly known as yellow pages) makes use of domains.

The `getdomainname` function returns the name of the domain for the current processor, as previously set by `setdomainname`. The parameter *namelen* specifies the size of the *name* array. The returned name is null-terminated unless insufficient space is provided.

The `setdomainname` function sets the domain of the host machine to *name*, which has the length *namelen*. This call is restricted to the super user and is normally used only when the system is bootstrapped.

NOTES

Domain names are limited to 64 characters.

RETURN VALUES

If the call succeeds, a value of 0 is returned. If the call fails, a value of –1 is returned, and an error code is placed in the global variable `errno`.

MESSAGES

This function can set `errno` to one of the following values (defined in header `errno.h`) on error:

Error Code	**Description**
`EFAULT`	The *name* parameter gave an invalid address.
`EPERM`	The caller was not the super user. This error applies only to function `setdomainname`.

SEE ALSO

errno.h(3C)

NAME

`getdtablesize` – Gets file descriptor table size

SYNOPSIS

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

```
int getdtablesize (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

Each process has a fixed-size file descriptor table. The entries in the file descriptor table are numbered with small integers, starting at 0. The `getdtablesize` function returns the size of this table (that is, the number of file descriptors).

The following example is a compatibility routine, which is provided to aid in the porting of code from other systems, and is implemented by the following code:

```
#include <unistd.h>

int getdtablesize(void);
{
        return(sysconf(_SC_OPEN_MAX));
}
```

SEE ALSO

`close`(2), `dup`(2), `open`(2), `select`(2), `sysconf`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

getenv – Returns the value for the specified environment name

SYNOPSIS

```
#include <stdlib.h>
char *getenv (const char *name);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The getenv function searches the environment list (see sh(1)) for a string of the form *name* = *value*; and it returns a pointer to *value* in the current environment if such a string is present. If such a string is not present, getenv returns a null pointer.

RETURN VALUES

The getenv function returns a pointer to a string associated with the matched list member. The string pointed to cannot be modified by the program, but may be overwritten by a subsequent call to the getenv function. If the specified *name* cannot be found, a null pointer is returned.

FORTRAN EXTENSIONS

On systems other than Cray MPP systems and the CRAY T90 series, the getenv function can be called from Fortran programs. It may be called as an integer function, as follows. The function PXFGETENV is available on all Cray systems and is a recommended alternative to GETENV.

```
CHARACTER name*m, value*n
INTEGER*8 GETENV , found
found = GETENV(name, value)
```

or

```
INTEGER*8 value(valuesz)
INTEGER*8 name
INTEGER*8 GETENV , found
found = GETENV(name, value, valuesz)
```

Function GETENV returns 1 if *name* was found in the environment, and 0 if it is not.

The getenv function can also be called from Fortran programs as a subroutine, as follows (as already stated, not on Cray MPP systems and CRAY T90 series):

```
CHARACTER name*m, value*n
CALL GETENV(name, value)
```

or

```
INTEGER*8 value(valuesz)
INTEGER*8 name
CALL GETENV(name, value, valuesz)
```

A status is not returned for the GETENV subroutine call.

Arguments *m* and *n* are integer constants specifying the length in characters of the character strings *name* and *value*, respectively.

NOTES

GETENV must be declared type integer to ensure proper testing of the return code.

SEE ALSO

putenv(3C), setenv(3C)

sh(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

getfsent, getfsspec, getfsfile, getfstype, setfsent, endfsent – Gets file system descriptor file entry

SYNOPSIS

```
#include <fstab.h>
struct fstab *getfsent (void);
struct fstab *getfsspec (char *spec);
struct fstab *getfsfile (char *file);
struct fstab *getfstype (char *type);
int setfsent (void);
int endfsent (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

These functions are included for compatibility with existing programs. They call the getmntent(3C) functions.

The getfsent, getfsspec, getfstype, and getfsfile functions each return a pointer to an object with the following structure, which contains the broken-out fields of a line in the file system description file, header file <fstab.h>, as follows:

```
struct fstab {
        char     *fs_spec;     /* block special device name */
        char     *fs_file;     /* file system path prefix */
        char     *fs_type;     /* file system type */
        int      fs_freq;      /* dump frequency, in days */
        int      fs_passno;    /* pass number on parallel check */
};
```

The getfsent function reads the next line of the file, opening the file if necessary.

The setfsent function opens and rewinds the file.

The endfsent function closes the file.

Functions getfsspec and getfsfile sequentially search from the beginning of the file until a matching special file name or file system file name is found, or until EOF is encountered. Function getfstype does likewise, matching on the file system type field.

NOTES

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

FILES

/etc/fstab

RETURN VALUES

All of these functions return a null pointer on EOF or error.

SEE ALSO

getmntent(3C)

fstab(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

getgrent, getgrgid, getgrgid_r, getgrnam, getgrnam_r, setgrent, endgrent, fgetgrent – Gets group file entry

SYNOPSIS

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

struct group *getgrent (void);

struct group *getgrgid (int gid);

int getgrgid_r (int gid, struct group *grp, char *buf, size_t bufsize, struct
group **result);

struct group *getgrnam (const char *name);

int getgrname_r (char *name, struct group *grp, char *buf, size_t bufsize, struct
group **result);

void setgrent (void);

void endgrent (void);

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX (getgrgid and getgrnam)
PThreads (getgrgid_r and getgrnam_r)
AT&T extension (endgrent, fgetgrent, getgrent, and setgrent)

DESCRIPTION

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

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

When first called, getgrent returns a pointer to the first group structure of the first line in the file; thereafter, it returns a pointer to the group structure of the next line in the file. Therefore, successive calls may be used to search the entire file. The `getgrgid` function searches from the beginning of the file until a numerical group ID matching *gid* is found and returns a pointer to the particular structure in which it was found. The `getgrnam` function searches from the beginning of the file until a group name matching *name* is found and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a null pointer.

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

The `fgetgrent` function returns a pointer to the next `group` structure in stream *f* that matches the format of `/etc/group`.

The functions whose names end with `_r` provide equivalent functionality but with an interface that is safe for multitasked applications. The primary difference is that, instead of returning a pointer to a structure, they place the results in the structure pointed to by the *grp* argument. In addition, they use the provided buffer *buf* of size *bufsize* to store auxilary data. The maximum size needed for this buffer can be determined with the `_SC_GETGR_R_SIZE_MAX` `sysconf` parameter. A `NULL` pointer is returned at the location pointed to by *result* on error or if the required entry is not found.

NOTES

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

WARNINGS

For groups with a large number of members, several lines in file `/etc/group` will be generated with the same group ID. Thus, to fully scan a particular group may require more than one `getgrent` call.

The preceding functions use header `stdio.h`, which causes them to increase the size of programs more than might otherwise be expected.

RETURN VALUES

For all interfaces other than `getgrgid_r` and `getgrnam_r`, a null pointer is returned on `EOF` or error. For `getgrgid_r` and `getgrnam_r`, 0 is returned on success. Otherwise an error number is returned:

`ERANGE` Insufficient storage was supplied via *buf* and *bufsize* to contain the data to be referenced by the resulting *struct group* structure.

FILES

`/etc/group`

SEE ALSO

`getlogin`(3C), `getpwent`(3C)

`group`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

gethostbyaddr, gethostbyname, gethostent, gethostlookup, sethostent, endhostent, sethostlookup – Gets a network host entry

SYNOPSIS

```
#include <sys/types.h>
#include <netdb.h>
#include <netinet/in.h>
struct hostent *gethostbyaddr (char *addr, int len, int type);
struct hostent *gethostbyname (char *name);
struct hostent *gethostent (void);
int gethostlookup (void);
int sethostlookup (int  lookup_type);
void sethostent (int stayopen);
void endhostent (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension (except gethostlookup and sethostlookup)
CRI extension (only gethostlookup and sethostlookup)

DESCRIPTION

The gethostbyaddr, gethostbyname, and gethostent functions each return a pointer to an object describing an Internet host entry. The information in the network host entry will be obtained either from the user's HOSTALIASES file, the /etc/host.aliases file, the network host database /etc/hosts (or its binary version, /etc/hosts.bin, if it exists), or from the domain name service provided by named(8), as determined by the gethostlookup function and the existance of an aliases file.

The following structure describes a network host entry:

```
    struct      hostent {
            char     *h_name;       /* official host name */
            char    **h_aliases;    /* alias list */
            int       h_addrtype;   /* address type */
            int       h_length;     /* length of address */
            char    **h_addr_list;  /* list of addresses */
    }
```

The members of this structure are as follows:

Member	**Description**
h_name	Official name of the host.
h_aliases	Zero-terminated array of alternative names for the host.
h_addrtype	Type of the address being returned; the only address type currently supported is AF_INET.
h_length	Length, in bytes, of the host address.
h_addr_list	List of network addresses for the host from the name server. Host addresses are in network byte order (bytes ordered from left to right). For backward compatibility, h_addr is the first entry in h_addr_list.

The gethostlookup function returns either HOSTLOOKUP_HOSTFILE or HOSTLOOKUP_NAMED (defined in netdb.h), depending on whether the network host information should be retrieved from the /etc/hosts database or the domain name service. The function checks for the existence of the environment variable HOSTLOOKUP. If the value of HOSTLOOKUP is either hosttable or named, gethostlookup returns the corresponding value. Otherwise, gethostlookup checks for the existence of the file /etc/hosts.usenamed. If this file exists, gethostlookup returns HOSTLOOKUP_NAMED; otherwise, it returns HOSTLOOKUP_HOSTFILE.

The sethostlookup function lets an application inform the gethostbyaddr, gethostbyname, and gethostlookup functions to use either the name server or /etc/hosts file. For an application, this would be equivalent to a user defining the environment variable HOSTLOOKUP.

If the input name contains no dot, and if the environment variable HOSTALIASES contains the name of an alias file, the alias file will be searched for an alias matching the input name before either the network host database (/etc/hosts) or the domain name service search happens. If the file /etc/hosts.aliases exists, that file will be searched next if no match is found in the HOSTALIASES file.

The aliases file should consist of lines made up of two columns separated by whitespace. The first column contains the hostname aliases, and the second column lists the hostname (or IP address) to be substituted for the alias listed in the first column.

When the domain name service is used for host lookup, the sethostent function instructs the service to use a virtual circuit for connections to name servers if the *stayopen* flag is not zero. When the host database is used for lookup, the sethostent function opens and rewinds either the /etc/hosts.bin file (if it exists) or the /etc/hosts file.

When the domain name service is used for host lookup, the endhostent function closes the virtual circuit connection to a name server, if one was used. When the host database is used for lookup, the endhostent function closes the /etc/hosts.bin or /etc/hosts file if the file is still open.

The gethostbyaddr function fetches information for the host with address *addr*. Address *addr* for the gethostbyaddr function is cast as a character pointer to a structure defined by in.h (struct in_addr). The gethostbyname function fetches information for the host with name (or alias) *name*. When host database lookup is used, the appropriate database (/etc/hosts.bin or /etc/hosts) is searched sequentially until the desired information is found or the last entry is reached. Both functions return host addresses in network byte order.

When using host database lookup, the gethostent function returns the next entry in the /etc/hosts.bin or /etc/hosts database, opening the file if necessary. When using the domain name service, it returns a null pointer.

NOTES

All information is contained in a static area that must be copied if it is to be saved. Only the Internet address and OSI are currently recognized by the UNICOS operating system.. The gethostbyaddr code checks only the first address in the list. The /etc/hosts.bin file is not automatically kept current with /etc/hosts. If the administrator forgets to run mkbinhost(8), users will get obsolete host information.

RETURN VALUES

For functions gethostbyaddr, gethostbyname, and gethostent, a null pointer (0) is returned at end-of-file or when an error occurs. When the null pointer is returned at end-of-file, this indicates that gethostbyaddr or gethostbyname did not find the specified name or address in the file.

FILES

/etc/hosts

/etc/hosts.bin

/etc/hosts.usenamed

/etc/hosts.aliases

/usr/include/netdb.h

/usr/include/netinet/in.h

SEE ALSO

gethostinfo(3C), resolver(3C)

hosts(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

mkbinhost(8), named(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

gethostinfo – Gets network host and service entry

SYNOPSIS

```
#include <netdb.h>
```

struct hostinfo *gethostinfo (char **host*, char **service*, int *family*,
int *type*, int *flags*);

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The gethostinfo function returns a pointer to an object describing a network host and service entry. Information on the host is obtained either from the network host database /etc/hosts (or its binary version, /etc/hosts.bin, if it exists), or from the domain name service provided by named(8), as determined by the gethostlookup(3C) function. Information on the service portion of the entry is obtained from the network service database /etc/services. These functions, unlike the gethost functions, have entry structures that can contain an entry for an OSI host.

The following structure describes a network host and service entry:

```
    struct  hostinfo {
            char     *h_name;               /* official host name */
            char    **h_aliases;            /* host alias list */
            struct hostserv **h_addr_serv;  /* list of services */
    };

  struct  hostserv {
          struct sockaddr *hs_addr;       /* address info */
          char   *hs_name;                /* official service name */
          char  **hs_aliases;             /* service alias list */
          int     hs_type;                /* socket type (for TCP only) */
  };
```

The members of these structures are as follows:

Member	**Description**
h_name	Official name of the host.
h_aliases	Zero-terminated array of alternative names for the host.
hs_addr_serv	List of network addresses and service information for the host.

`hs_addr`	Address information for the host. This field serves as a place holder to be overlayed with either a `struct sockaddr_in` or a `struct sockaddr_iso` structure. The `sockaddr_in` structure is used for Internet entries; `sockaddr_iso` is used for ISO (OSI) entries.
`hs_name`	Official service name.
`hs_aliases`	Zero-terminated array of alternative names for the service.
`hs_type`	Socket type.

The `gethostinfo` function returns host information by either host name or host address. If `GHI_HOST_ADDR` (defined in `netdb.h`) is set in the *flags* argument, the *host* field is treated as if it pointed to a `struct sockaddr` structure. The `gethostinfo` function searches for a match to the address contained in the `sockaddr` structure. If `GHI_HOST_ADDR` is not set in the *flags* argument, the *host* field is treated as a host name to be searched for in the database. If *host* is null, only service information is returned from the `gethostinfo` call.

The `gethostinfo` function returns service information by either service name or service address. If `GHI_SERV_ADDR` is set in the *flags* argument, the *service* field is treated as if it pointed to a `struct sockaddr` structure. The `gethostinfo` function searches for a match to the socket port number or address contained in the `sockaddr` structure. If `GHI_SERV_ADDR` is not set in the *flags* argument, the *service* field is treated as a service name to be searched for in the database. If *service* is null, only host information is returned from the `gethostinfo` call.

The *family* field must be `AF_INET` for Internet entries or `AF_ISO` for ISO (OSI) entries.

The *type* field is used for searches based on Internet socket types. These types are listed in the `socket`(2) manual page.

NOTES

All information is contained in a static area that must be copied if it is to be saved. The `/etc/hosts.bin` file is not automatically kept current with `/etc/hosts`. If the administrator forgets to run `mkbinhost`(8), users will get obsolete host information.

RETURN VALUES

A null pointer (0) is returned at end-of-file or when an error occurs. When the null pointer is returned at end-of-file, this indicates that `gethostinfo` did not find the specified name or address in the file.

FILES

```
/etc/hosts
/etc/hosts.bin
/etc/host.usenamed
/etc/services
/usr/include/sys/socket.h
/usr/include/netinet/in.h
/usr/include/netiso/iso.h
/usr/include/netdb.h
```

SEE ALSO

`gethost`(3C), `resolver`(3C)

`socket`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`hosts`(5), `services`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`mkbinhost`(8), `named`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

getlogin, getlogin_r – Gets login name

SYNOPSIS

```
#include <unistd.h>

char *getlogin (void);

int getlogin_r (char *bufname, size_t bufsize);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX (getlogin)
PThreads (getlogin_r)

DESCRIPTION

The getlogin function returns a pointer to the login name as found in /etc/utmp; getlogin_r is the same but specifies a buffer for the name. These may be used in conjunction with getpwnam (see getpwent(3C)) to locate the correct password file entry when the same user ID is shared by several login names.

If getlogin is called within a process that is not attached to a terminal, it returns a null pointer. The correct procedure for determining the login name is to call cuserid(3C), or getlogin, and, if that function fails, to then call getpwuid (see getpwent(3C)).

The getlogin_r function provides functionality equivalent to the getlogin function, but with an interface that is safe for multitasked applications; the caller provides a buffer *bufname* of size *bufsize* for the storage of the login name. The maximum size needed for this buffer can be determined with the _SC_GETLOGIN_R_SIZE_MAX sysconf parameter.

NOTES

The return values from getlogin(3C) point to static data that is overwritten by each call.

RETURN VALUES

The getlogin function returns a null pointer if the process is not attached to a terminal.

On success, the getlogin_r function returns 0. Otherwise it returns an error number:

ERANGE The value of *namesize* is smaller than the length of the string to be returned including the terminating null character.

FILES

`/etc/utmp`	File of user information

SEE ALSO

`cuserid`(3C), `getgrent`(3C), `getpwent`(3C)

`utmp`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

`setmntent`, `getmntent`, `endmntent`, `listmntent`, `getmntinfo`, `findmntentry`, `freemntlist`, `freemntent`, `dupmntent`, `hasmntopt` – Gets file system descriptor file entry or kernel mount table entry

SYNOPSIS

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

FILE *setmntent (char *fname);

struct mntent *getmntent (FILE *filep);

int endmntent (FILE *filep);

int listmntent (struct tabmntent **entlist, char *fname, char *comp,
int (*func)());

struct kmntinfo *getmntinfo (char *fname);

struct mntent *findmntentry (struct tabmntent **entlist,
struct mntent *mnt, int flag, int upd);

void freemntlist (struct tabmntent *tabmnt);

void freemntent (struct mntent *mnt);

struct mntent *dupmntent (struct mntent *mnt);

char *hasmntopt (struct mntent *mnt, char *opt);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

`setmntent`, `getmntent`, `endmntent`, and `hasmntopt` – These routines may exist on other operating systems, but the parameters may be different.
Cray Research extension (`listmntent`, `getmntinfo`, `findmntentry`, `freemntlist`, `freemntent`, and `dupmntent`)

DESCRIPTION

These functions access the file system description file `/etc/fstab` or the mounted file systems in the kernel mount table.

The behavior of the setmntent function depends on whether a file name is specified. If *fname* is specified (usually /etc/fstab), the setmntent function opens a file system description file and returns a file pointer that can then be used with getmntent or endmntent.

If *fname* is not specified (NULL), the setmntent function obtains data about mounted file systems from the kernel mount table and puts it into mntent structures with the following format:

```
struct mntent {
       char  *mnt_fsname;     /* file system name */
       char  *mnt_dir;        /* file system path prefix */
       char  *mnt_type;       /* 4.2, nfs, swap, or xx */
       char  *mnt_opts;       /* ro, quota, etc. */
       int  mnt_freq;         /* dump frequency, in days */
       int  mnt_passno;       /* pass number on parallel fsck */
};
```

For ease of use, the following define statementss were added to the mntent.h include file:

```
#define FSTAB   "/etc/fstab"
#define KMTAB   NULL
```

Depending on the file pointer that the setmntent function returns, the getmntent function reads the next line from *filep* or obtains information about the next mounted file system. The getmntent function returns a pointer to an object with the structure defined as in the preceding mntent structure. The pointer contains either the broken-out fields of a line in the file system description file or the same fields from the kernel mount table. See fstab(5) for a description of the fields.

The mnt_freq and mnt_passno fields are meaningless for the kernel table. Any field that is meaningless contains a pointer to a zero byte.

The endmntent function uses malloc(3C) to close the file or free space previously reserved; the information that is copied from the kernel mount table depends on the file descriptor that the setmntent function returns.

The listmntent function combines the setmntent, getmntent, and endmntent functions to build a linked list of objects with the structure defined in the mntent structure shown previously. The linked list is defined as follows:

```
struct tabmntent {
       struct mntent    *ment;
       struct tabmntent *next;
};
```

After it is built, the list contains either a description of all file systems found in the file *fname*, or a description of all mounted file systems, depending on whether *fname* is specified. *fname* follows the same convention as that of the setmntent function. The listmntent function allows you to build a subset list of file system information, depending on the values of *comp* and *func*. *func* is the name of the comparison function that determines whether file system information should be added to the list. Use the following arguments to call this function:

```
    int func( char *comp, struct mntent *mnt);
```

Generally, *comp* is defined as the argument of the `listmntent` function. If the *mnt* structure must be added to the list, the *func* function always returns 0; otherwise, it returns a nonzero value.

The `getmntinfo` function obtains general information about the file system description file or the kernel mount table, depending on *fname*. *fname* follows the same convention as the `setmntent` function. The `getmntinfo` function returns a pointer to an object with the following structure:

```
  struct kmntinfo {
         int  nbent;
         long lastchge;   /* last time the file or mount table changed */
         :
         :
  };
```

The `findmntentry` function returns a specific entry in a linked list built with the `listmntent` function. The function bases its search on the *flag* definition. Available flags are as follows:

Flag	**Description**
`DIRFLAG`	Finds the entry that has the specified mount point.
`FSFLAG`	Finds the entry that has the specified file system name.
`OPTFLAG`	Finds the first entry with the specified mount options.
`TYPEFLAG`	Finds the first file system with the specified type.

The flags are defined in the `mntent.h` file.

The `findmntentry` function compares data in each entry of the linked list with data found in the *mnt* structure. You can update the pointer to the beginning of the linked list, depending on the value of the *upd* flag. The values are as follows:

Flag	**Description**
`UPD`	Specifies that the beginning of the list points to the entry found during the search.
`UPREC`	Specifies that the beginning of the list points to the entry previous to the one found during the search.
`NOUP`	Specifies that the list remains the same.

These flags are defined in the `mntent.h` file.

If you use `UPD` to call the `findmntentry` function, the function does not release the space for the entries on the list that precedes the found entry. The user must keep a second pointer to those entries so that they can be released later.

The `freemntlist` function frees the linked list that was built with the `listmntent` function.

The `dupmntent` function returns a pointer to a new `mntent` structure, which is a duplicate of the structure to which *mnt* points. Use `malloc`(3C) to obtain the space for the new structure.

The `freemntent` function frees the `mntent` structure to which *mnt* points.

The `hasmntopt` function scans the `mnt_opts` field of the `mntent` structure *mnt* for a substring that matches *opt*. It returns the address of the substring if a match is found; otherwise, it returns 0.

NOTES

The returned `mntent` structure points to static information that is overwritten in each call.

RETURN VALUES

If *fname* is specified, the `setmntent` function returns the following:

- The file pointer on success
- A NULL value on error

If *fname* is NULL or `KMTAB`, the `setmntent` function returns the following:

- `KMNT_FP` on success (defined in `mntent.h`)
- A NULL value on error.

The `getmntent` function returns the following:

- A pointer to the next `mntent` entry on success
- A NULL value on EOF

The `listmntent` function returns the following:

- Zero on success and if the *entlist* argument points to the linked list
- Nonzero on error

The `getmntinfo` function returns the following:

- A pointer to the general information structure on success
- A NULL value on error

The `findmntentry` function returns the following:

- A pointer to the entry found in the list on success
- A NULL value if no entry has been found

The `dupmntent` function returns the following:

- A pointer to the duplicate structure on success
- A NULL value on error

FILES

`/etc/fstab`	File containing static information about system files

EXAMPLES

The following program prints the list of mounted file systems:

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

main()
{
   struct mntent *mnt;
   FILE *fd;

   if ((fd = setmntent(KMTAB)) == NULL) {
           fprintf(stderr,"Cannot get information from the mount table\n");
           exit(1);
   }

   while ((mnt = getmntent(fd)) != NULL) {
           fprintf(stdout, "File system name: %s\n", mnt->mnt_fsname);
           fprintf(stdout, "Mount point: %s\n", mnt->mnt_dir);
           fprintf(stdout, "Options: %s\n", mnt->mnt_opts);
           fprintf(stdout, "File system type: %s\n", mnt->mnt_type);
   }

   endmntent(fd);
}
```

The following program builds a linked list of NFS mounted file systems and prints it:

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

int
nfstype (comp, mnt)
   char          *comp;
   struct mntent *mnt;
{
   return(strcmp(comp, mnt->mnt_type));
}

main()
{
   struct tabmntent *tabmnt;
   struct tabmntent *mnt;

   if (listmntent(&tabmnt, KMTAB, MNTTYPE_NFS, nfstype) != 0) {
           fprintf(stderr, "Cannot build list of NFS mounted FS\n");
           exit(1);
   }

   for (mnt = tabmnt; mnt; mnt = mnt->next) {
          fprintf(stdout, "File system name: %s\n", mnt->ment->mnt_fsname);
          fprintf(stdout, "Mount point: %s\n", mnt->ment->mnt_dir);
          fprintf(stdout, "Options: %s\n", mnt->ment->mnt_opts);
          fprintf(stdout, "File system type: %s\n", mnt->ment->mnt_type);
   }

   freemntlist(tabmnt);
}
```

The following program finds the first NFS mounted file system:

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

main()
{
   struct tabmntent *tabmnt;
   struct mntent    *mnt, template;

   if (listmntent(&tabmnt, KMTAB, NULL, NULL) != 0) {
           fprintf(stderr, "Cannot build list of mounted FS\n");
           exit(1);
   }

   if ((template.mnt_type = (char *)malloc(strlen(MNTTYPE_NFS))) == NULL) {
           fprintf(stderr, "Cannot malloc space\n");
           exit(1);
   }

   strcpy(template.mnt_type, MNTTYPE_NFS);

   if ((mnt = findmntentry(&tabmnt, &template, TYPEFLAG, NOUP)) != NULL) {
          fprintf(stdout, "File system name: %s\n", mnt->mnt_fsname);
          fprintf(stdout, "Mount point: %s\n", mnt->mnt_dir);
          fprintf(stdout, "Options: %s\n", mnt->mnt_opts);
          fprintf(stdout, "File system type: %s\n", mnt->mnt_type);
   }

   freemntlist(tabmnt);
}
```

SEE ALSO

`fopen`(3C), `getfsent`(3C), `malloc`(3C)

`fstab`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

endnetent, getnetbyaddr, getnetbyname, getnetent, setnetent – Gets network entry

SYNOPSIS

```
#include <netdb.h>

int endnetent (void);

struct netent *getnetbyaddr (int net, int type);

struct netent *getnetbyname (char *name);

struct netent *getnetent (void);

int setnetent (int stayopen);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The getnetbyaddr, getnetbyname, and getnetent functions each return a pointer to an object in the network database, /etc/networks. The following structure contains the fields of a line in the network database. In future releases of the UNICOS operating system, the location and format of this database may change, but this interface will remain.

```
struct  netent {
        char            *n_name;        /* official name of net */
        char            **n_aliases;    /* alias list */
        int             n_addrtype;     /* net number type */
        unsigned long   n_net;          /* net number */
};
```

The members of this structure are as follows:

Member	**Description**
n_name	Official name of the network.
n_aliases	Zero-terminated list of alternative names for the network.
n_addrtype	Type of the network number returned; currently only AF_INET and AF_ISO are supported.
n_net	Network number; network numbers are returned in host byte order.

The `setnetent` function opens and rewinds the `/etc/networks` file.

The `endnetent` function closes the `/etc/networks` file.

The `getnetbyaddr` function searches for the network address, and the `getnetbyname` function searches for the network name (or alias), sequentially from the first entry in the database. The search continues until the desired information is found or until the last entry is reached. These functions return network addresses in host byte order. Because `getnetbyaddr` and `getnetbyname` use `setnetent` and `endnetent`, they open and close the file if the *stayopen* flag is 0.

The `getnetent` function reads the next entry in the `/etc/networks` database, opening the database if necessary.

NOTES

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

RETURN VALUES

A null pointer (0) is returned upon end-of-file or error. For `getnetbyaddr` and `getnetbyname`, a null pointer returned upon end-of-file indicates that an entry containing the specified name or address was not found.

FILES

`/etc/networks`

`/usr/include/netdb.h`

SEE ALSO

`networks`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

getopt, optarg, optind, opterr, optopt – Parses command options

SYNOPSIS

```
#include <unistd.h>

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

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX

DESCRIPTION

The getopt function is a command line parser. The *argc* parameter specifies the argument count and the *argv* parameter specifies the argument array. The *optstring* argument contains a string of recognized option characters; if a character is followed by a colon, the option takes an argument.

The variable optind specifies the index of the next element of the *argv* array to be processed. The system initializes it to 1, and the getopt function updates it with each element of *argv*.

The getopt function returns the next option character from the *argv* parameter if one is found that matches a character in the *optstring* argument. If the option takes an argument, the getopt function sets the variable optarg to point to the option argument according to the following rules:

- If the option was the last character in the string, optarg contains the next element of the *argv* parameter, and the optind variable is incremented by 2. If the resulting value of optind is greater than or equal to argc, the getopt function returns an error status.
- Otherwise, optarg points to the string following the option character in that element of the *argv* parameter, and the optind variable is incremented by 1.

If the following conditions are true when the getopt function is called, the getopt function returns a -1 without changing the optind variable:

- *argv*[optind] is a null pointer
- **argv*[optind] is not the character ´-´
- *argv*[optind] points to the string "-"

If the *argv*[optind] parameter points to the "--" string, the getopt function returns -1 after incrementing the optind variable.

If the getopt function encounters an option character that is not contained in the *optstring* parameter, it returns a question mark (?) character. If it detects a missing option argument, it returns a colon (:) character if the first character of the *optstring* parameter was a colon. Otherwise, it returns a question mark (?) character. In either case, the getopt function sets the variable optopt to the option character that caused the error. If the application has not set the variable opterr to 0, and the first character of the *optstring* parameter is not a colon, the getopt function also prints a diagnostic message to the stderr file in the format specified by the getopts(1) command.

WARNINGS

The getopt function uses the header file stdio.h, which causes it to increase the size of programs more than might otherwise be expected.

RETURN VALUES

The getopt function returns the next option character specified on the command line.

A colon (:) is returned if the getopt function detects a missing argument and the first character of the *optstring* argument was a colon.

A question mark (?) is returned if the getopt function encounters an option character not in the *optstring* argument or detects a missing argument and the first character of the *optstring* argument is not a colon.

Otherwise, the getopt function returns -1 when all command line options are parsed.

EXAMPLES

The following code fragment shows how you 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 <stdlib.h>
#include <stdio.h>
#include <unistd.h>

main (int argc, char * argv[])
{
        int c;
        .
        .
        .
        while ((c = getopt (argc, argv, "abf:o:")) != -1){
                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++;
                  break;
                }
        }
        if (errflg) {
                fprintf (stderr, "usage: . . . \n");
                exit (2);
        }
        for ( ; optind < argc; optind++) {
                if (access (argv[optind], 4)) {
                .
                .
                .
                }
        }
}
```

FORTRAN EXTENSION

The functionality of `getopt` is available in Fortran through the integer functions `GETOPTC`, `GETVARGC`, and `GETOARGC`. For most applications, only `GETOPTC` is needed. `GETOPTC` returns the next character found in the string of characters, *optstr*, or -1 when no option characters can be found.

The following example shows the call to GETOPTC:

```
INTEGER*8 GETOPTC, IARG
CHARACTER OPTSTR *n, OPTARG *m
IARG = GETOPTC(optstr, optarg)
```

Both arguments must always be present, but optarg is used only when an individual option letter (IARG) has arguments.

The GETOPTC call works the same as getopt, with the following exceptions:

- If a letter in *optstr* is followed by a colon (:), exactly one argument is expected for the option; it is copied into optarg.
- If a letter in *optstr* is followed by a semicolon (;), zero or more arguments are expected for the option. You must then call GETVARGC to get the variable arguments until GETVARGC returns 0 before the next call to GETOPTC.

The GETVARGC call has the following format:

```
INTEGER*8 GETVARGC, MOREARG
CHARACTER VARG *n
MOREARG = GETVARGC (varg)
```

The next variable argument is copied into the character variable *varg*. GETVARGC returns 0 when no more variable arguments exist.

After GETOPTC returns -1, you can call GETOARGC to get the remaining arguments from the command line.

GETOARGC has the following format:

```
INTEGER*8  GETOARGC, MOREARG
CHARACTER OARG *n
MOREARG = GETOARGC (oarg)
```

GETOARGC returns 0 if there are no more arguments. The next remaining argument is copied into the character variable *oarg*.

If GETOPTC is not used, GETOARGC can be called to get the command line arguments in order, starting with the first argument.

On systems others than Cray MPP systems and the CRAY T90 series, the integer functions GETOPT, GETVARG, and GETOARG are also available. These provide functionality similar to GETOPTC, GETVARGC, and GETOARGC. They are called as follows:

```
INTEGER*8 GETOPT, IARG
IARG = GETOPT(optsh,optarg,optargsz)
```

The `optarg` argument is an array of *optargsz* words into which `GETOPT` places the string of characters that represents the argument associated with `IARG`. The *optargsz* argument is ignored if `optarg` is a character variable.

The `GETVARG` call has the following format:

```
INTEGER*8 GETVARG, MOREARG
MOREARG = GETVARG (varg, vargsz)
```

The next variable argument is copied into the array *varg* (of size *vargsz*). The `GETVARG` call returns `0` when no more variable arguments exist.

After `GETOPT` returns -1, you can call `GETOARG` to get the remaining arguments from the command line.

The `GETOARG` call has the following format:

```
INTEGER*8 GETOARG, MOREARG
MOREARG = GETOARG (oarg, oargsz)
```

The `GETOARG` call returns 0 if there are no more arguments. The next remaining argument is copied into the array *oarg* (of size *oargsz*).

If `GETOPT` is not used, `GETOARG` can be called to get the command line arguments in order, starting with the first argument.

Fortran Examples

Example 1: The following example shows how the options of a command might be processed using `GETOPTC`. This example assumes `a` and `b`, which have arguments, and `x` and `y`, which do not.

```
      CHARACTER*8 OPTIONS
      CHARACTER*80 ARGMNTS
      CHARACTER OPLET
      INTEGER*8 GETOPTC
      INTEGER*8 OPTVAL
      DATA OPTIONS/'a:b;:xy'/

 100  CONTINUE
      OPTVAL = GETOPTC(OPTIONS,ARGMNTS)
      IF (OPTVAL .EQ. -1) GOTO 200
      OPLET = CHAR(OPTVAL)
      IF (OPLET .EQ. 'a') THEN
C   Analyze arguments from ARGMNTS
      ELSEIF (OPLET .EQ. 'b') THEN
C   Analyze arguments from ARGMNTS
      ELSEIF (OPLET .EQ. 'x') THEN
C   Process x option
      ELSEIF (OPLET .EQ. 'y') THEN
C   Process y option
      ENDIF
      GOTO 100
 200  CONTINUE
```

Example 2: The following example illustrates the use of GETOPT and GETOARG together:

```
      PROGRAM TEST
      EXTERNAL GETOPT, GETOARG
      INTEGER*8 GETOPT, GETOARG
      INTEGER*8 ARGLEN
      PARAMETER (ARGLEN = 10)
      INTEGER*8 OPT, DONE, ARGBUF(ARGLEN)

 10   CONTINUE       OPT = GETOPT('abo:',ARGBUF, ARGLEN)
      IF (OPT .GE. 0) THEN           IF (OPT .EQ. 'a'R) THEN
          PRINT '(a)', ' option -a- present '

          ELSEIF (OPT .EQ. 'b'R) THEN
          PRINT '(a)', ' option -b- present '
          ELSEIF (OPT .EQ. ’o’R) THEN
          PRINT ’(a,a8)’, ’ option -o- present-’,argbuf(1)
          ELSE C    unknown option
          PRINT ’(a,a8)’, ’ bad option present-’,opt           ENDIF
          GOTO 10       ENDIF C    all options processed C
C    Get arguments  20   CONTINUE
      DONE = GETOARG(ARGBUF, ARGLEN)       IF (DONE .NE. 0) THEN
      PRINT ’(a,a8)’, ’ argument present-’,argbuf(1)       GOTO 20
      ENDIF C    Done processing arguments       END
```

SEE ALSO

`getopts`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

`getoptlst` – Gets option argument list

SYNOPSIS

```
#include <stdlib.h>

int getoptlst (char *optarg, char ***optargv);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The `getoptlst` function parses the list to which the given *optarg* points. This list is not modified. A single block of memory, which must be large enough to contain pointers to consecutive elements and the elements themselves, is allocated by this function, using `malloc`(3C). The returned *optargv* serves as a pointer both to this single block of memory and to consecutive element pointers. Thus, the returned *optargv* can be used by the calling function to obtain consecutive elements in the list and, once the list is complete, to free (using `free`) the space allocated by `malloc`.

An array of pointers is stored in consecutive locations, beginning in the location indicated by the returned *optargv*. These pointers point to elements that were obtained from the list to which the given *optarg* points. The first pointer points to the first element, the second pointer points to the second element, and so on. The number of pointers is represented by the return value of this function. Also, `getoptlst` inserts a null pointer at the end of the array. Elements to which the array of pointers point are represented as null-terminated character strings.

The list to which the given *optarg* points is assumed to be a null-terminated string of characters, which is not modified by this function. Elements in this string are separated by one of the following: white space (including blank, tab, or new-line chracters), a comma, or the final null character. Any character (other than a null character) that is preceded by a backslash is interpreted as the single character following the backslash (the \ is removed). The special meaning for that character, if any, is removed. Thus, you can force a backslash, blank, tab, new-line character, or comma into any element within the list by preceding it with a backslash character. If the backslash character is the last character of the string to which the given *optarg* points (that is, it precedes a terminating null byte), the backslash is treated as a normal character and is not removed.

An empty list contains one empty element. Empty fields can be recognized when the comma or null-byte terminator is used as a separator. Example:

```
,  element2
```

The above list contains one empty element followed by a second element that has a value of `element2`.

RETURN VALUES

If an error is encountered in this function, the return value is –1. (The only possible error is the failure of `malloc`(3C).) Otherwise, the return value equals the number of elements in the list. If the given *optarg* is null, the return value is set to 0; the *optargv* returned is set to null in this case. If the given *optarg* is not null, the return value is set to 1 or greater (there is always at least one element in this case, even though it may be an empty element).

EXAMPLES

The `getoptlst` function is intended for use with the `getopt`(3C) library function. The following code fragment shows how you might process the arguments of a command by using both `getopt` and `getoptlst`. The `-l` option requires an argument in this example; this argument is processed as a list of elements.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

main(argc, argv)
int argc;
char ∗argv[ ];
{
   static int aflg = 0, bflg = 0, errflg = 0;
   static char ∗ifile, ∗ofile;
   char ∗∗optargv;
   int optargc, c, i;
   void bproc (void);

   while ((c = getopt(argc, argv, "abf:l: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 'l':
         if ((optargc = getoptlst(optarg, &optargv)) < 0)
            errflg++;
         else {
            fprintf(stdout, "Elements: %d\n", optargc);
            for (i = 0; i < optargc; i++) {
               fprintf(stdout, "optargv[%d]: '%s'\n", i, optargv[i]);
            }
            free(optargv);
         }
         break;
      case 'o':
         ofile = optarg;
         break;
      case '?':
         errflg++;
   }  }
   if (errflg) {
      fprintf(stderr, "Usage: ...\n");
      exit (2);
   }
   for (; optind < argc; optind++) {
      if (!access(argv[optind], 4)) {
         fprintf(stdout, "%s readable\n", argv[optind]);
      }
      else {
         fprintf(stdout, "%s NOT readable\n", argv[optind]);
      }
   }
}

void bproc(void);
{
   fprintf(stdout, "bproc called\n");
}
```

SEE ALSO

getopt(3C), malloc(3C)

NAME

getpass – Reads a password

SYNOPSIS

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

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The getpass function reads up to a new-line character or EOF from file /dev/tty; before doing so, it prompts on the standard error output with the null-terminated string *prompt* and disables echoing.

Upon successful completion, a pointer is returned to a null-terminated string of at most PASS_MAX (defined in <limits.h>) characters. If /dev/tty cannot be opened, a null pointer is returned. An interrupt terminates input and sends an interrupt signal to the calling program before returning.

NOTES

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

FILES

/dev/tty

SEE ALSO

getpwent(3C), libudb(3C)

udb(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

endprotoent, getprotobyname, getprotobynumber, getprotoent, setprotoent – Gets protocol entry

SYNOPSIS

```
#include <netdb.h>

int endprotoent (void);

struct protoent *getprotobyname (char *name);

struct protoent *getprotobynumber (int proto);

struct protoent *getprotoent (void);

int setprotoent (int stayopen);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The getprotobynumber, getprotobyname, and getprotoent functions each return a pointer to an object in the network protocol database, /etc/protocols. The following structure contains the fields of a line in the network protocol database:

```
struct  protoent  {
        char    *p_name;        /* official name of protocol */
        char    **p_aliases;    /* alias list                */
        int     p_proto;        /* protocol number           */
};
```

The members of this structure are as follows:

Member	**Description**
p_name	Official name of the protocol.
p_aliases	Zero-terminated list of alternative names for the protocol.
p_proto	Protocol number; protocol numbers are returned in host byte order.

The setprotoent function opens and rewinds the /etc/protocols file. If the *stayopen* flag is nonzero, the /etc/protocols file remains open across getproto* calls until closed by entprotoent.

The `endprotoent` function closes the `/etc/protocols` file only if the *stayopen* flag to `setprotoent` is 0. Otherwise, `endprotoent` leaves the file open.

The `getprotobyname` function searches for the protocol name (or alias), *name*, and the `getprotobynumber` function searches for the protocol number, *proto*, sequentially from the first entry in the database. The search continues until the desired information is found or until the last entry is reached.

The `getprotoent` function reads the next entry in the database, opening the database if necessary.

NOTES

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

RETURN VALUES

A null pointer (`0`) is returned upon end-of-file or error. For `getprotobyname` and `getprotobynumber`, a null pointer returned upon end-of-file indicates that the search did not find the specified name or number.

FILES

`/etc/protocols`

`/usr/include/netdb.h`

SEE ALSO

`protocols`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

getpw – Gets name from UID

SYNOPSIS

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

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

AT&T extension

DESCRIPTION

The getpw function searches the password file for a user ID number that equals *uid*, copies the line of the password file in which *uid* was found into the array pointed to by *buf*, and returns 0. The getpw function returns a nonzero value if *uid*" cannot be found.

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

WARNINGS

The preceding function uses header stdio.h, which causes it to increase the size of programs more than might otherwise be expected.

FILES

/etc/passwd

RETURN VALUES

The getpw function returns nonzero on error.

SEE ALSO

getpwent(3C), libudb(3C)

passwd(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

getpwent, getpwuid, getpwuid_r, getpwnam, getpwnam_r, setpwent, endpwent, fgetpwent – Gets password file entry

SYNOPSIS

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

struct passwd *getpwent (void);

struct passwd *getpwuid (int uid);

int getpwuid_r (int uid, struct passwd *pwd, char *buf, size_t bufsize, struct
passwd **result);

struct passwd *getpwnam (const char *name);

int getpwnam_r (char *name, struct passwd *pwd, char *buf, size_t bufsize,
struct passwd **result);

void setpwent (void);

void endpwent (void);

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX (getpwnam and getpwuid)
PThreads (getpwnam_r and getpwuid_r)
AT&T extension (getpwent, setpwent, endpwent, and fgetpwent)

DESCRIPTION

The getpwent, getpwuid, and getpwnam functions each return a pointer to an object with the following structure, which contains the broken-out fields of one entry from either the /etc/udb or the /etc/udb.public file. The UNICOS user-information database, or udb(5), file is a superset of the information in the passwd(5) file; its use is mandatory. Generally, the udb file can be accessed only by super users. The udb.public file is always available and provides information to nonprivileged programs.

The information for one user's entry is described in the following structure, which is declared in header file pwd.h:

```
struct passwd {
       char   *pw_name;         /* login name */
       char   *pw_passwd;       /* encrypted password */
       uid_t  pw_uid;           /* UID */
       gid_t  pw_gid;           /* GID */
       char   *pw_age;          /* password age */
       char   *pw_comment;      /* comment */
       char   *pw_gecos;
       char   *pw_dir;          /* default login directory */
       char   *pw_shell;        /* default login shell / program */
};
```

The pw_comment and pw_gecos fields point to the same string.

The getpwent function, when first called, returns a pointer to the first passwd structure in the user data-base; thereafter, it returns a pointer to the next passwd structure in the file. Therefore, successive calls can be used to search the entire file.

The getpwuid function uses the getudbuid function to find the first numerical user ID matching *uid*, translates the udb information into the passwd structure, and returns a pointer to the structure containing the information for the entry associated with *uid*.

The getpwnam function uses the getudbnam function to find a login name matching *name*, translates the udb information into the passwd structure, and returns a pointer to the structure containing the information for the entry associated with *name*.

If an error is encountered on accessing the udb, or if the requested information could not be found, these functions return a null pointer.

A call to setpwent uses the function setudb, which has the effect of rewinding the udb to allow repeated searches. The endpwent function may be called to close the file when processing is complete.

The fgetpwent function returns a pointer to the next passwd structure in stream *f*; which must match the format of /etc/passwd (see passwd(5)). This function is included only for compatibility with prior systems; use of fgetpwent in new code is discouraged.

The functions whose names end with _r, getpwuid_r and getpwnam_r, provide equivalent functionality but with an interface that is safe for multitasked applications. The primary difference between these interfaces is that instead of returning a pointer to a structure, they put the results into the structure pointed to by the *pwd* argument. In addition, they use the provided buffer *buf* of size *bufsize* to store auxilary data. The maximum size needed for this buffer can be determined with the _SC_GETPW_R_SIZE_MAX sysconf parameter. A NULL pointer is returned at the location pointed to by *result* on error or if the required entry is not found.

NOTES

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

Unless the caller is a super user, calls using function `getpwnam` return the indicated information minus the encrypted password field.

Since these calls use the `getudb`*xxx* functions to perform their function, mixing `getpw`*xxx* and `getudb`*xxx* calls may have unexpected side effects. This is a concern if sequential reading is being done through `getpwent` while `getudb`*xxx* calls are also being issued in the same program.

WARNINGS

Successive calls to `getpwent`, `getpwuid`, and `getpwnam` return a pointer to the same static `passwd` structure each time they are called; these calls overwrite the same data area. Use caution when working with more than one `passwd` structure at a time.

The `getpwent routine leaves the udb` file open to assure reasonable performance for multiple calls; the `getpwuid` and `getpwnam` calls close the `udb` file before returning. If it is important that the program in which the `getpwent` calls are made can be restarted, an `endpwent` call must be made to close the `udb` file after the access is complete.

RETURN VALUES

For all interfaces other than `getpwuid_r` and `getpwnam_r`, a null pointer is returned on `EOF` or error. For `getpwuid_r` and `getpwnam_r`, 0 is returned on success. Otherwise an error number is returned:

`ERANGE` Insufficient storage was supplied via *buf* and *bufsize* to contain the data to be referenced by the resulting *struct passwd* structure.

FILES

`/etc/passwd`

`/etc/udb`

`/etc/udb.public`

SEE ALSO

`getgrent`(3C), `getlogin`(3C), `id2nam`(3C), `libudb`(3C)

`passwd`(5), `udb`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

getrpcent, endrpcent, getrpcbyname, getrpcbynumber, setrpcent – Gets remote procedure call entry

SYNOPSIS

```
#include <rpc/netdb.h>
```

```
struct rpcent *getrpcent(void)
```

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

struct rpcent *getrpcbynumber (int *number*);

int setrpcent (int *stayopen*);

```
int endrpcent (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The getrpcent, getrpcbyname, and getrpcbynumber functions each return a pointer to an object with the following structure that contains the broken-out fields of a line of the remote procedure call (RPC) program number database (/etc/rpc):

```
 struct  rpcent {
         char    *r_name;        /* name of server for this RPC program */
         char    **r_aliases;    /* Alias list */
         long    r_number;       /* RPC program number */
 };
```

A breakdown of this structure is as follows:

Member	**Description**
r_name	The name of the server for this RPC program
r_aliases	A zero-terminated list of alternative names for the RPC program
r_number	The RPC program number for this service

The getrpcent function reads the next line of the file and opens the file if necessary.

The `setrpcent` function opens and rewinds the file. If the *stayopen* flag is nonzero, the RPC program number database does not close after each call to `getrpcent`, whether the call is direct or indirect (that is, made through one of the other `getrpcent` calls).

The `endrpcent` command closes the file.

The `getrpcbyname` and `getrpcbynumber` function search sequentially from the beginning of the file until a matching RPC program name or program number is found, or until an end-of-file (EOF) marker is encountered.

NOTES

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

RETURN VALUES

A null pointer (0) is returned when reaching EOF or an error.

FILES

`netdb.h`

`/etc/rpc`

`/etc/yp/`*domainname*`/rpc.bynumber`

SEE ALSO

`rpc`(3C)

`rpcinfo`(8), `ypserv`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`gets`, `fgets`, `fgetws` – Gets a string from a stream

SYNOPSIS

```
#include <stdio.h>
char *gets (char *s);
char *fgets (char *s, int n, FILE *stream);
#include <wchar.h>
wchar_t *fgetws(wchar_t *ws, int n, FILE *stream);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (`gets` and `fgets` only)
XPG4 (`fgetws` only)

DESCRIPTION

The `gets` function reads characters from the standard input stream, `stdin`, into the array to which *s* points, until a newline character is read or an end-of-file (EOF) condition is encountered. The newline character is discarded, and the string is terminated with a null character.

The `fgets` function reads characters from the specified *stream* into the array to which *s* points, until *n*– *1* characters are read, a newline character is read and transferred to *s*, or an EOF condition is encountered. The string is then terminated with a null character.

The `fgetws` function reads characters from the *stream*, converts these to the corresponding wide-character codes, places them in the `wchar_t` array to which *ws* points, until *n*– 1 characters are read, or a newline character is read, converted and transferred to *ws*, or an EOF condition is encountered. The wide character string, *ws*, is then terminated with a null wide-character code. The `fgetws` function may mark the `st_atime` field of the file associated with *stream* for update. The `st_atime` field is marked for update by the first successful execution of `fgetc`(3C), `fgets`, `fgetwc`(3C), `fgetws`, `fread`(3C), `fscanf`(3C), `getc`(3C), `getchar`(3C), `gets`, or `scanf`(3C) by using *stream* that returns data not supplied by a prior call to `ungetc`() or `ungetwc`().

CAUTIONS

The use of `gets` is discouraged because of the potential for memory overwrites.

RETURN VALUES

These functions return *s* or *ws* if successful. If an EOF is encountered and no characters have been read, no characters are transferred to *s* and a null pointer is returned. (To determine if an EOF was reached, call `feof`(3C).) If a read error occurs, such as that caused by trying to use these functions on a file that has not been opened for reading, the array contents are indeterminate and a null pointer is returned.

FORTRAN EXTENSIONS

You also can call the `fgets` function from Fortran programs, as follows:

```
INTEGER*8 FGETS, stream, s (m), n, I
I = FGETS(s, n, stream)
```

or

```
CHARACTER *len s (m), n, I
I = FGETS(s, n, stream)
```

Argument *m* is an integer constant that specifies the number of elements in array *s*. If the second declaration is used for array *s*, *len* is the length in characters of each element in character array *s*.

SEE ALSO

`ferror`(3C), `fopen`(3C), `fread`(3C), `getc`(3C), `scanf`(3C)

NAME

endservent, getservbyname, getservbyport, getservent, setservent – Gets service entry

SYNOPSIS

```
#include <netdb.h>

int endservent (void);

struct servent *getservbyname (char *name, char *proto);

struct servent *getservbyport (int port, char *proto);

struct servent *getservent (void);

int setservent (int stayopen);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The getservbyname, getservbyport, and getservent functions each return a pointer to an object in the network services database, /etc/services. The following structure contains the fields of a line in the network services database:

```
    struct  servent {
            char    *s_name;        /* official name of service */
            char    **s_aliases;    /* alias list */
            int     s_port;         /* port service resides at */
            char    *s_proto;       /* protocol to use */
    };
```

The members of this structure are as follows:

Member	**Description**
s_name	Official name of the service.
s_aliases	Zero-terminated list of alternative names for the service.
s_port	Port number at which the service resides; port numbers are returned in network byte order.
s_proto	Name of the protocol to use when contacting the service.

The setservent function opens and rewinds the /etc/services file. If the stayopen flag is nonzero, the /etc/service file will remain open across getservbyname and getservbyport calls until closed by the endservent function.

The endservent function closes the /etc/services file. Otherwise, endservent leaves the file open.

The getservbyname function searches for the service name (or alias) *name*, and the getservbyport function searches for the port number *port* at which the service resides, sequentially from the first entry in the database. If the address type proto is nonzero, the s_proto field of the database entry must also match proto; otherwise, the s_proto field is ignored. The search continues until the desired information is found or until the last entry is reached. If the optional proto argument is specified, the proto argument must match the s_proto field in the database entry. Because getservbyname and getservbyport use setservent and endservent, they open and close the file if the *stayopen* flag is 0.

The getservent function reads the next entry in the database, opening the database if necessary.

All of these functions call gethostinfo(3C) functions to perform the searches.

NOTES

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

RETURN VALUES

A null pointer (0) is returned upon end-of-file or error. For getservbyname and getservbyport, a null pointer returned upon end-of-file indicates that an entry containing the specified name or port number was not found in the database.

FILES

/etc/services

/usr/include/netdb.h

SEE ALSO

gethostinfo(3C), getprot(3C)

services(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

endtosent, gettosbyname, gettosent, parsetos, settosent – Gets network Type Of Service information

SYNOPSIS

```
#include <netdb.h>

int endtosent (void);

struct tosent *gettosbyname (char *name, char *proto);

struct tosent *gettosent (void);

int parsetos (char *name, char *proto);

int settosent (int stayopen);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The gettosbyname and gettosent functions each return a pointer to an object that describes a Type Of Service (TOS) entry. The information in the TOS entry is obtained from the file /etc/iptos. The following structure describes a TOS entry:

```
struct      tosent {
        char     *t_name;
        char    **t_aliases;
        char     *t_proto;
        int       t_tos;
}
```

The members of this structure are as follows:

Member	**Description**
t_name	Official name of the TOS.
t_aliases	Zero-terminated array of alternative names for the TOS.
t_proto	The name of the IP protocol for which the TOS entry applies. Examples are tcp, udp, icmp, and the wildcard, *.
t_tos	The actual TOS bits for this entry.

The `settosent` function opens and rewinds the `/etc/iptos` file. If the *stayopen* flag is nonzero, successive calls to `gettosbyname` do not close and reopen the `/etc/iptos` file.

The `endtosent` function closes the `/etc/iptos` file.

The `gettosbyname` function fetches information for the TOS with name (or alias) *name* for the protocol *proto*. If *proto* is null or the string `*` (a single asterisk), the `gettosbyname` function fetches information for the first encountered TOS with name *name*, regardless of protocol. The `gettosbyname` function uses the `settosent` and `endtosent` functions, thus opening and closing the file, if the *stayopen* flag is 0.

The `gettosent` function returns the next entry in the `/etc/iptos` database, opening the file if necessary.

The `parsetos` function returns the actual `t_tos` TOS value from the `tosent` structure for the specified *name* and *proto* fields, as returned by `gettosbyname`. If the `gettosbyname` function does not find an appropriate `tosent` value, the `parsetos` function returns the presumed numeric value that is specified in the string `name`.

NOTES

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

RETURN VALUES

The `gettosbyname` function returns NULL at the end-of-file or when an error occurs. When the null pointer is returned at end-of-file, this indicates that `gettosbyname` did not find the specified name or address in the file.

The `parsetos` function returns the actual TOS value, or returns –1 and sets `errno` if it detects an error, as follows:

Error	**Description**
`EINVAL`	No TOS entry for the name *name* is found, and *name* is not a numeric string.
`ERANGE`	The specified TOS value is outside the legal range of TOS values (`0` to `255`).

FILES

`/etc/iptos`

`/usr/include/netdb.h`

SEE ALSO

`iptos`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

getusershell, setusershell, endusershell – Gets user shells

SYNOPSIS

```
#include <stdlib.h>
char *getusershell (void);
int setusershell (void);
int endusershell (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The getusershell function returns a pointer to a user shell as defined by the system manager in the file /etc/shells. If /etc/shells does not exist, pointers to the standard system shells /bin/sh, /bin/csh, and /bin/ksh are returned.

The getusershell function reads the next line (opening the file if necessary); setusershell rewinds the file; endusershell closes it.

NOTES

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

RETURN VALUES

The getusershell function returns a null pointer on end-of-file or error.

FILES

/etc/shells	File that contains a list of available shells.

NAME

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

SYNOPSIS

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

struct utmp *getutent (void);

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

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

struct utmp *pututline (const struct utmp *utmp);

void setutent (void);

void endutent (void);

int utmpname (const char *file);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

AT&T extension

DESCRIPTION

The getutent, getutid, and getutline functions each return a pointer to a structure of type struct utmp, which is defined in header file <utmp.h>. (See utmp(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014.)

The getutent function reads in the next entry from a utmp-like file. If the file is not already open, getutent opens it. If getutent reaches the end of the file, it returns a null pointer.

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

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

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

The setutent function resets the input stream to the beginning of the file. This should be done before each search for a new entry if the entire file is to be examined.

The endutent function closes the currently open file.

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

NOTES

The most current entry is saved in a static structure. Multiple accesses require that the current entry be copied before further accesses are made. Upon each call, getutid or getutline examine the static structure before performing more I/O. If the contents of the static structure match what the function is searching for, it looks no further. For this reason, using getutline to search for multiple occurrences necessitates zeroing out the static structure after each success to prevent getutline from returning the same pointer over and over again.

There is one exception to the rule about removing the structure before further reads are done. The static structure contents are not harmed in an implicit read done by pututline if the function finds that it is not already at the correct place in the file. This is true even if you have just modified those contents and passed the pointer back to pututline.

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

RETURN VALUES

The pututline function returns a null pointer if it fails; otherwise, it returns a pointer to a copy of the structure.

The utmpname function returns 0 if it fails; otherwise, it returns 1.

The other functions return a null pointer upon failure to read (whether due to the lack of necessary permissions or due to reaching the end-of-file) or upon failure to write.

FILES

/etc/utmp	File of user information

`/etc/wtmp`	File of user information

SEE ALSO

`utmp`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

`getwd` – Gets current directory path name

SYNOPSIS

```
#include <sys/param.h>
#include <unistd.h>
```

```
char *getwd (char *pathname);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The `getwd` function copies the absolute path name of the current directory to *pathname* and returns a pointer to the result.

CAUTIONS

The length of the *pathname* array should be at least `PATH_MAX` characters, as defined in the header file `sys/param.h`.

RETURN VALUES

The `getwd` function returns `0` and places a message in *pathname* if an error occurs.

SEE ALSO

`getcwd`(3C)

NAME

glob, globfree – Generates path names matching a pattern

SYNOPSIS

```
#include <glob.h>

int glob(const char *pattern, int flags,
int (*errfunc) (const char *, int), glob_t *pglob);

void globfree (glob_t *pglob);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX

DESCRIPTION

The glob function is a path name generator that implements the rules for file name pattern matching used by the shell.

The include file glob.h defines the structure type glob_t, which contains at least the following fields:

```
typedef struct {
      int gl_pathc;          /* count of total paths so far */
      int gl_matchc;         /* count of paths matching pattern */
      int gl_offs;           /* reserved at beginning of gl_pathv */
      int gl_flags;          /* returned flags */
      char **gl_pathv;       /* list of paths matching pattern */
} glob_t;
```

The argument *pattern* is a pointer to a path name pattern to be expanded. The glob argument matches all accessible path names against the pattern and creates a list of the path names that match. In order to have access to a path name, glob requires search permission on every component of a path except the last and read permission on each directory of any file name component of pattern that contains any of the special characters *, ?, or [.

The glob argument stores the number of matched path names into the gl_pathc field, and a pointer to a list of pointers to path names into the gl_pathv field. The first pointer after the last path name is NULL. If the pattern does not match any path names, the returned number of matched paths is set to zero.

It is the caller's responsibility to create the structure pointed to by pglob. The glob function allocates other space as needed, including the memory pointed to by gl_pathv.

The argument *flags* is used to modify the behavior of glob. The value of *flags* is the bitwise inclusive OR of any of the following values defined in glob.h :

Value **Description**

GLOB_APPEND Appends path names generated to those from a previous call (or calls) to glob. The value of gl_pathc will be the total matches found by this call and the previous call(s). The path names are appended to, not merged with the path names returned by the previous call(s). Between calls, the caller must not change the setting of the GLOB_DOOFFS flag, nor change the value of gl_offs when GLOB_DOOFFS is set, nor (obviously) call globfree for pglob.

GLOB_DOOFFS Causes the gl_offs field to specify how many null pointers should be prepended to the beginning of the gl_pathv field. That is, gl_pathv will point to gl_offs null pointers, followed by gl_pathc path name pointers, followed by a null pointer.

GLOB_ERR Causes glob to return when it encounters a directory that it cannot open or read. Ordinarily, glob continues to find matches.

GLOB_MARK Appends a slash to each path name that is a directory matching pattern.

GLOB_NOCHECK Causes the following result if *pattern* does not match any path name: glob returns a list consisting of only *pattern*, with the total number of path names set to 1, and the number of matched path names set to 0. If GLOB_QUOTE is set, its effect is present in the pattern returned.

GLOB_NOMAGIC Has the same effect as GLOB_NOCHECK but appends pattern only if it contains none of the special characters *, ?, or [. GLOB_NOMAGIC is needed only to simplify implementation of the historic behavior of glob under csh(1).

GLOB_NOSORT Disables sorting of path names in ascending ASCII order; this increases the performance of glob.

GLOB_QUOTE Enables the backslash (\) character for quoting. Every occurrence of a backslash followed by a character in the pattern is replaced by that character, preventing any special interpretation of the character.

If, during the search, a directory is encountered that cannot be opened or read and errfunc is non-NULL, glob calls (*errfunc)(path,errno). This may be counterintuitive: a pattern such as */Makefile will try to stat foo/Makefile even if foo is not a directory, resulting in a call to errfunc. The error routine can suppress this action by testing for ENOENT and ENOTDIR; however, the GLOB_ERR flag will still cause an immediate return when this happens.

If errfunc returns nonzero, glob stops the scan and returns GLOB_ABEND after setting gl_pathc and gl_pathv to reflect any paths already matched. This happens also if an error is encountered and GLOB_ERR is set in *flags*, regardless of the return value of errfunc, if called. If GLOB_ERR is not set and either errfunc is NULL or errfunc returns zero, the error is ignored.

The globfree function frees any space associated with pglob from a previous call(s) to glob.

RETURN VALUES

On successful completion, glob returns zero. In addition, the fields of pglob contain the following values:

Value	Description
gl_pathc	Total number of matched path names so far. This includes other matches from previous invocations of glob if GLOB_APPEND was specified.
gl_matchc	Number of matched path names in the current invocation of glob.
gl_flags	Copy of the flags parameter with the GLOB_MAGCHAR bit set if pattern contained any of the special characters *, ? or [; the bit is cleared if not these characters were absent.
gl_pathv	Pointer to a NULL-terminated list of matched path names. However, if gl_pathc is zero, the contents of gl_pathv are undefined.

If glob terminates due to an error, it sets errno and returns one of the following nonzero constants, which are defined in the include file glob.h:

Constant	Description
GLOB_NOSPACE	An attempt to allocate memory failed.
GLOB_ABEND	The scan was stopped because an error was encountered and either GLOB_ERR was set or (*errfunc)() returned nonzero.

The arguments pglob->gl_pathc and pglob->gl_pathv are still set as specified above.

NOTES

Patterns longer than MAXPATHLEN may cause unchecked errors.

The glob argument may fail and set errno for any of the errors specified for the stat(2) system call, and the library routines closedir(3C), opendir(3C), readdir(3C), malloc(3C), and free(3C).

EXAMPLES

A rough equivalent of "ls -l *.c *.h" can be obtained with the following code:

```
GLOB_t g;

g.gl_offs = 2;
glob("*.c", GLOB_DOOFFS, NULL, &g);
glob("*.h", GLOB_DOOFFS | GLOB_APPEND, NULL, &g);
g.gl_pathv[0] = "ls";
g.gl_pathv[1] = "-l";
execvp("ls", g.gl_pathv);
```

SEE ALSO

sh(1) and csh(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

fnmatch(3C), regexp.h(3C), wordexp(3C)

NAME

`herror` – Produces host lookup error messages

SYNOPSIS

```
#include <netdb.h>
void herror (char *s);
int h_nerr;
char *h_errlist[];
int h_errno;
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The `herror` function writes a short error message to `stderr`, describing the last error encountered during a host lookup. The argument string *s* is printed first (if it is not null). Next, a colon and a blank are printed, followed by the message and a new line. To be of most use, the argument string should include the name of the program (and possibly the name of the subfunction) that encountered the error. The error number is taken from the external variable `h_errno`, which is set when host lookup errors occur, but not cleared when successful calls are made.

To simplify variant formatting of messages, the vector of message strings, known as the `h_errlist` table, is provided. `h_errno` can be used as an index in this table to get the message string without the new line. The number of messages provided for in the table is stored in `h_nerr`. This field should be checked to ensure that an error code in `h_errno` has a corresponding message string in the table.

NOTES

The function `herror` and the objects `h_nerr`, `h_errlist`, and `h_errno` will not work with code that is multitasked.

SEE ALSO

`gethost`(3C), `resolver`(3C), `stdio.h`(3C)

NAME

`hsearch`, `hcreate`, `hdestroy` – Manages hash search tables

SYNOPSIS

```
#include <search.h>
ENTRY *hsearch (ENTRY item, ACTION action);
int hcreate (size_t nel);
void hdestroy (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The hash-table search function `hsearch` is generalized from Knuth (6.4) Algorithm D. It returns a pointer into a hash table indicating the location at which an entry can be found. *item* is a structure of type `ENTRY` (defined in header file <`search.h`>) 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 `void` should be cast to pointer-to-void.) *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` function allocates sufficient space for the table and must be called before `hsearch` is used. The *nel* argument is an estimate of the maximum number of entries that the table will contain. You can use the algorithm to adjust this number upward to obtain certain mathematically favorable circumstances.

The `hdestroy` function destroys the search table, and it can be followed by another call to `hcreate`.

NOTES

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

The `hsearch` function uses open addressing with a multiplicative hash function. Its source code, however, has many other options available; you may select options by compiling the `hsearch` source with the following symbols defined to the preprocessor:

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

USCR Use a user-supplied comparison function for ascertaining table membership. The function should be named hcompar and should behave in a manner similar to strcmp (see string(3C)).

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

START Places new entries at the beginning of the linked list; default is placement at the end.

SORTUP Keeps the linked list sorted by key in ascending order.

SORTDOWN Keeps the linked list sorted by key in descending order.

Additionally, there are preprocessor options for obtaining debugging printout (-DDEBUG) and for including a test driver in the calling function (-DDRIVER). See the source code for further details.

WARNINGS

Both hsearch and hcreate use the malloc(3C) function to allocate space.

RETURN VALUES

The hsearch function returns a null pointer if the action is FIND and the item could not be found, or if the action is ENTER and the table is full.

If it cannot allocate sufficient space for the table, hcreate returns 0.

EXAMPLES

The following example reads in strings, followed by two numbers, and stores them in a hash table, discarding duplicates. It then reads in strings, finds the matching entry in the hash table, and prints it out.

```
#include <stdio.h>
#include <string.h>
#include <search.h>
struct info {                  /* This is the info stored in the table */
      int age, room;           /* other than the key. */
};
#define NUM_EMPL    5000        /* # of elements in search table */

main( )
{
     char string_space[NUM_EMPL*20];     /* space to store strings */
     struct info info_space[NUM_EMPL];   /* space to store employee info */
     char *str_ptr = string_space;       /* next avail space in string_space */
     struct info *info_ptr = info_space; /* next avail space in info_space */
     ENTRY item, *found_item;
     char name_to_find[30];              /* name to look for in table */
     int i = 0;
     (void) hcreate(NUM_EMPL);           /* create table */
     while (scanf("%s%d%d", str_ptr, &info_ptr->age,
            &info_ptr->room) != EOF && i++ < NUM_EMPL) {
            item.key = str_ptr;

           /* put info in structure, and structure in item */
            item.data = (void *)info_ptr;
            str_ptr += strlen(str_ptr) + 1;
            info_ptr++;
            (void) hsearch(item, ENTER);
            /* put item into table */
     }
     item.key = name_to_find;            /* access table */
     while (scanf("%s", item.key) != EOF) {
         /* if item is in the table */
         if ((found_item = hsearch(item, FIND)) != NULL) {
             (void)printf("found %s, age = %d, room = %d\n",
               found_item->key,
               ((struct info *)found_item->data)->age,
               ((struct info *)found_item->data)->room);
         } else {
                (void)printf("no such employee %s\n", name_to_find);
         }
     }
}
```

SEE ALSO

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

NAME

`ia_failure` – Processes identification and authentication (I&A) failures

SYNOPSIS

```
#include <ia.h>
#include <udb.h>
```

```
int ia_failure(
      ia_failure_t *paramsent,
      ia_success_ret_t *paramret);
```

IMPLEMENTATION

All Cray Research systems except Cray MPP systems running UNICOS MAX

DESCRIPTION

The `ia_failure` routine provides the following functionality:

- Manages the updating of the authentication failure information in the user database (UDB).
- Performs I/A failure auditing.
- Processes delayed logging; this is not done for batch jobs.

paramsent contains a pointer to the structure that contains the input parameters. *paramret* contains a pointer to the structure that contains the output parameters.

RETURN VALUES

If successful, `IA_NORMAL` is returned. Otherwise, an IA exception code is returned. This routine does not return if the exit code supplied in *paramsent* is nonzero.

NOTES

This routine supports two user exits, `ia_uex_failure` (which is called on entry to this routine) and `ia_uex_failaudit` (which is called after normal auditing is performed).

SEE ALSO

`getconfval`(3C), `ia_success`(3C), `ia_user`(3C)

`slgentry`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`exit`(3C), `time`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`confval`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014, for descriptions of `login`-related UNICOS centralized user Identification/Authentication (I/A) options

`udb`(5) for a description of the UNICOS user database file

NAME

ia_mlsuser – Determines the user's mandatory access control (MAC) attributes

SYNOPSIS

```
#include <sys/mac.h>
#include <sys/udb.h>

int ia_mlsuser(
       struct udb *ueptr,
        struct secstat *sbptr,
        struct usrv *usptr,
        mls_t *rlabptr;
        int prntactive);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The ia_mlsuser routine determines the security attributes of the session based on the attributes of the user and the connection. The attributes of the session are not set.

ueptr is a pointer to the user database (UDB) entry of the user. *sbptr* is the pointer to the attributes of the connection. *usptr* is the structure in which the attributes of the session are returned to the caller. *rlabptr* is a required active label of the session. If int prntactive is nonzero, the active label of the session is echoed.

The label range for the session is the intersection of the label range of the user and the label range of the connection. If specified, the required active label must be within the range of the session. If the required active label is null, the active label of the session is set to the default label of the user. The active label is set to the minimum label of the session if the default is not within the range of the session.

NOTES

No auditing is performed by this routine; the caller must perform auditing.

The label on the current process is not changed.

RETURN VALUES

IA_NORMAL is returned for successful completion. Otherwise, IA_MAC is returned.

SEE ALSO

`getconfval`(3C), `mls_create`(3C), `mls_free`(3C), `mls_glb`(3C), `mls_lub`(3C),

`setusrv`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`confval`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014, for descriptions of centralized user identification and authentication options

NAME

`ia_success` – Processes identification and authentication (I&A) successes

SYNOPSIS

```
#include <ia.h>
#include <udb.h>

int ia_success(
          ia_success_t *paramsent,
          ia_success_ret_t *paramret);
```

IMPLEMENTATION

All Cray Research systems except Cray MPP systems running UNICOS MAX

DESCRIPTION

The `ia_success` routine provides the following functionality:

- Manages the updating of the authentication success information in the user database (UDB).
- Performs the I/A success auditing.

paramsent contains a pointer to the structure that contains the input parameters. *paramret* contains a pointer to the structure that contains the output parameters.

RETURN VALUES

If successful, `IA_NORMAL` is returned. Otherwise, an IA exception code is returned.

NOTES

This routine supports two user exits, `ia_uex_success` (which is called on entry to this routine) and `ia_uex_succaudit` (which is called at the end of this routine).

SEE ALSO

`ia_failure`(3C), `ia_user`(3C)

`time`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`slgentry`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`udb`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

ia_user – Performs user identification and authentication (I&A)

SYNOPSIS

```
#include <ace.h>
#include <ia.h>
#include <udb.h>
```

```
int ia_user(
         ia_user_t *param,
         ia_user_ret_t *ret);
```

IMPLEMENTATION

All Cray Research systems except Cray MPP systems running UNICOS MAX

DESCRIPTION

The ia_user routine provides a common UNICOS identification and authentication mechanism. The caller specifies what authentication to perform and the order of authentication. The *pswdlist* field in the *param* structure is the list and order of authentication to be performed.

The following types of authentication are supported by this routine:

Type	**Description**
IA_DIALUP	Dialup authentication; not supported in batch.
IA_SECURID	SecurID passcode identification.
IA_UDB	User database (UDB) password identification.
IA_WAL	Workstation access list (WAL) verification.

These authentications can be requested in any order and combination. However, the order and handling of IA_SECURID and IA_UDB authentication have special rules, which are as follows:

Combination	**Description**
IA_SECURID then IA_UDB	This combination should be the first choice for authentication. UDB authentication is performed only if the SecurID acm_both flag is set, or the user is not configured for SecurID authentication.
IA_UDB only	This is the authentication choice if SecurID is bypassed.
IA_SECURID only	This is the authentication choice if UDB authentication is bypassed. Neither SecurID or UDB authentication is performed if the user is not configured with SecurID.

IA_UDB then IA_SECURID

UDB authentication is performed, then SecurID authentication is performed if the user has a SecurID account. In this case, the acm both flag has no meaning. Both authentications are always checked.

IA_SECURID can be specified regardless of whether SecurID is configured at your site. All SecurID authentication is bypassed if your site is not configured with SecurID. No warning is returned.

The caller identifies itself to this routine. The following list describes known callers; special handling is noted for several of the callers:

Caller	**Description**
IA_DGDAEMON	No special handling.
IA_FTAMD	Sets *uname* to ftp if the IA_GUEST flag is set.
IA_FTPD	Sets *uname* to ftp if the IA_GUEST flag is set.
IA_LOGIN	Supports the login back door. Only allows root to log in from the console when CONSOLE is defined.
IA_NQS	No special handling of root.
IA_REXECD	No special handling.
IA_RSHD	No special handling.
IA_SU	Supports no authentication for authorized users.

The *flags* field in the *param* structure is a bit mask. The following list describes each of the supported flags:

Flag	**Description**
IA_FFLAG	Indicates authentication can be skipped. This flag implies the same functionality as the login implementation.
IA_GUEST	Indicates anonymous ftp or ftam.
IA_IDENTIFICATION	Indicates that only identification should be performed. The private UDB is returned to authorized callers, while the public UDB is returned to unauthorized users. IA_PUBLIC is returned if the public UDB entry is returned.
IA_INTERACTIVE	Indicates an interactive session and that the user can be prompted for information. If this flag is not set, passwords must be supplied.
IA_PUBLICIDENT	Indicates that identification should be performed. The public UDB entry is returned. Does not require any other flag. If this flag is set, the private UDB is never returned. IA_IDENTIFICATION has no meaning and is not processed.
IA_RFLAG	Indicates that this is a remote I&A and clears IA_FFLAG.

NOTES

To be an authorized user you must have read access to the private UDB.

This routine does not perform auditing or update the UDB based on the status of the I&A request. The caller must perform auditing and ensure that the UDB is updated, which can be done by using the `ia_success`(3C) and `ia_failure`(3C) routines.

This routine supports three user exits, `ia_uex_authrep` (which is called on entry to this routine); `ia_uex_authadd` (which is called after the requested authentication has been performed); and `ia_uex_authend` (which is called at the end of this routine). `ia_uex_authadd` is not called if `IA_IDENTIFICATION` or `IA_PUBLICIDENT` are specified.

RETURN VALUES

The following return values are possible:

Value	**Description**
`IA_BACKDOOR`	Access allowed through the back door.
`IA_BADAUTH`	Unknown authorization type.
`IA_DIALUPERR`	An error was encountered when processing the dial-up authentication.
`IA_DISABLED`	The account is disabled; disabled flag is set in the UDB.
`IA_GETSYSV`	The `getsysv`(2) system call failed.
`IA_LOCALHOST`	Access from `localhost` not allowed.
`IA_MAXLOGS`	Access denied; maximum failures on account reached.
`IA_NOPASS`	User allowed to bypass authentication.
`IA_NORMAL`	Normal return code.
`IA_PUBLIC`	The user was identified and the public UDB entry was returned. Only returned if `IA_IDENTIFICATION` is set and `IA_PUBLICIDENT` is not set.
`IA_SECURIDERR`	An error was encountered when processing SecurID authentication.
`IA_TRUSTED`	Trusted user not allowed.
`IA_UDBERR`	An error was encountered when processing UDB authentication.
`IA_UDBEXPIRED`	Authentication successful; however the UDB password has expired. Caller must process expired passwords.
`IA_UDBPWDNULL`	The password in the UDB is null, and the user is not configured for SecurID authentication.
`IA_UDBWEEK`	The password expires within the week.
`IA_UNKNOWN`	Identification error; unknown user.

`IA_UNKNOWNYP`	User known in UDB, but configured for network information services (NIS) and not known to NIS.
`IA_WALERR`	The workstation access list (WAL) denied access.

EXAMPLES

The following examples show how to use the `ia_user` routine.

Example 1: This example shows how `ia_user` can be called to identify a user. This example returns the public UDB entry.

```
ia_user_ret_t  uret;    /* Parameters returned from ia_user. */
ia_user_t  usent;       /* Parameters sent to ia_user.*/
struct  udb ue;

/*
 * Set up request structure.
 */
usent.revision = 0;
usent.uname = u_name;   /* May be null for interactive*/
usent.host = NULL;
usent.ttyn = ttyn;
usent.caller = IA_LOGIN;
usent.pswdlist = NULL;
usent.ueptr = &ue;

/*
 * Initialize the return structure.
 */
uret.revision = 0;
uret.pswd = NULL;
uret.normal = 0;

/*
 * Set flag requesting public udb entry.
 */
usent.flags = IA_PUBLICIDENT;

retcode = ia_user(&usent, &uret);

if (retcode == IA_NORMAL) /* User identified, public UDB entry returned.*/
else
                          /* User identified failed, UDB entry NOT returned. */
```

Example 2: This example shows how `ia_user` can be called to identify a user. This example returns either the public or private UDB entry.

```
ia_user_ret_t  uret;    /* Parameters returned from ia_user. */
ia_user_t  usent;       /* Parameters sent to ia_user.*/
struct  udb ue;

/*
 * Set up request structure.
 */
usent.revision = 0;
usent.uname = u_name;   /* May be null for interactive*/
usent.host = NULL;
usent.ttyn = ttyn;
usent.caller = IA_LOGIN;
usent.pswdlist = NULL;
usent.ueptr = &ue;

/*
 * Initialize the return structure.
 */
uret.revision = 0;
uret.pswd = NULL;
uret.normal = 0;

/*
 * Set flag requesting identification only.
 */
usent.flags = IA_IDENTIFICATION;

retcode = ia_user(&usent, &uret);

if (retcode == IA_NORMAL) /* User identified, private UDB entry returned.*/
else if (retcode == IA_PUBLIC)
                          /* User identified, public UDB entry returned.*/
else
                          /* User identified failed, UDB entry NOT returned. */
```

Example 3: This example shows how to call `ia_user` and bypass authentication. In this example, the `ia_uex_authadd` user exit is still called. The difference between this example and example 2 is that disabled account, password expiration, and so on are all processed.

```
    ia_user_ret_t  uret;    /* Parameters returned from ia_user. */
    ia_user_t  usent;       /* Parameters sent to ia_user.*/
    struct  udb ue;

    /*
     * Set up request structure.
     */
    usent.revision = 0;
    usent.uname = u_name;  /* May be null for interactive */
    usent.host = utmp.ut_host;
    usent.ttyn = ttyn;
    usent.caller = IA_SU;
    usent.pswdlist = NULL;
    usent.ueptr = &ue;

    /*
     * Set interactive flag.  This indicates that
     * the user can be prompt for name and password.
     */
    usent.flags = IA_INTERACTIVE;

    /*
     * Initialize the return structure.
     */
    uret.revision = 0;
    uret.pswd = NULL;
    uret.normal = 0;

    retcode = ia_user(&usent, &uret);
```

Example 4: This example shows how `login`(1) can be used to call `ia_user`. In this example, `login` calls `ia_user` to perform SecurID, UDB, DIALUP, and WAL access verification. The verification is performed in this order because of the order of the linked list.

```
ia_user_ret_t  uret;    /* Parameters returned from ia_user. */
ia_user_t  usent;       /* Parameters sent to ia_user.*/
passwd_t pwdacm,        /* Verification elements.     */
         pwddialup,
         pwdudb,
         pwdwal;
struct  udb ue;

/*
 * Set up the verification list.  The order of the list
 * is the order verification will be performed.
 */
pwdacm.atype = IA_SECURID;
pwdacm.pwdp = NULL;
pwdacm.next = &pwdudb;

pwdudb.atype = IA_UDB;
pwdudb.pwdp = NULL;
pwdudb.next = &pwddialup;

pwddialup.atype = IA_DIALUP;
pwddialup.pwdp = NULL;
pwddialup.next = &pwdwal;

pwdwal.atype = IA_WAL;
pwdwal.pwdp = NULL;
pwdwal.next = NULL;

/*
 * Set up request structure.
 */
usent.revision = 0;
usent.uname = u_name;  /* May be null for interactive */
usent.host = utmp.ut_host;
usent.ttyn = ttyn;
usent.caller = IA_LOGIN;
usent.pswdlist = &pwdacm;
usent.ueptr = &ue;

/*
 * Set interactive flag.  This indicates that
 * the user can be prompt for name and password.
 */
usent.flags = IA_INTERACTIVE;

/*
 * Initialize the return structure.
 */
```

```
uret.revision = 0;
uret.pswd = NULL;
uret.normal = 0;

retcode = ia_user(&usent, &uret);
```

SEE ALSO

ia_failure(3C), ia_mlsuser(3C), ia_success(3C) slgentry(2)

time(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

udb(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

`iconv`, `iconv_close`, `iconv_open` – Code conversion function

SYNOPSIS

`#include <iconv.h>`

`size_t iconv (iconv_t` *cd*`, const char **`*inbuf*`, size_t *`*inbytesleft*`, char **`*outbuf*`, size_t *`*outbytesleft*`);`

`iconv_t iconv_open (const char *`*tocode*`, const char *`*fromcode*`);`

`int iconv_close (iconv_t` *cd*`);`

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The `iconv` function converts the sequence of characters from one codeset, in the *inbuf* array, into a sequence of corresponding characters in another codeset in the *outbuf* array. Codesets are specified in the `iconv_open` call that returned the conversion descriptor, *cd*. The *inbuf* argument points to a variable that points to the first character in the input buffer; *inbytesleft* indicates the number of bytes to the end of the buffer to be converted. The *outbuf* argument points to a variable that points to the first available byte in the output buffer, and *outbytesleft* indicates the number of the available bytes to the end of the buffer.

For state-dependent encodings, the conversion descriptor *cd* is placed into its initial shift state by a call for which *inbuf* is a null pointer, or for which *inbuf* points to a null pointer. When `iconv` is called in this way, and if *outbuf* is not a null pointer or a pointer to a null pointer, and *outbytesleft* points to a positive value, `iconv` places, into the output buffer, the byte sequence to change the output buffer to its initial shift state. If the output buffer is too small to hold the entire reset sequence, `iconv` fails and sets `errno` to `[E2BIG]`. Subsequent calls with *inbuf* as other than a null pointer or a pointer to a null pointer cause the conversion to occur from the conversion descriptor’s current state.

If a sequence of input bytes forms no valid character in the specified codeset, conversion stops after the previous successfully converted character. If the input buffer ends with an incomplete character or shift sequence, conversion stops after the previous successfully converted bytes. If the output buffer is too small to hold the entire converted input, conversion stops just before the input bytes that would cause the output buffer to overflow. The variable to which *inbuf* points is updated to point to the byte following the last byte successfully used in the conversion. The value to which *inbytesleft* points is decremented to reflect the number of still-unconverted bytes in the input buffer. The variable to which *outbuf* points is updated to point to the byte following the last byte of converted output data. The value to which *outbytesleft* points is decremented to reflect the number of bytes still available in the output buffer. For state-dependent encodings, the conversion descriptor is updated to reflect the shift state in effect at the end of the last successfully converted byte sequence.

If `iconv` encounters a character in the input buffer that is legal, but for which an identical character does not exist in the target codeset, `iconv` performs an implementation-defined conversion on this character.

The `iconv_open` function returns a conversion descriptor that describes a conversion from the codeset specified by the string to which the *fromcode* argument points to the codeset specified by the string to which the *tocode* argument points. For state-dependent encodings, the conversion descriptor is in a codeset-dependent initial shift state, ready for immediate use with the `iconv` function.

Settings of *fromcode* and *tocode* and their permitted combinations depend on the implementation. A conversion descriptor remains valid in a process until that process closes it.

The `iconv_close` function deallocates the conversion descriptor *cd* and all other associated resources allocated by the `iconv_open` function. If a file descriptor is used to implement the type `iconv_t`, that file descriptor is closed.

RETURN VALUES

The `iconv` function updates the variables to which the arguments point to reflect the extent of the conversion and returns the number of nonidentical conversions performed. If the entire string in the input buffer is converted, the value to *inbytesleft* points is 0. If the input conversion is stopped due to any of the preceding conditions, the value to which *inbytesleft* points is nonzero and `errno` is set to indicate the condition. If an error occurs, `iconv` returns `(size_t)-1` and sets `errno` to indicate the error.

If successful, the `iconv_open` function returns a conversion descriptor for use on subsequent calls to `iconv`; otherwise, `iconv_open` returns `(iconv_t)-1` and sets `errno` to indicate the error.

If successful, the `iconv_close` function returns 0; otherwise, it returns –1 and sets `errno`.

MESSAGES

The `iconv` function fails if any of the following errors occur:

`[EILSEQ]` Input conversion stopped due to an input byte that does not belong to the input codeset.

`[E2BIG]` Input conversion stopped due to lack of space in the output buffer.

`[EINVAL]` Input conversion stopped due to an incomplete character or shift sequence at the end of the input buffer.

The `iconv` function may fail if the following occurs:

`[EBADF]` The *cd* argument is an open conversion descriptor that is not valid.

The `iconv_open` function may fail if any of the following errors occur:

`[EMFILE]` The *cd* argument is not a valid open conversion descriptor.

`[ENFILE]` Input conversion stopped due to an input byte that does not belong to the input codeset.

`[ENOMEM]` Input conversion stopped due to lack of space in the output buffer.

`[EINVAL]` Input conversion stopped due to an incomplete character or shift sequence at the end of the input buffer.

The `iconv_close` function may fail if the following occurs:

`[EBADF]` The conversion descriptor is not valid.

SEE ALSO

`locale`(3C), `locale.h`(3C), `localeconv`(3C), `setlocale`(3C)

NAME

uid2nam, gid2nam, acid2nam, nam2uid, nam2gid, nam2acid, gidnamfree, acidnamfree – Maps IDs to names

SYNOPSIS

```
#include <stdlib.h>
char *uid2nam (int uid);
char *gid2nam (int gid*C);
char *acid2nam (int acid);
int nam2uid (char *uname);
int nam2gid (char *gname);
int nam2acid (char *aname);
void gidnamfree (void);
void acidnamfree (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The uid2nam function maps a numerical user ID to a character string; nam2uid maps a character string to a numerical user ID.

The gid2nam function maps a numerical group ID to a character string; nam2gid maps a character string to a numerical group ID.

The acid2nam function maps a numerical account ID to a character string; nam2acid maps a character string to a numerical account ID.

The acidnamfree and gidnamfree functions update the mapping information from the map files.

These functions provide fast mapping of numerical IDs to names, and vice versa, in the UNICOS user-information database. (See newacct(1) and udb(5).)

The acid functions copy the corresponding map files into main memory upon the first call and use either a binary search algorithm (for ID-to-name translations) or a linear search algorithm (for name-to-ID translation); the gid functions call getgrgid and getgrnam (see getgrent(3C)).

If an application depends on the most recent data in the map files and has run for a considerable amount of time, the gidnamfree and acidnamfree functions may be used to force the translation functions to update the memory copy of the map files.

WARNINGS

The nam2uid and uid2nam routines leave the udb file open to assure reasonable performance for multiple calls. If it is important that the program in which the calls are made can be restarted, call endpwent or endudb to close the udb file after the access is complete.

The nam2acid and acid2nam routines leave the account ID file open for the same reason. If it is important that the program in which the calls are made can be restarted, call acidnamfree to close the udb file after the access is complete.

The nam2gid and gid2nam functions close the group file before returning.

RETURN VALUES

If no match is found, acid2nam, uid2nam, and gid2nam return a null pointer. The nam2uid, nam2gid, and nam2acid functions all return –1 if no match is found for the name.

FILES

/etc/acid

/etc/group

/etc/udb.public

SEE ALSO

getpwent(3C), getgrent(3C), libudb(3C)

newacct(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

udbgen(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

acctid(2) in *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

udb(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

ieee_float – Introduction to the IEEE floating-point environment

IMPLEMENTATION

Cray MPP systems
CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

ANSI/IEEE Std 754-1985
X3/TR-17:199x

DESCRIPTION

The man pages in this section describe the header files, types, macros, and functions developed to support the Cray Research implementation of the IEEE floating-point standard in the Cray C and C++ compilers. The corresponding Fortran routines are described in the *Fortran Language Reference Manual, Volume 2*, Cray Research publication SR–3903.

ASSOCIATED HEADERS

<fp.h>
<fenv.h>

ASSOCIATED TYPES

fenv.h Types

Type	Description
fenv_t	Represents the entire floating-point environment
fexcept_t	Represents the floating-point exception flags
fetrap_t	Represents the floating-point trap flags

ASSOCIATED MACROS

fp.h Macros

Macro	Description
HUGE_VAL, HUGE_VALF, HUGE_VALL	Expand to positive infinity
INFINITY	Expands to positive infinity
NAN	Expands to a quiet NaN
FP_NAN, FP_INFINITE, FP_NORMAL, FP_SUBNORMAL, FP_ZERO	Represent the mutually exclusive kinds of floating-point values
DECIMAL_DIG	Represents the digits supported by conversion to internal floating-point formats

`fpclassify`(3C)	Returns the macro (`FP_NAN`, and so on) that identifies its argument
`isfinite`	Determines if its argument value is finite
`isnan`	Determines if its argument value is a NaN
`isnormal`	Determines if its argument value is normal
`signbit`(3C)	Determines if its argument value is negative
`isgreater`(3C)	Determines if its first argument is greater than its second
`isgreaterequal`	Determines if its first argument is greater than or equal to its second
`isless`	Determines if its first argument is less than its second
`islessequal`	Determines if its first argument is less than or equal to its second
`islessgreater`	Determines if its first argument is less than or greater than its second
`isunordered`	Determines if its arguments compare unordered

`fenv.h` Macros

Macro	**Description**
`FE_INEXACT`	Represents the inexact exception flag
`FE_DIVBYZERO`	Represents the divide-by-zero exception flag
`FE_UNDERFLOW`	Represents the underflow exception flag
`FE_OVERFLOW`	Represents the overflow exception flag
`FE_INVALID`	Represents the invalid exception flag
`FE_EXCEPTINPUT`	Represents the exceptional input exception flag
`FE_ALL_EXCEPT`	Represents the bitwise OR of all exception macros
`FE_TRAP_INVALID`	Represents the invalid operation trap flag
`FE_TRAP_DIVBYZERO`	Represents the divide-by-zero trap flag
`FE_TRAP_OVERFLOW`	Represents the overflow trap flag
`FE_TRAP_UNDERFLOW`	Represents the underflow trap flag
`FE_TRAP_INEXACT`	Represents the inexact trap flag
`FE_ALL_TRAPS`	Represents all of the trap flags
`FE_TONEAREST`	Round toward nearest
`FE_UPWARD`	Round toward positive infinity
`FE_DOWNWARD`	Round toward negative infinity
`FE_TOWARDZERO`	Round toward zero
`FE_DFL_ENV`	Represents the default floating-point environment

ASSOCIATED FUNCTIONS

`fp.h` Functions

Function	**Description**
`logb`(3C), `logbf`, `logbl`	Return the signed exponent of their arguments
`scalb`(3C), `scalbf`, `scalbl`	Compute x * `FLT_RADIX`^{n} efficiently

`rint`(3C), `rintf`, `rintl`	Round arguments to an integral value in floating-point format
`rinttol`(3C)	Rounds a floating-point number to a long integer value
`remainder`(3C), `remainderf`, `remainderl`	Divide their arguments and return the remainder
`copysign`(3C), `copysignf`, `copysignl`	Assign the sign of the second argument to the value of the first argument
`nextafter`(3C), `nextafterf`, `nextafterl`	Return the next value in the direction of the second argument

`fenv.h` Functions

Function	**Description**
`feclearexcept`(3C)	Clears exception flags
`fegetexceptflag`	Stores the representation of the exception flags
`feraiseexcept`	Raises exceptions
`fesetexceptflag`	Restores the representation of the exception flags
`fetestexcept`	Determines which exception flags are currently set
`fesetround`(3C)	Establishes the rounding direction
`fegetround`	Gets the current rounding direction
`fegetenv`(3C)	Stores the current floating-point environment
`feholdexcept`	Saves the environment, clears exception flags, and disables traps
`fesetenv`	Establishes the floating-point environment
`feupdateenv`	Saves the current exceptions, installs a new environment, and raises the saved exceptions
`fedisabletrap`(3C)	Disables traps
`feenabletrap`	Enables traps
`fegettrapflag`	Stores the representation of the trap flags
`fesettrapflag`	Restores the representation of the trap flags
`fetesttrap`	Determines which traps are currently enabled

SEE ALSO

Migrating to the CRAY T90 Series IEEE Floating Point, Cray Research publication SN–2194

NAME

`IHPSTAT` – Returns statistics about the heap

SYNOPSIS

value=`IHPSTAT(`*code*`)`

IMPLEMENTATION

UNICOS and UNICOS/mk systems

DESCRIPTION

`IHPSTAT` returns statistics about the heap.

When using the CF90 compiler on UNICOS or UNICOS/mk systems, all arguments must be of default kind unless documented otherwise. On UNICOS and UNICOS/mk, the default kind is `KIND=8` for integer, real, complex, and logical arguments.

The following is a list of valid arguments for this routine.

value Requested information.

code Code for the type of information requested, as follows:

Code	**Meaning**
1	Current heap length
4	Number of allocated blocks
10	Size of the largest free block
11	Amount by which the heap can shrink
12	Amount by which the heap can grow
13	First word address of the heap; on UNICOS/mk systems, byte addresses are returned.
14	Last word address of the heap; on UNICOS/mk systems, byte addresses are returned.
22	Amount by which the shared heap can grow.

All values returned by `IHPSTAT` are in words.

SEE ALSO

`HPALLOC`(3F), `HPCHECK`(3F), `HPCLMOVE`(3F), `HPDEALLC`(3F), `HPDUMP`(3F), `HPNEWLEN`(3F), `HPSHRINK`(3F), `IHPLEN`(3F), `IHPVALID`(3F)

NAME

`index`, `rindex` – Locates characters in string

SYNOPSIS

```
#include <string.h>
char *index (const char *s, int c);
char *rindex (const char *s, int c);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The `index` function returns a pointer to the first occurrence of character *c* in string *s*, or null if *c* does not occur in the string.

The `rindex` function returns a pointer to the last occurrence of character *c* in string *s*, or null if *c* does not occur in the string.

These functions operate on null-terminated strings.

NOTES

Functions `strchr` and `strrchr` (see `string`(3C)) are the same as `index` and `rindex`, respectively, and they should be used in all new codes. Functions `index` and `rindex` are provided only for compatibility with other BSD codes.

SEE ALSO

`string`(3C)

NAME

inet_addr, inet_lnaof, inet_makeaddr, inet_netof, inet_network, inet_ntoa, inet_subnetof, inet_subnetmaskof – Manipulates Internet address

SYNOPSIS

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

unsigned long inet_addr (char *cp);

int inet_lnaof (struct in_addr in);

struct in_addr inet_makeaddr (int net, int host);

int inet_netof (struct in_addr in);

unsigned long inet_network (char *cp);

char *inet_ntoa (struct in_addr in);

unsigned long inet_subnetof (struct in_addr in);

unsigned long inet_subnetmaskof (struct in_addr in);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The `inet_network` and `inet_addr` functions each interpret character strings representing numbers expressed in the Internet standard "." notation (dot notation), returning numbers suitable for use as Internet network numbers and Internet addresses, respectively.

The `inet_netof` and `inet_lnaof` functions break apart Internet host addresses, returning the network number and local network address part, respectively.

The `inet_makeaddr` function takes an Internet network number and the host number and constructs an Internet address from them.

The `inet_ntoa` function takes an Internet address and returns an ASCII string representing the address in "." notation.

The `inet_subnetof` and `inet_subnetmaskof` functions return the subnet and subnet mask, respectively, of the Internet address *in*. These functions determine the actual subnet mask by consulting the configured subnet masks of the active network interfaces on the system the first time either function is called. This information is cached, and later calls to either function consult the configured network interfaces again only when a search of the accumulated information fails to match the network portion of *in*, and only for those interfaces whose associated addresses or flags have changed (for example, to detect a newly configured interface).

All Internet addresses are returned in network byte order, except for single port addresses. All network numbers and local address parts are returned as machine-format integer values.

INTERNET ADDRESSES

Values specified using the Internet `"."` notation take one of the following forms:

a.b.c.d
a.b.c
a.b
a

When a four-part address is specified, each part is interpreted as a byte of data and is assigned, from left to right, to the 4 bytes of an Internet address.

When a three-part address is specified, the last part is interpreted as a 16-bit quantity and is placed in the rightmost 2 bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as *net.net.host*.

When a two-part address is supplied, the last part is interpreted as a 24-bit quantity and is placed in the rightmost 3 bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as *net.host*.

When only a one-part address is specified, the value is stored directly in the network address without any byte rearrangement, in host byte order.

All numbers supplied as parts in a `"."` notation address may be in decimal, octal, or hexadecimal format, as specified in the C language (that is, a leading 0x or 0X implies hexadecimal, and a leading 0 implies octal; otherwise, the number is interpreted as decimal).

NOTES

The problem of host byte ordering versus network byte ordering is confusing.

A simple way to specify Class C network addresses in a manner similar to that used for specifying Class B and Class A addresses is needed.

The string returned by `inet_ntoa` resides in a static memory area that must be copied if it is to be used.

For inet_subnetof and inet_subnetmaskof, checking the cached information first means that they might use or return an old, incorrect subnet mask if an interface is configured down and configured back up with the same address, but a different subnet mask, between calls to either function. In practice, this should rarely happen.

Relying on active network interfaces for subnet mask information means that inet_subnetof and inet_subnetmaskof are useless without networking facilities (for example, in single-user mode). Similarly, neither function can be of any help for networks that are not directly connected to the system (for example, for networks that are not directly connected, a return value of ˜0L means "I don’t know if this is a subnet," not "this is definitely not a subnet").

RETURN VALUES

The inet_addr and inet_network functions return a value of ˜0L for incorrect requests.

inet_subnetof and inet_subnetmaskof return the value ˜0L if the network portion of the address cannot be matched with a configured interface, and 0 for addresses whose network portions are matched with an interface that has no subnet mask. Both functions set errno to EINVAL if the system has more interfaces than they can support.

FILES

/usr/include/arpa/inet.h

/usr/include/netinet/in.h

/usr/include/sys/socket.h

/usr/include/sys/types.h

SEE ALSO

gethost(3C), getnet(3C)

hosts(5), networks(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

`initgroups` – Initializes group access list

SYNOPSIS

```
#include <grp.h>
int initgroups (char *name, int basegid);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The `initgroups` function reads through the `/etc/group` file and uses the `setgroups`(2) call to set up the group access list for the user specified by *name*. The *basegid* is automatically included in the groups list. Typically, this value is given as the group number from the user database.

NOTES

The `initgroups` function uses the functions based on `getgrent`(3C). If the invoking program uses any of these functions, the `group` structure is overwritten in the call to `initgroups`.

FILES

`/etc/group`

RETURN VALUES

If it was not invoked by the super user, `initgroups` returns –1.

SEE ALSO

`getgrent`(3C), `getpwent`(3C)

`setgroups`(2) in *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`groups`(1B), `udb`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

`inter_lang` – Introduction to interlanguage communications functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The interlanguage communications functions provide various means for passing information between functions written in C and functions written in Fortran, Pascal, or Cray Assembly Language (CAL).

These languages use different calling sequences and have different representation for some data types; as explained in this entry, these differences must be understood and the correct conventions must be followed to ensure correct interlanguage communication. The following subsections describe these differences and conventions.

Calling Conventions

The Cray Standard C compiler running in extended mode, and all Cray Research Fortran compilers, generate code that uses the call-by-register convention for math library functions (it is the fastest calling sequence). These functions, called VFUNCTIONS, follow the call-by-register naming convention, which requires that the name be of the form `NAME%` or `%NAME%`. The compilers automatically translate the math function names to the VFUNCTION names. For further information about VFUNCTIONS, see the *Cray Standard C Reference Manual*, Cray Research publication SR–2074.

The Cray Standard C compiler running in strict conformance mode generates code that uses the call-by-value math functions. These functions perform argument domain and range checking. The names of these functions do not get translated.

In a Fortran program, if a math library function is declared `EXTERNAL` or `INTRINSIC`, the Fortran compilers generate code that uses the call-by-address math functions. The names of these functions do not get translated.

Character Pointers/Character Descriptors

The C language does not explicitly support a character string type, but by convention, C character pointers typically point to character arrays that are terminated with a 0 byte, and several functions in the C library process such strings (for more information, see `character`(3C)). A C character pointer, like a Fortran character descriptor, contains a character location. Unlike a Fortran character descriptor, a C character pointer does not contain a length. A C character pointer cannot be passed to a Fortran function or subroutine that expects a character argument. The format of the C character pointer is not compatible with the format of a Fortran character descriptor.

A Fortran character variable has a length associated with it that tells the number of characters in a variable. Generally, character strings used in Fortran programs are stored in character variables, rather than in arrays of single characters. A character argument can have an actual argument that is a substring or an array element that does not begin on a word boundary, so that the address of a Fortran character argument has both a word address and a bit offset. A character argument can be declared as `CHARACTER *(*)`, which means that the length of the argument is not known at compile time and must be passed to the subprogram at execution time. A Fortran character descriptor that contains the first character location and the length of a single entity (scalar or array element) is always passed for a character argument.

The interlanguage convention for passing character strings is through the use of Fortran character descriptors. Although this is automatic from Fortran, you, as a C user, must use the functions described in header file `fortran.h` to do the necessary conversions.

C Boolean Data versus Fortran Logical

C users must use functions provided as CRI extensions to pass C Boolean values as Fortran logical values. The interlanguage convention for the representation of logical values is that of Fortran type `LOGICAL`.

Calling C Functions from Fortran Functions

The Fortran language is case-insensitive; therefore, the CRI Fortran compilers map all code into uppercase. This means that functions that have lowercase names cannot be called from Fortran programs.

Calling Fortran Functions from C Functions

The C language is case-sensitive, so you must use the exact case specified in the documentation when coding references to a function.

All of the functions documented in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165, and in the *Scientific Libraries Reference Manual*, Cray Research publication SR–2081, are callable from C programs. If the manual entry for the function does not explicitly provide the C synopsis, the following rules can be used:

- Because the function is not declared in a C header, explicitly declare the function as external and specify the type of the return value.
- When calling the function, pass the address of the arguments by using the address operator (&) for each argument, or by using a pointer to the argument for the argument. Array names are considered to be addresses; therefore, the address operator is not needed when using them.
- The value returned by the function is the value, not the address of the value.
- Specify the `fortran` keyword. The `fortran` keyword is a CRI extension to the C language and is useful when a C program calls a function following the Cray Fortran calling sequence. Specifying the `fortran` keyword causes the C compiler to verify that the arguments used in each call to the function are pass-by-address. For more information on the `fortran` keyword, see the *Cray Standard C Reference Manual*, Cray Research publication SR–2074.

Calling C Functions from CAL Functions

External references from CAL are case-sensitive, so you must use the exact name for functions as specified in the appropriate manual. Cray Research supports two standard calling methods for math library functions: call-by-register and call-by-address. Cray Research supports only the call-by-address method for the scientific library. For more information on the details of calling sequences, see the documentation for the `CALL` macro for the machine you are using. (See the *UNICOS Macros and Opdefs Reference Manual*, Cray Research publication SR–2403.)

You should use the `CALL` macros to do function linkage. Avoid direct user calls to functions that use the return-jump instruction.

Scalar functions return the result in registers S1 (and S2 if needed). Vector functions return their result in registers V1 (and V2 if needed). The contents of the vector-length register (VL) upon entry determine the number of elements computed for vector functions.

Calling CAL Functions from C Functions

The following example shows a C program that calls the CAL `ALOG` function:

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

extern double ALOG(double*);

main()
{
        double x, y;

        x = 1.2345;
        y = ALOG(&x);         /* use call-by-address to pass argument */
        printf("ALOG(%f) = %f\n", x, y);
}
```

The output from the execution of this program is as follows:

```
ALOG(1.234500) = 0.210666
```

It is also possible to call library functions from the scientific library, as shown in the following example:

```
#include <math.h>
#include <stdio.h>
                    n x ix y iy

extern double SDOT(int *, double [ ], int *, double [ ], int*);

main()
{
        double sx[ ] = {1.0,2.0,3.0,4.0,5.0,6.0};
        double sy[ ] = {1.0,2.0,3.0,4.0,5.0,6.0};
        double answer;
        int  n, incx, incy;

        n     = 6;      /* number of elements in array */
        incx  = 1;      /* increment between elements in words */
        incy  = 1;

        /* Note that arrays are already passed-by-address,
           but other arguments are passed by value */

        answer = SDOT(&n, sx, &incx, sy, &incy);
        printf("Dot product of x and y is %f\n", answer);
}
```

To execute this program (in source file `b.c`), enter the following commands. To get access to the `SDOT` function, you must link to the scientific library.

```
$ cc -lsci b.c
$ a.out
```

The output is as follows:

```
Dot product of x and y is 91.000000
```

Naming Conventions

Most of the Cray Research math library functions adhere to the following naming conventions:

`NAME`	Entry for scalar call-by-address
`%NAME`	Entry for vector call-by-address
`NAME%`	Entry for scalar call-by-register
`%NAME%`	Entry for vector call-by-register

CAL does not support generic function names or automatic data type conversion. For example, no math library `LOG` function exists for the logarithm function. The user must specify either `ALOG`, DLOG, or `CLOG` for real, double-precision, or complex logarithm, respectively, and the argument must be of the correct type (real, double precision, or complex).

Associated Headers

`<fortran.h>`

Associated Functions

Function	**Description**
`_btol`	Converts a 0 to a Fortran logical `.FALSE.` and a nonzero value to a Fortran logical .TRUE. (see `_ltob`)
`_cptofcd`	Converts a C character pointer to a Fortran character descriptor
`_fcdtocp`	Converts a Fortran character descriptor to a C character pointer (see `_cptofcd`)
`_fcdlen`	Extracts the byte length from the Fortran character descriptor (see `_cptofcd`)

`_ltob`	Converts a Fortran logical `.FALSE.` to a `0` and a Fortran logical .TRUE. to a `1` (see `_cptofcd`)
`_isfcd`	Determines whether a generic pointer is a Fortran character descriptor

SEE ALSO

Cray Standard C Reference Manual, Cray Research publication SR–2074

Application Programmer's Library Reference Manual, Cray Research publication SR–2165

UNICOS Macros and Opdefs Reference Manual, Cray Research publication SR–2403

Interlanguage Programming Conventions, Cray Research publication SN–3009

CF77 Commands and Directives, Cray Research publication SR–3771

NAME

`i_o` – Introduction to input/output functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The input/output functions provide various means for getting data into an executing program and for sending data out of an executing program. Some functions perform input or output on streams, and some perform input or output on files. The following subsections describe these two fundamental concepts.

Streams

Input and output, whether to or from physical devices such as terminals and tape drives, or to or from files supported on structured storage devices, are mapped into logical data *streams*, whose properties are more uniform than their various inputs and outputs. UNICOS supports two forms of mapping, for *text streams* and for *binary streams*. Under the Cray Research operating system UNICOS, text streams and binary streams are implemented identically. This may not be true for other implementations.

A *text stream* is an ordered sequence of characters composed into *lines*, each line consisting of 0 or more characters plus a terminating newline character. Characters may have to be added, altered, or deleted on input and output by library functions to conform to differing conventions for representing text in the host environment. Thus, a one-to-one correspondence may not exist between the characters in a stream and those in the external representation. Data read in from a text stream compares equal to the data that was earlier written out to that stream only if the following conditions are met:

- The data consists only of printable characters, and the control characters consist only of horizontal tabs and newline characters.
- No newline character is immediately preceded by space characters.
- The last character is a newline character.

An implementation defines whether space characters that are written out immediately before a newline character appear when the data is read in. On all Cray Research systems, space characters that are written out immediately before a newline character do appear when read.

A *binary stream* is an ordered sequence of characters that can transparently record internal data. Data read in from a binary stream compares equal to the data that was written earlier out to that stream, under the same implementation. However, such a stream may have an implementation-defined number of null characters appended to the end of the stream on some systems. Under the Cray Research operating system UNICOS, no null characters are appended.

Files

A stream is associated with an external file (or physical device) by *opening* a file or *creating* a new file if no file exists. Creating an existing file causes its former contents to be discarded if necessary. If a file can support positioning requests, a *file position indicator* associated with the stream is positioned at the start (character number zero) of the file; for example, a disk file supports positioning requests, but a terminal does not. If, however, a file that supports positioning is opened with append mode, it is implementation-defined whether the file position indicator is initially positioned at the beginning or the end of the file. On Cray Research systems, the file position indicator is maintained by subsequent reads, writes, and positioning requests ensuring an orderly progression through the file.

Usage

Binary files are not truncated, except as defined in function `fopen`. Whether a write on a text stream causes the associated file to be truncated beyond that point is implementation-defined. On Cray Research systems, the associated file is not truncated.

When a stream is *unbuffered,* characters are intended to appear from the source or at the destination as soon as possible; otherwise, characters may be accumulated and transmitted to or from the host environment as a block. When a stream is *fully buffered*, characters are intended to be transmitted to or from the host environment as a block when a buffer is filled. When a stream is *line-buffered*, characters are intended to be transmitted to or from the host environment as a block when a newline character is encountered. Furthermore, characters are intended to be transmitted as a block to the host environment when a buffer is filled, when input is requested on an unbuffered stream, or when input is requested on a line-buffered stream that requires the transmission of characters from the host environment. Support for these characteristics is implementation-defined and can be affected by use of the `setbuf` and `setvbuf` functions.

To disassociate a file from a controlling stream, *close* the file. Output streams are flushed (any unwritten buffer contents are transmitted to the host environment) before the stream is disassociated from the file. The value of a pointer to a `FILE` object is indeterminate after the associated file is closed (including the standard text streams). Whether a file of 0 length (on which no characters have been written by an output stream) actually exists is implementation-defined. On Cray Research systems, the file exists.

The file may be subsequently reopened, by the same or another program execution, and its contents reclaimed or modified (if it can be repositioned at its start). If the `main` function returns to its original caller, or if the `exit` function is called, all open files are closed and all output streams are flushed before program termination. Other paths to program termination, such as calling the `abort` function, need not close all files properly.

The address of the `FILE` object used to control a stream may be significant; a copy of a `FILE` object may not necessarily serve in place of the original.

At program startup, three text streams are predefined and need not be opened explicitly: *standard input* (for reading conventional input), *standard output* (for writing conventional output), and *standard error* (for writing diagnostic output). When opened, the standard error stream is not fully buffered; the standard input and standard output streams are fully buffered if and only if the stream can be determined not to refer to an interactive device.

Functions that open additional (nontemporary) files require a *file name*, which is a string. The rules for composing valid file names are implementation-defined. Whether the same file can be simultaneously open multiple times is also implementation-defined. For Cray Research systems, file names can consist of letters, numbers, periods, and the underscore symbol and the same file can be open multiple times simultaneously.

Associated Headers

`<ffio.h>`	File for flexible file I/O (FFIO) functions
`<stdio.h>`	File for input and output functions

Associated Functions

The I_O(3C) function performs the following associated functions:

Character I/O Functions

Function	Description
`fgetc`	Gets a character from a stream (see `getc`)
`fgets`	Gets a string from a stream (see `gets`)
`fputc`	Puts a character on a stream (see `putc`)
`fputs`	Puts a string on a stream (see `puts`)
`getc`	Gets a character from a stream
`getchar`	Gets a character from a stream (see `getc`)
`gets`	Gets a string from a stream
`putc`	Puts a character on a stream
`putchar`	Puts a character on a stream (see `putc`)
`puts`	Puts a string on a stream
`ungetc`	Pushes a character back into the input stream

Direct I/O Functions

Function	Description
`fread`	Reads input
`fwrite`	Writes output (see `fread`)
`getw`	Gets word from stream (see `getc`)
`putw`	Puts a word on a stream (see `putc`)

File Access Functions

Function	Description
`dup2`	Duplicates an open file descriptor
`fclose`	Closes a stream
`fdopen`	Associates stream with file descriptor (see `fopen`)
`fflush`	Flushes a stream (see `fclose`)
`fopen`	Opens a stream
`freopen`	Substitutes named file for stream (see `fopen`)
`getdtablesize`	Gets descriptor table size
`pclose`	Closes a pipe to a process (see `popen`)
`popen`	Initiates a pipe to a process

`setbuf`	Assigns buffering to a stream
`setvbuf`	Assigns buffering to a stream (see `setbuf`)

File Error Handling Functions

Function	**Description**
`clearerr`	Clears error and EOF indicators (see `ferror`)
`feof`	Tests EOF indicator (see `ferror`)
`ferror`	Tests error indicator

File Positioning Functions

Function	**Description**
`fgetpos`	Stores the value of the file position indicator
`fseek`	Repositions a file pointer in a stream
`fsetpos`	Sets file position indicator for stream
`ftell`	Repositions a file pointer in a stream (see `fseek`)
`rewind`	Repositions a file pointer in a stream (see `fseek`)

Flexible File I/O (FFIO) Functions

Function	**Description**
`ffbksp`	Repositions an FFIO file (see `ffseek`)
`ffclose`	Closes a file using FFIO (see `ffopen`)
`fffcntl`	Performs functions on files opened using FFIO
`fflistio`	Initiates a list of I/O requests using FFIO
`ffopen`	Opens a file using FFIO
`ffopens`	Opens a file using FFIO (see `ffopen`)
`ffpos`	Positions files opened using FFIO
`ffread`	Provides FFIO
`ffreada`	Provides asynchronous read using FFIO
`ffseek`	Repositions an FFIO file
`ffsetsp`	Initiates EOV processing for files opened using FFIO
`ffweod`	Provides FFIO (see `ffread`)
`ffweof`	Provides FFIO (see `ffread`)
`ffwrite`	Provides FFIO (see `ffread`)
`ffwritea`	Provides asynchronous write using FFIO

For more information about these routines, see the *Application Programmer's I/O Guide*, Cray Research publication SG–2168. Man pages for these routines are found in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165.

Formatted I/O Functions

Function	**Description**
`fprintf`	Prints formatted output (see `printf`)
`fscanf`	Converts formatted input (see `scanf`)
`printf`	Prints formatted output

`scanf`	Converts formatted input
`sprintf`	Prints formatted output (see `printf`)
`sscanf`	Converts formatted input (see `scanf`)
`vfprintf`	Prints formatted output of a `varargs` argument list (see `vprintf`)
`vprintf`	Prints formatted output of a `varargs` argument list
`vsprintf`	Prints formatted output of a `varargs` argument list (see `vprintf`)

Operations on Files

Function	**Description**
`fileno`	Returns indication of stream status
`ftruncate`	Truncates a file to a specified length
`mktemp`	Makes a unique file name
`remove`	Removes files
`rename`	Renames a file
`tempnam`	Creates a name for a temporary file (see `tmpnam`)
`tmpfile`	Creates a temporary binary file
`tmpnam`	Creates a name for a temporary file

User Information Functions

Function	**Description**
`ctermid`	Generates file name for terminal
`cuserid`	Gets character login name of the user

NAME

`ISELFADD`, `ICRITADD` – Allows performance of *ivar* = *ivar*+`IVALUE` under the protection of a hardware semaphore

SYNOPSIS

jvar = `ISELFADD(`*ivar*, *ivalue*`)`

`CALL ICRITADD(`*ivar*, *ivalue*`)`

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

`ISELFADD` is a function, and `ICRITADD` is a routine.

The following is a list of valid arguments:

Argument	**Description**
ivar	Integer variable to be incremented by *ivalue*.
ivalue	Amount by which *ivar* should be incremented.

A call to `ISELFADD` is functionally equivalent to, but considerably faster than, the following code block:

```
CALL LOCKON(lockvar)
jvar = ivar
ivar = ivar + ivalue
CALL LOCKOFF(lockvar)
```

A call to `ICRITADD` is functionally equivalent to, but considerably faster than, the following code block:

```
CALL LOCKON(lockvar)
ivar = ivar + ivalue
CALL LOCKOFF(lockvar)
```

SEE ALSO

`XSELFADD`(3F)

NAME

ISELFMUL, ICRITMUL – Allow performance of *ivar* = *ivar**IVALUE under the protection of hardware semaphore

SYNOPSIS

jvar = ISELFMUL(*ivar*, *ivalue*)

CALL ICRITMUL(*ivar*, *ivalue*)

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

ISELFMUL is a function, and ICRITMUL is a routine.

The following is a list of valid arguments:

Argument	**Description**
ivar	Integer variable to be multiplied by *ivalue*.
ivalue	Amount by which *ivar* should be multiplied.

A call to ISELFMUL is functionally equivalent to, but considerably faster than, the following code block:

```
CALL LOCKON(lockvar)
jvar = ivar
ivar = ivar*ivalue
CALL LOCKOFF(lockvar)
```

A call to ICRITMUL is functionally equivalent to, but considerably faster than, the following code block:

```
CALL LOCKON(lockvar)
ivar = ivar*ivalue
CALL LOCKOFF(lockvar)
```

SEE ALSO

XSELFMUL(3F)

NAME

ISELFSCH – Allows performance of *ivar* = *ivar*+1 under the protection of a hardware semaphore

SYNOPSIS

jvar = ISELFSCH(*ivar*)

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

ISELFSCH allows performance of *ivar* = *ivar*+1 under the protection of a hardware semaphore.

The following is a valid argument for this routine:

Argument	**Description**
ivar	Integer variable to be incremented

A call to ISELFSCH is equivalent to, but considerably faster than, the following code block:

```
CALL LOCKON(lockvar)
jvar = ivar
ivar = ivar+1
CALL LOCKOFF(lockvar)
```

SEE ALSO

XSELFMUL(3F)

NAME

`isgreater`, `isgreaterequal`, `isless`, `islessequal`, `islessgreater`, `isunordered` – Determines the relationship between two arguments

SYNOPSIS

```
#include <fp.h>

int isgreater (floating-type x, floating-type y);
int isgreaterequal (floating-type x, floating-type y);
int isless (floating-type x, floating-type y);
int islessequal (floating-type x, floating-type y);
int islessgreater (floating-type x, floating-type y);
int isunordered (floating-type x, floating-type y);
```

IMPLEMENTATION

CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

ANSI/IEEE Std 754-1985
X3/TR-17:199x

DESCRIPTION

The relational and equality operators (<, >, >=, <=, ==, and !=) support the usual mathematical relationships between numeric values. For any ordered pair of numeric values, exactly one of the relationships (less, greater, or equal) is true. Relational operators may raise the invalid exception when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the unordered relationship is true. The macros described here are *quiet* (do not raise exceptions) versions of the relational operators that facilitate writing efficient code that accounts for NaNs without raising the invalid exception.

The *floating-type* type in the SYNOPSIS section indicates an argument of any floating type. If the argument is not a floating type, the behavior is undefined.

If any of the macro definitions are suppressed in order to access an actual function, or if a program defines an external identifier with the name of one of the macros, the behavior is undefined.

The `isgreater` macro determines whether its first argument is greater than its second argument.

The `isgreaterequal` macro determines whether its first argument is greater than or equal to its second argument.

The `isless` macro determines whether its first argument is less than its second argument.

The `islessequal` macro determines whether its first argument is less than or equal to its second argument.

The `islessgreater` macro determines whether its first argument is less than or greater than its second argument.

The `isunordered` macro determines whether its arguments are unordered (that is, at least one argument is a NaN).

The IEEE standard enumerates 26 functionally distinct comparison predicates, including combinations of the four comparison results and whether invalid is raised. The following table shows how the Cray Research implementation covers all important cases.

IEEE comparisons

				Raises	
Greater	Less	Equal	Unordered	exception	Cray implementation
		X			`x == y`
X	X		X		`x != y`
X				X	`x > y`
X		X		X	`x >= y`
	X			X	`x < y`
	X	X		X	`x <= y`
			X		`isunordered(x,y)`
X	X			X	N/A
X	X	X		X	N/A
X			X		`! islessequal(x,y)`
X		X	X		`! isless(x,y)`
	X		X		`! isgreaterequal(x,y)`
	X	X	X		`! isgreater(x,y)`
		X	X		`! islessgreater(x,y)`
	X	X	X	X	`! (x > y)`
	X		X	X	`! (x >= y)`
X		X	X	X	`! (x < y)`
X			X	X	`! (x <= y)`
X	X	X			`! isunordered(x,y)`
		X	X	X	N/A
			X	X	N/A
	X	X			`islessequal(x,y)`
	X				`isless(x,y)`
X		X			`isgreaterequal(x,y)`
X					`isgreater(x,y)`
X	X				`islessgreater(x,y)`

RETURN VALUES

The `isgreater` macro returns a nonzero value if its first argument is greater than its second argument.

The `isgreaterequal` macro returns a nonzero value if its first argument is greater than or equal to its second argument.

The `isless` macro returns a nonzero value if its first argument is less than its second argument.

The `islessequal` macro returns a nonzero value if its first argument is less than or equal to its second argument.

The `islessgreater` macro returns a nonzero value if its first argument is less than or greater than its second argument.

The `isunordered` macro returns a nonzero value if its arguments are unordered.

SEE ALSO

Migrating to the CRAY T90 Series IEEE Floating Point, Cray Research publication SN–2194

NAME

`isnan` – Test for NaN

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The `isnan` function tests whether *x* is NaN (not a number).

RETURN VALUES

On Cray Research machines supporting IEEE arithmetic, the `isnan` function returns a nonzero value if *x* is NaN; otherwise, a 0 is returned.

On Cray Research machines with Cray Research floating-point format, `isnan` always returns a 0.

NOTES

A function-like macro version of `isnan` is implemented on Cray MPP systems and CRAY T90 systems with IEEE-standard floating-point hardware (see the `fpclassify`(3C) man page for more information). This IEEE version is defined in the `<fp.h>` header file.

If you are using a CRAY T90 system with IEEE-standard floating-point hardware, the `<fp.h>` version of `isnan` offers the advantage of accepting `float` and `long double` arguments as well as `double` arguments. The Cray MPP systems version accepts only `double` arguments.

The `<math.h>` version described on this man page offers XPG4 compatibility, and it is available on all Cray Research systems.

SEE ALSO

`fpclassify`(3C) on Cray Research IEEE systems for a description of the `<fp.h>` version of `isnan`

NAME

iso_addr, iso_ntoa – Manipulates ISO/OSI address

SYNOPSIS

```
#include <sys/types.h>
#include <netiso/iso.h>

struct iso_addr *iso_addr (char *cp);

char *iso_ntoa (struct iso_addr *isoa);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The iso_addr function interprets a character string (*cp*) representing numbers that reference an ISO network service access point (NSAP) address, returning a structure that is a valid ISO address.

The iso_ntoa function does the reverse, taking an ISO address structure (*isoa*) and returning an ASCII string representing the NSAP address.

RETURN VALUES

The iso_addr function returns a pointer to an iso_addr structure. The iso_addr function returns a null pointer for requests that are not in an accepted format of hexadecimal characters or single-quoted ASCII characters.

The iso_ntoa function always returns a character pointer.

FILES

/usr/include/netiso/iso.h

/usr/include/sys/types.h

SEE ALSO

gethostinfo(3C)

hosts(5), networks(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

kerberos_rpc, authkerb_getucred, authkerb_seccreate, svc_kerb_reg – Library routines for remote procedure calls that use Kerberos authentication

SYNOPSIS

```
cc LDFLAGS += -lkrb -lcraylm
#include <rpc/rpc.h>
#include <sys/types.h>
```

int authkerb_getucred(struct svc_req **rqst*, uid_t **uid*, gid_t **gid*, short **grouplen*, int grouplist[*NGROUPS*]);

AUTH *authkerb_seccreate(char **service*, char **srv_inst*, char **realm*, unsigned int *window*, char **timehost*, int **status*);

int svc_kerb_reg(SVCXPRT **xprt*, char **name*, char **inst*, char **realm*);

IMPLEMENTATION

All Cray Research systems licensed for Open Network Computing Plus (ONC+)

DESCRIPTION

Remote Procedure Call (RPC) library routines let C programs make procedure calls on other machines across the network.

RPC supports various authentication flavors, including the following:

Flavor	**Description**
AUTH_DES	DES encryption-based authentication
AUTH_KERB	Kerberos encryption-based authentication
AUTH_NULL	(none) No authentication
AUTH_SHORT	Shorthand form of UNICOS credentials
AUTH_UNIX	Traditional UNIX-style authentication

The authkerb_getucred, authkerb_seccreate, and svc_kerb_reg routines implement the AUTH_KERB authentication flavor. The user must have run kinit(1) or ksrvtgt(1) in all cases. This man page discusses only the AUTH_KERB style of authentication.

For more information about the AUTH_NULL, AUTH_UNIX and AUTH_DES styles of authentication, see the rpc(3C) man page. See the *Remote Procedure Call (RPC) Reference Manual*, Cray Research publication SR–2089, for a definition of the AUTH data structure.

authkerb_getucred

The server side routine, authkerb_getucred, converts an AUTH_KERB credential received in an RPC request, which is operating-system independent, into an AUTH_UNIX credential. If this routine succeeds, it returns 1; if it fails, it returns 0.

The *uid* is set to the numerical ID of the user associated with the RPC request referenced by *rqst*. *gid* is set to the numerical ID of the user's group. The numerical IDs of the other groups to which the user belongs are stored in `grouplist`(). *grouplen* is set to the number of valid group ID entries returned in `grouplist`(). All information that this routine returns is based on the Kerberos principal name contained in `rqst`. This principal name is assumed to be the login name of the user, and the IDs returned are the same as if that user had physically logged in to the system.

authkerb_seccreate

The client side routine, `authkerb_seccreate`, returns an authentication handle that enables the use of the Kerberos authentication system. The *service* parameter is the Kerberos principal name of the service to be used. This name is generally a constant with respect to the service being used.

The *srv_inst* is the instance of the service to be called. *realm* is the Kerberos realm name of the desired service; if it is `NULL`, the local default *realm* is used.

The *window* parameter validates client credential, with time measured in seconds. If the difference in time between the client's clock and the server's clock exceeds the time value of *window*, the server rejects the client's credentials, and the clock must be resynchronized. On a Cray Research machine the `ntpd`(8) command provides this function. A small window is more secure than a large one.

The *timehost* parameter is optional and does nothing. Client and server should run the network time protocol (NTP) to synchronize time.

The *status* parameter is also optional. If you specify status, it is used to return a Kerberos error status code if an error occurs. If status is `NULL`, no detailed error codes are returned.

If `authkerb_seccreate` fails, it returns `NULL`.

svc_kerb_reg

The server routine, `svc_kerb_reg`, performs registration tasks in the server that are required before `AUTH_KERB` requests are processed. *xprt* is the UDP RPC transport handle which is associated with this information. Only the UDP transport handles may be registered with the *xprt* parameter. If *xprt* is `NULL`, this registration is effective for any requests that arrive on transports that have not been specifically registered. If you use the *xprt* parameter to register transports, you must use a separate `svc_kerb_reg` call for each transport.

The *name*, *inst* and *realm* parameters describe the Kerberos principal identity that this server assumes. This identity must be the same identity that the clients use when requesting Kerberos tickets for authentication. The required *name* parameter is the principal name of the service. *inst* is the instance; most common value for *inst* is `*`, which allows the Kerberos library to determine the correct instance to use, (such as the hostname on which the service is running). *realm* is the Kerberos `realm` name to use in validating tickets. If it is `NULL`, the local default *realm* is used.

Generally, `svc_kerb_reg` should be called immediately before `svc_run`. If the routine succeeds, it returns 0; if it fails, it returns -1. Kerberos RPC servers must be run as root to access the `/etc/srvtab` file to decrypt authentication messages.

NOTES

You must be licensed for Open Network Computing Plus (ONC+) to use Kerberos encryption-based authentication.

You must load the following library routines and include files along with your C program:

```
cc LDFLAGS += -lkrb -lcraylm
#include <rpc/rpc.h>
#include <sys/types.h>
```

You must install Kerberos enigma for Kerberized RPC to function. These interfaces are unsafe in multithreaded applications; therefore you should call unsafe interfaces only from the main thread.

SEE ALSO

`kerberos`(3K) in the *Kerberos User's Guide*, Cray Research publication SG–2409

`rpc`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`kinit`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`kerberos`(7) available only online

`ntpd`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`killpg` – Sends signal to a process group

SYNOPSIS

```
#include <signal.h>
int killpg (int pgrp, int sig);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The `killpg` function sends the signal *sig* to the process group *pgrp*. See `signal`(2) for a list of signals.

The sending process and members of the process group must have the same effective user ID, or the sender must be the super user.

The `killpg` function is provided as a compatibility function. It is equivalent to the `kill`(2) system call with arguments *sig*, and *pgrp* multiplied by –1.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned, and `errno` is set to indicate the error. See `kill`(2) for a list of error codes.

SEE ALSO

`errno.h`(3C)

`kill`(2), `signal`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

`l3tol`, `ltol3` – Converts between 3-byte integers and long integers

SYNOPSIS

```
#include <stdlib.h>
void l3tol (long *lp, char *cp, int n);
void ltol3 (char *cp, long *lp, int n);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

AT&T extension

DESCRIPTION

The `l3tol` function converts a list of *n* 3-byte integers packed into a character string to which *cp* points into a list of long integers to which *lp* points.

The `ltol3` function performs the reverse conversion from long integers (*lp*) to 3-byte integers (*cp*).

These functions are useful for file-system maintenance in which the block numbers consist of 3 bytes.

CAUTIONS

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

SEE ALSO

`fs`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

lgamma, gamma, signgam – Computes log gamma function

SYNOPSIS

```
#include <math.h>

double gamma (double x);

double lgamma (double x);

int signgam;
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The gamma function behaves identically to the lgamma function, which will be referred to on the remainder of this page. Use of the name lgamma is preferred to gamma, which may be withdrawn in a future release.

The lgamma function returns $\ln(|\Gamma(x)|)$, where $\Gamma(x)$ is defined as $\int_0^\infty e^{-t} t^{x-1} dt$. The sign of $\Gamma(x)$ is returned in the external integer *signgam*. If x is negative, it must not have an integral value. Argument x may not be 0.

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

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

LN_MAXDOUBLE is the least value that causes exp to return a range error. LN_MAXDOUBLE is defined in the header file values.h(3C).

Vectorization is inhibited for loops containing calls to the lgamma function.

RETURN VALUES

For nonpositive integer arguments, HUGE_VAL is returned, and errno is set to EDOM.

If the correct value would overflow, lgamma returns HUGE_VAL and sets errno to ERANGE.

On Cray MPP systems and CRAY T90 systems with IEEE arithmetic, `lgamma(`*NaN*`)` and `gamma(`*NaN*`)` return NaN and `errno` is set to `EDOM`.

On Cray MPP systems and CRAY T90 systems with IEEE arithmetic, the value returned by the `lgamma` and `gamma` functions when a domain error occurs can be selected by setting the environment variable `CRI_IEEE_LIBM`. The second column describes what is returned when `CRI_IEEE_LIBM` is not set, or is set to a value other than 1. The third column describes what is returned whe `CRI_IEEE_LIB` is set to 1.

Error	`CRI_IEEE_LIB`=0	`CRI_IEEE_LIB`=1
`lgamma(`*x*`)`, where *x* is less than zero	`HUGE_VAL`	NaN
`gamma(`*x*`)`, where *x* is less than zero	`HUGE_VAL`	NaN

SEE ALSO

`exp`(3C), `values.h`(3C)

`values`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

addudb, deleteudb, endudb, getsysudb, gettrustedudb, getudb, getudbchain, getudbdefault, getudbnam, getudbstat, getudbtmap, getudbuid, lockudb, rewriteudb, setudb, setudbdefault, setudbpath, setudbtmap, udbisopen, unlockudb, zeroudbstat – Library of user database access functions

SYNOPSIS

```
#include  <udb.h>

int addudb (struct udb *udb);

int deleteudb (char *name);

void endudb (void);

void getsysudb (void);

void gettrustedudb (void);

struct udb *getudb (void);

struct udb *getudbchain (int option);

struct udbdefault *getudbdefault (void);

struct udb *getudbnam (char *name);

struct udbstat *getudbstat (void);

struct udbtmap *getudbtmap (void);

struct udb *getudbuid (int uid);

int lockudb (void);

int rewriteudb (struct udb *udb);

int setudb (void);

int setudbdefault (struct udbdefault *def);

int setudbpath (char *path);

int setudbtmap (struct udbtmap *tmap);

int udbisopen (void);

extern int udb_errno;

void unlockudb (void);

void zeroudbstat (void);
```

The following routines are for Cray Research internal use only:

```
const Udbhdr *udb_header_access(const long magic, const Hdrfield field,
const void *value);
```

```
const char *udb_strerror(const int code);
```

```
int resetmaxuid(void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The user database (UDB) contains control information for users of the UNICOS operating system and for the fair-share scheduler's resource groups. The UDB files replace the /etc/passwd file as the primary source for user validation and control information.

The UDB consists of the following files:

- /etc/udb
- /etc/udb.public
- /etc/udb_2/udb.index
- /etc/udb_2/udb.priva
- /etc/udb_2/udb.pubva

The files in the directory /etc/udb_2 extend the capability of the UDB beyond what was available in previous releases.

To allow users access to nonsensitive UDB information, the files /etc/udb.public, /etc/udb_2/index, and /etc/udb_2/udb.pubva are publicly readable. The other files contain privileged information, such as encrypted passwords and security information, and can be read only by privileged callers. Write access to all files is restricted to privileged users.

The UDB files are binary files that are carefully constructed (and carefully accessed) so that multiple reader processes can access them without being disturbed by ongoing modifications of the database. That is, privileged processes can update the database concurrently with other processes reading the database. In addition, the libudb(3C) functions use hash lists and a direct-access index to speed the search for locating user information by user ID (UID), resource group ID, user name, and resource group name.

Because of this capability for multiple accesses, the UDB should be accessed only through the provided library functions described in the following paragraphs. These functions provide access to the user information and system defaults contained in the UDB. Other access methods might corrupt the database and require regeneration of the files.

The `getudb`, `getudbnam`, and `getudbuid` functions each return a pointer to an object with the following UDB structure containing the broken-out fields of one entry from the UDB. If the pointer returned has the value `UDB_NULL`, a record could not be returned for the reason set in the `udb_errno` file. When `getudb` is being used for sequential reading of the database, a `UDB_NULL` pointer along with the value `UDBERR_END` in `udb_errno` signals the end of the database.

If more than one record has the same UID, then `getudbuid` returns the most recently added record. (To check for multiple identical UIDs after a call to `getudbuid`, use the `getudbchain` function with the `UDBCHAIN_PRIOR` direction option and check the UID of the returned record.)

The `getudbchain` function is used to take advantage of the UID chain maintained in the database. The chain is bidirectional, ordered by UID value. The direction of ascending value of UID is called *next*, and the direction of descending value of UID is called *prior*. Each `getudb`, `getudbnam`, or `getudbuid` call sets the current position to that of the returned record; a call to `setudb` or to `getudbchain` with *option* set to `UDBCHAIN_FIRSTNR` sets the current position so that the next record is the one with the smallest UID value. Definitions for the information shown in the structure are found in `udb.h`.

Option **Description**

`UDBCHAIN_FIRST` UID value 0. Returns the first record in the UID chain and sets the current position to that record.

`UDBCHAIN_FIRSTNR` UID value 1. Sets the current chain position so that the next record is the first record in the UID chain (smallest numeric UID value). A record is not returned, so the returned pointer is `UDB_NULL` and `udb_errno` is `0`. An error has occurred if `udb_errno` is nonzero.

`UDBCHAIN_LAST` UID value 2. Returns the last record in the UID chain and sets the current position to that record.

`UDBCHAIN_LASTNR` UID value 3. Sets the current chain position so that the next record is the last record in the UID chain (largest numeric UID value). A record is not returned, so the returned pointer is `UDB_NULL` and `udb_errno` is `0`. An error has occurred if `udb_errno` is nonzero.

`UDBCHAIN_NEXT` UID value 4. Returns the next record in the UID chain. The end of the chain is indicated by the returned pointer being set to `UDB_NULL` and `udb_errno` being set to `UDBERR_END`.

`UDBCHAIN_PRIOR` UID value 5. Returns the prior record in the UID chain. The beginning of the chain is indicated by the returned pointer being set to `UDB_NULL` and `udb_errno` being set to `UDBERR_END`.

The addudb and rewriteudb functions take a struct udb object and create or update a named record in the UDB; deleteudb removes the named record. When the return value is UDB_FAIL, it means that the requested function could not be done for the reason set in udb_errno. Before any of these functions is called, a call to lockudb is required to prevent multiple processes from writing to the database at the same time, thereby possibly corrupting it. The database is automatically unlocked upon return from these functions. Each call to lockudb should be checked for the return value UDB_FAIL, which indicates that the lock could not be obtained. All of these functions require write permission to the database.

A call to setudb has the effect of rewinding the database to allow sequential reading with getudb, and it also positions the chain pointer to the first record (lowest UID value) for use by getudbchain. The endudb function can be called to close the database when processing is complete.

For security reasons, UDB information is divided into protected and public categories; the access method retrieves only the public information unless special library set-up calls are made. A caller needing specific access to the protected category of information (to read or update the information in any way) must make a call to getsysudb or gettrustedudb after the optional setudbpath call but before any other calls to the database are made. If this is not done, the public file will be opened and no updating or access to protected information will be possible unless endudb is called to close the files. To guarantee predictable behavior for the system security programs, the various get requests will fail if the protected copy of the database cannot be accessed. The getsysudb function also causes such behavior.

The difference between getsysudb and gettrustedudb is that no files remain open for writing after an unlockudb call (including the implicit calls from addudb, deleteudb, and rewriteudb) unless gettrustedudb is called. Except for updates of multiple records, use getsysudb when updating the database, because this improves data integrity.

The setudbpath function changes the path name to the database files and can be used to support multiple user databases. The path must be accessible to the caller and must end in the name of a directory. If this function is called after the database is opened and the name changes because of this call, the previously accessed database will be closed. If the pointer or the string is null, the path will be restored to the default. The entire file name is limited in length to the value UDBFNAMELEN (1024 characters). If a stat(2) call using *path_name*/udb_2 returns a directory named udb_2, the library assumes that the files udb.index, udb.priva, and udb.pubva will be found in that directory. If udb_2 is not found, those files are assumed to exist in *path_name* rather than *path_name*/udb_2.

The getudbstat function returns a pointer to a udbstat structure (shown later in this section) that contains statistical and other information related to the UDB access method. The values in the structure are not guaranteed to be accurate and static except immediately following this call and before any other UDB library calls are made. (Make a private copy of the returned structure if historical information is needed.) The zeroudbstat function resets all statistical information and may be used after setudbpath is called to restart the gathering of statistics on the new database.

The udbisopen function returns the open state of UDB files. When no files are open, 0 is returned; otherwise the return values are as follows:

State	**Value**
Protected open read	00001

Protected open write	00002
Public open read	00004
Public open write	00010
Index open read	00020
Index open write	00040

These values may be returned in combinations.

The `getudbdefault` function returns a pointer to the current default table (`struct udbdefault`). This table contains the defaults currently recorded in the UDB. The `setudbdefault` function replaces the default table in the UDB with a new table. The `setudbdefault` function requires that the calling process have write permission to the UDB.

The `getudbtmap` function returns a pointer to the current global tape name structure (`struct udbtmap`). Each of the eight tape name ordinals is represented in this table. If an ordinal does not have an associated global name, the name string is null. The `setudbtmap` function replaces the existing tape name map in the UDB with a new tape name map. The `setudbtmap` function requires that the calling process have write permission to the UDB.

The `udb_strerror` function returns the message string associated with a given UDB error code.

Description of struct udb

```
struct udb {
    char      ue_passwd[MAXUE_EPASSWD + 1];      encrypted password;
    char      ue_comment[MAXUE_COMMENT + 1];     comment
    char      ue_dir[MAXUE_HOMEDIR + 1];         default login directory
    char      ue_shell[MAXUE_SHELL + 1];         default login shell or program
    char      ue_age[MAXUE_AGE + 1];             included for compatibility; not used
    int       ue_acids[MAXVIDS];                 valid account ids
    int       ue_gids[MAXVIDS];                  valid group ids
    char      ue_root[MAXUE_LOGINROOT + 1];      login root directory
    char      ue_logline[MAXUE_LOGLINE + 1];     line used for last login
    char      ue_loghost[MAXUE_HOSTNAME + 1];    hostname for last login
    char      ue_batchhost[MAXUE_HOSTNAME + 1];  hostname of last batch req origin
    long      ue_logtime;                        time of last login (GMT in secs)
    long      ue_batchtime;                      time of last batch request
    long      ue_permbits;                       user permission bits
    long      ue_sitebits;                       site supported permission bits
    long      ue_archlim;                        disk space protected from archiving
    int       ue_archmed;                        archiving medium selector
    long      ue_jproclim[MAXUE_RCLASS];         per job max # processes
    long      ue_jcpulim[MAXUE_RCLASS];          per job cpu limit [seconds]
    long      ue_pcpulim[MAXUE_RCLASS];          per proc. cpu limit [seconds]
    long      ue_jmemlim[MAXUE_RCLASS];          per job mem. limit [512 words]
    long      ue_pmemlim[MAXUE_RCLASS];          per proc. mem. limit [512 words]
    long      ue_pfilelim[MAXUE_RCLASS];         per proc. file size limit: [512 words]
    unsigned char                                per job tape limit
              ue_jtapelim[MAXUE_RCLASS][MAXUE_TAPETYPE];
    int       ue_nice[MAXUE_RCLASS];             user’s nice value (0..19)
    int       ue_logfails;                       #of consecutive login failures
    int       ue_deflvl;                         default security level
    int       ue_maxlvl;                         maximum security level
    int       ue_minlvl;                         minumum security level
    long      ue_defcomps;                       default compartments at login
    long      ue_comparts;                       valid compartments
    int       ue_permits;                        valid permissions
    int       ue_disabled;                       user login disabled flag
    int       ue_trap;                           trap on login
    char      ue_name[16];                       user’s login name
    int       ue_uid;                            UID
    int       ue_resgrp;                         resource group UID
    long      ue_shflags;                        share flags
    short     ue_shares;                         allocated shares
    short     ue_shplimit;                       included for compatibility; not used
    mlimit_t  ue_shmlimit;                       included for compatibility; not used
    long      ue_jsdslim[MAXUE_RCLASS];          per job sds limit
    long      ue_psdslim[MAXUE_RCLASS];          per process sds limit
    float     ue_shusage;                        decaying accum costs
    float     ue_shcharge;                       long-term accum cost
    long      ue_shextime;                       time last lnode freed
    long      ue_limflags;                       included for compatibility; not used
    int       ue_umask;                          included for compatibility; not used
    int       ue_warnings;                       included for compatibility; not used
    int       ue_reason;                         included for compatibility; not used
    int       ue_intcls;                         default integrity class (obsolete)
    int       ue_maxcls;                         maximum integrity class (obsolete)
    long      ue_intcat;                         default integrity categories
    long      ue_valcat;                         valid integrity categories
    long      ue_lastlogtime;                    time of last login attempt
    int       ue_cpu_quota;                      CPU quota (seconds * 10)
    int       ue_cpu_quota_used;                 accumulated CPU time (seconds * 10)
    long      ue_jfilelim[MAXUE_RCLASS];         per job new file space allocation limit
    struct ue_pwage {
       long   time;                              second clock when password was changed
       int    flags;                             see PWFL_xxx flags
       int    maxage;                            maximum password age in seconds
       int    minage;                            minimum password age in seconds
    } ue_pwage;
    int       ue_parentuid;                      UID of this user's administrator
    int       ue_adminmax;                       number of users this UID can administer
    int       ue_jpelimit[MAXUE_RCLASS];         per job max # MPP PEs
    int       ue_jmpptime[MAXUE_RCLASS];         per job max time MPP rsvd
    int       ue_jmppbarrier[MAXUE_RCLASS];      per job max MPP barriers
    int       ue_pmpptime[MAXUE_RCLASS];         per process max time MPP rsvd
    int       ue_pcorelim[MAXUE_RCLASS];         per proc core file limit [512 words]
    int       ue_pfdlimit[MAXUE_RCLASS];         per process open file limit
    int       ue_mincomps;                       default compartments at login
    int       ue_jshmsegs[MAXUE_RCLASS];         per job max shared memory segments
    int       ue_jshmsize[MAXUE_RCLASS];         per job max shared memory [512 words]
    int       ue_jsocbflim[MAXUE_RCLASS];        per job max socket buffer limit [512 words]
};
```

`ue_passwd` Encrypted password of no more than 15 characters.

`ue_comment` Arbitrary string of no more than 39 characters usually including the user's name, department, and other personal information.

`ue_dir` Default login directory (home directory) name of no more than 63 characters.

`ue_shell` Default login shell or program name up to 63 characters in length.

`ue_age` The `ue_age` field is obsolete, but remains present for compatibility reasons. It is a read-only field which is constructed by the interface library from the newly added `ue_pwage` structure. For more information, see the description of `ue_pwage`.

ue_acids List of zero or more (maximum of 64) numerical account IDs (ACIDs) for the user. When fewer than 64 ACIDs are declared, the list is terminated by an entry with the value −1; udbgen(8) and udbsee(1) present these as comma-separated values.

ue_gids List of zero or more (maximum of 64) numerical group IDs (GIDs) for the user. When fewer than 64 GIDs are declared, the list is terminated by an entry with the value −1; udbgen(8) and udbsee(1) present these as comma-separated values.

ue_root Login root directory name consisting of a maximum of 63 characters.

ue_logline A string of no more than 15 characters specifying the line or port from which the most recent login originated.

ue_loghost A string of no more than 31 characters specifying the host from which the most recent login originated.

ue_batchhost

A string of no more than 31 characters naming the host from which the most recently submitted batch job originated.

ue_logtime The time at which the most recent login occurred.

ue_batchtime

The time at which the most recently submitted batch job arrived.

ue_permbits User permission bits. This is a bit list in which the meaning of each bit is defined in udb.h; udbgen(8) and udbsee(1) present these as named bits separated by commas. For a list that maps user permission bits to kernel permbits, see getpermit(2).

Bit	**Description**
PERMBITS_ACCT	Accounting permission. Allows the user to run the accton(8) and csaswitch(8) commands. The accton(8) command turns process accounting on and off. The csaswitch(8) command checks the status of and enables or disables process, daemon, and record accounting.
PERMBITS_ACCTID	Allows the user to use the diskusg(8) command to merge intermediate disk accounting records. By using the chacid(1) command, the user may set the account ID of a file that is owned by another user and may specify any account ID value. By using the quota(1) command, the user can see all acccount IDs, group IDs, and user IDs. By using the newacct(1) utility, the user can change the account ID of the calling shell.

`PERMBITS_ASKACID` Allows queries for active account ID. When this permbit is assigned to a user login account in the UDB, and the account also has multiple account IDs or the `acctid` (`PERMBITS_ACCTID`) permbit, `login` prompts the user for the account ID that should be assigned to the session.

`PERMBITS_BYPASSLABEL` Allows the user to bypass label processing. This permission bit replaces the `lbypass` permit.

`PERMBITS_CHOWN` Allows the user to change owner (`chown`(2)), change group (`chgrp`(1)), or change permissions (`chmod`(1)) for any file owned by that user.

`PERMBITS_CHROOT` Allows the user to use the `chroot`(8) command to execute a command relative to a newroot.

`PERMBITS_DEDIC` Allows the user to dedicate a CPU to a process.

`PERMBITS_DEVMAINT` Allows the user to use the `ddms`(8) (disk diagnostic and maintenance system) command without being super user.

`PERMBITS_GROUPADM` Allows the user to be a group administrator. The `xadmin`(8) utility supports group administration and this permission controls which users are group administrators. Other configuration of `xadmin`(8) is also required to support group administration.

`PERMBITS_GUARD` Driver DONUT guard mode. When running disk diagnostics, a user with this permission can access user data space. This is denied without this privilege.

`PERMBITS_GUEST` Allows use of guest operating system. Allows the user access to most of the functionality provided by the `guest`(1) command. This includes starting, stopping, changing and dumping a guest. This permission is not needed to obtain guest status.

`PERMBITS_GUESTADM` Allows administration of guest operating system. The use of certain `guest`(1) command line options is controlled through this permission. The controlled options are those that may have a global system effect, including the following:

`-T` Enable or disable extended kernel tracing.

`-K` Enable or disable panicking the host if the guest panics.

`-D` Dump all active systems.

`-O` Usurp guest system from the original owner.

-P Change CPU percentages of active guests.

PERMBITS_ID
Allows ID changes. Allows the user to set his or her real, effective, and saved-set user IDs (setuid(2)). Allows the user to set his or her real, effective, saved-set group IDs (setgid(2)), and group list (setgroups(2)). Allows the user to set his or her account ID (acctid(2)) and the account ID of a file (chacid(2)).

PERMBITS_IPCPERSIST
Allows the user to allocate persistent shared memory blocks. When a user with this permission uses the msgget(2), semget(2) or shmget(2) system calls, reserved memory segments are retained beyond the life of the creating session.

PERMBITS_MKNOD
Allows the user to use the mknod(2) system call with a mode other than S_IFIFO.

PERMBITS_MLSMNT
Unused. This permbit is available to assign user accounts, but it no longer grants special abilities.

PERMBITS_MOUNT
Allows mount. The mount(2) system call that allows a user to mount a file system, requires this permission or super user privilege and returns an EPERM error code if the user is not permitted. The umount(2) system call that allows a user to unmount a file system, requires this permission or super user privilege and returns an EPERM error code if the user is not permitted.

PERMBITS_NICE
Allows nice negative values. The nice(2) system call that allow users to change their nice value requires this privilege or super-user if the nice value is not zero (0). The system call will return an EPERM error code if the user is not permitted.

PERMBITS_NOBATCH
Denies users permission to run batch processes. The setlimits() call from the login(1) command for a batch job request will fail if the user has this permission and the user will see the message "setlimits: batch sessions not permitted".

PERMBITS_NOIACTIVE
Denies users permission to run interactive processes. The setlimits() call from the login(1) command for an interactive job request will fail if the user has this permission and the user will see the message "setlimits: interactive sessions not permitted".

PERMBITS_PLOCK

Allows a process to lock itself in memory (plock(2)). The plock(2) system call that allows a process to become locked into memory during execution requires this privilege or super-user. The system call will return an EPERM error code if the user is not permitted. This privilege is added to the process permits during a cpuopen() call.

PERMBITS_REALTIME

Allows the user to activate real-time processes. The cpucntl(CPU_SETRT) call that allows the process to set the real-time execution state requires this privilege or super-user. The system call will return an EPERM error code if the user is not permitted.

PERMBITS_RESLIM

Resource limits permission. The limits(2) system call for function L_SETLIM requires this privilege or super-user to connect the process to a new limits structure that is passed as an lnode structure. The system call will return an EPERM error code if the user is not permitted.

The limits(2) system call for function L_DEADGROUP requires this privilege or super-user to collect the dead limits structure and returns an lnode structure. The system call will return an EPERM error code if the user is not permitted.

The limits(2) system call for function L_SETIDLE requires this privilege or super-user to set the limits fields in an idle limits structure that is passed as an lnode structure. The system call will return an EPERM error code if the user is not permitted.

The limits(2) system call for function L_CHNGLIM requires this privilege or super-user to change the limits fields in the limits structure that is passed as an lnode structure with the correct user ID. The system call will return an EPERM error code if the user is not permitted.

The limits(2) system call for function L_UPDATEKN requires this privilege or super-user to allow the shrdaemon to update the kernel lnode fields previously calculated by the kernel. The system call will return an EPERM error code if the user is not permitted.

The ulimit(2) system call for function UL_SETFSIZE requires this privilege or super-user if the file size limit is being increased. The system call will return an EPERM error code if the user is not permitted.

The setpermit(2) system call requires this privilege or super-user if the permits for a job or process are being increased. The system call will return an EPERM error code if the user is not permitted.

The cpu(4) system call with an ioctl for function CPU_RTPERMIT requires this privilege or super-user. The system call will return an EPERM error code if the user is not permitted.

PERMBITS_RESTRICTED Restricts system access (udbrstrict(8)). Set and cleared by the udbrstrict(8) command to allow or disallow creation of sessions, either batch or interactive, by the user.

PERMBITS_SIGANY Allows the user to send signals to any process, regardless of ownership.

PERMBITS_SUSPRES Allows the user to suspend and resume processes outside their own session.

PERMBITS_SYSPARAM Allows the user to change various system parameters. These include:

- System tick rate
- Maximum user error interrupts
- Memory scheduling parameters (nschedv(8))
- System memory size (chmem(8))
- Fair-share parameters
- CPU characteristics (target(1))
- Time-of-day

PERMBITS_TAPEMANAGER Allows the user special tape access privileges such as allowing tape formatting and mounting of tapes owned by other users.

PERMBITS_WRUNLABEL Allows the user to read and write unlabeled tapes. This permission bit replaces the wrunlab permit. For information on labeling tapes, see tplabel(8).

PERMBITS_YP Network information services (NIS) reference flag.

ue_sitebits There are 32 site-defined permission bits. These bits are named site1 (octal 01) through site32 (octal 020000000000). These bits can be set, cleared, and displayed through udbgen(8) and udbsee(1) by either their generic name or octal value.

ue_archlim Disk space immune to data migration.

`ue_archmed` An index to the medium selected for data migration. The meaning of this field is site-specific except for the value 0, which is reserved for the default medium; 0 is always valid in the released system.

`ue_jproclim` Job process limit. Two values exist, one for batch and the other for interactive work; `udbgen`(8) and `udbsee`(1) present the values as `jproclim[b]` for batch and `jproclim[i]` for interactive.

`ue_jcpulim` Job CPU time limit in seconds. Two values exist, one for batch and the other for interactive work; `udbgen`(8) and `udbsee`(1) present the values as `jcpulim[b]` for batch and `jcpulim[i]` for interactive.

`ue_pcpulim` Per-process CPU time limit in seconds. Two values exist, one for batch and the other for interactive work; `udbgen`(8) and `udbsee`(1) present the values as `pcpulim[b]` for batch and `pcpulim[i]` for interactive.

`ue_jmemlim` Job memory limit in units of 512 words (4096 characters). Two values exist, one for batch and the other for interactive work; `udbgen`(8) and `udbsee`(1) present the values as `jmemlim[b]` for batch and `jmemlim[i]` for interactive.

`ue_pmemlim` Per-process memory limit in units of 512 words (4096 characters). Two values exist, one for batch and the other for interactive work; `udbgen`(8) and `udbsee`(1) present the values as `pmemlim[b]` for batch and `pmemlim[i]` for interactive.

`ue_pfilelim` Per-process file allocation limit in units of 4096 characters (512 words). Two values exist, one for batch and the other for interactive work; `udbgen`(8) and `udbsee`(1) present the values as `pfilelim[b]` for batch and `pfilelim[i]` for interactive.

`ue_jtapelim` Job tape assignment limit. Sixteen values exist, 8 for batch and 8 for interactive work. Commands `udbgen`(8) and `udbsee`(1) present the values as `jtapelim[b][`*t*`]` for batch and `jtapelim[i][`*t*`]` for interactive; *t* is a tape type represented by an integer from 0 through 7.

`ue_nice` Job nice increment in the range 0 through 19; default is 0. Two values exist, one for batch and the other for interactive work; `udbgen`(8) and `udbsee`(1) present the values as `nice[b]` for batch and `nice[i]` for interactive. See `nice`(1) for more information.

`ue_logfails` The field in which the number of failed password attempts is recorded. Login is prohibited when the value of this field exceeds the limit defined at installation.

`ue_deflvl` Default security level assigned to the user at login. This is a numeric value with a default of 0.

`ue_maxlvl` Maximum security level allowed for the user; the default value is 0.

`ue_minlvl` Minimum security level allowed for the user; the default value is 0.

`ue_defcomps` Default security compartments, which are compartments assigned to the user at login. The format and meaning are included in the following `ue_comparts` description. The default is no initial security compartment assignment.

`ue_comparts` Authorized security compartments. These are compartments the user may select. The field is actually a bit list, but it is represented externally as a comma-separated list of compartments; the default is no authorized compartments. The valid compartment names and bits are defined in the `sys/secparm.h` include file, as follows:

Name	**Description**
`crayri`	Cray Research, Inc.
`netadm`	Network administrator
`secadm`	Security administrator
`sysadm`	System administrator
`sysops`	System operator
`unicos`	UNICOS system (obsolete)

`ue_permits` Permissions attributed to the user. The field is actually a bit list, but it is represented externally as a comma-separated list of permission names; the default is no permissions. The valid permissions are defined in the `sys/secparm.h` include file, as follows:

Permission	**Description**
`install`	This field is obsolete.
`lbypass`	This field is obsolete; see the `bypasslabel` permbit.
`reclsfy`	This field is obsolete.
`rmtaccs`	This field is obsolete.
`suidgid`	Allows the user to set the set-UID or set-GID bits for a file.
`usrtrap`	Sets the user in trap mode during login; all discretionary and mandatory access attempts are logged.
`wrunlab`	This field is obsolete; see the `wrunlab` permbit.

`ue_disabled` User password lock. When this field is nonzero, the user is not allowed to access the system.

`ue_trap` "Trap on login" security feature.

`ue_name` The user name, consisting of up to 8 alphanumeric characters. The first character in this field must be a letter; uppercase letters are allowed but are not recommended. This field **must** be defined and unique.

`ue_uid` Numeric user ID (UID); used internally in UNICOS utilities and the operating system. This field must be defined.

`ue_resgrp` Fair-share resource group UID.

`ue_shflags` Fair-share flags as defined in `sys/share.h`.

`ue_shares` Fair-share user's allocated shares.

`ue_shplimit` Included for compatibility; obsolete.

`ue_shmlimit` Included for compatibility; obsolete.

`ue_jsdslim` Job secondary data segment limit in units of 512 words (4096 characters). Two values exist, one for batch and the other for interactive work; `udbgen`(8) and `udbsee`(1) present the values as `jsdslim[b]` for batch and `jsdslim[i]` for interactive. The field is present but not used on Cray Research systems without an SSD.

`ue_psdslim` Process secondary data segment limit in units of 512 words (4096 characters). Two values exist, one for batch and the other for interactive work; `udbgen`(8) and `udbsee`(1) present the values as `pmemlim[b]` for batch and `pmemlim[i]` for interactive. This field is present but not used on Cray Research systems without an SSD.

`ue_shusage` Fair-share decaying accumulated costs.

`ue_shcharge` Fair-share user's long-term accumulated costs.

`ue_shextime` Time at which this user's last fair-share scheduler lnode was released. The *lnode* is a kernel structure that holds running user control information for the scheduler.

`ue_limflags` Included for compatibility; obsolete.

`ue_umask` Included for compatibility; obsolete.

`ue_warnings` Included for compatibility; obsolete.

`ue_reason` Included for compatibility; obsolete.

`ue_intcls` Default integrity class assigned to an administrator at login. This is a numeric value with a default of 0. This field is obsolete.

`ue_maxcls` Maximum integrity class allowed for an administrator. This is a numeric value with a default of 0. This field is obsolete.

`ue_intcat` Default category assigned to an administrator at login. The format and meaning are described in `ue_valcat`. The default is no initial integrity category assignment.

`ue_valcat` Authorized categories assigned to an administrator. These are categories that an administrator may enable. This field is a word, in which each category is represented by a single bit. A name is associated with each category. Externally, authorized categories are displayed as a comma-separated list of category names. Valid category names and bits are defined in the `sys/tfm.h` include file. Categories that may be assigned to an administrator are as follows:

Category	**Description**
`secadm`	Security administrator
`sysadm`	System administrator
`sysops`	System operator
`unicos`	UNICOS system (obsolete)
`sysfil`	System file (obsolete)

ue_lastlogtime
The time at which a user's most recent login attempt occurred. This time is updated whether or not the login attempt was successful.

ue_cpu_quota
CPU quota in seconds * 10. If this field is nonzero, the user will not be permitted to accumulate (in ue_cpu_quota_used) more than the value in this field.

ue_cpu_quota_used
Accumulated CPU time in seconds * 10. If ue_cpu_quota is nonzero and if the value in this field exceeds ue_cpu_quota, the user will not be permitted to run.

ue_jfilelim Per-job file allocation limit in units of 4096 characters (512 words). Two values exist, one for batch and the other for interactive work; udbgen(8) and udbsee(1) present the values as jfilelim[b] for batch and jfilelim[i] for interactive.

ue_pwage This is a structure containing a number of fields specifying the password age control for the record. Password age makes it possible for the administrator to control how long passwords remain valid and how they can be changed. Each of the fields in the structure will be described separately.

Field	Description
ue_pwage.time	A copy of the time of day clock when the password was last changed.
ue_pwage.flags	Control flags. If the value is PWFL_FORCE, the user must change the password at the next login. If the value is PWFL_SUPERUSER, only the administrator is allowed the change the password.
ue_pwage.maxage	The maximum age in seconds the password may have. If the difference between ue_pwage.time and the current time exceeds this value, the user will be required to change the password.
ue_pwage.minage	The minimum amount of time that must elapse before a password may be changed.

ue_parentuid
UID of the group administrator (used by xadmin(8)).

ue_adminmax Reserved for future use.

ue_jpelimit Per-job maximum number of massively parallel processing (MPP) processing elements (PEs). Two values exist, one for batch and the other for interactive work; udbgen(8) and udbsee(1) present the values as jpelimit[b] for batch and jpelimit[i] for interactive. A value of 0 in this field makes the MPP system inaccessible by the job. This field is present but not used on Cray Research systems without an MPP system.

`ue_jmpptime` Per-job maximum amount of wall-clock time (in seconds) that the MPP system can be reserved by the job. Two values exist, one for batch and the other for interactive work; `udbgen`(8) and `udbsee`(1) present the values as `jmpptime[b]` for batch and `jmpptime[i]` for interactive. A value of 0 in this field makes the MPP system inaccessible by the job. This field is present but not used on Cray Research systems without an MPP system.

`ue_jmppbarrier`
Per-job maximum number of MPP barriers. Two values exist, one for batch and the other for interactive work; `udbgen`(8) and `udbsee`(1) present the values as `jmppbarrier[b]` for batch and `jmppbarrier[i]` for interactive. This field is present but not used on Cray Research systems without an MPP system.

`ue_pmpptime` Per-process maximum amount of wall-clock time (in seconds) that the MPP system can be reserved by the process. Two values exist, one for batch and the other for interactive work; `udbgen`(8) and `udbsee`(1) present the values as `pmpptime[b]` for batch and `pmpptime[i]` for interactive. A value of 0 in this field makes the MPP system inaccessible by the process. This field is present but not used on Cray Research systems without an MPP system.

`ue_pcorelim` Per-process maximum core file limit in units of 512 words. This represents the maximum size of a core file that the process can create. If the size of the process is larger than this limit, a partial core file will be created. A partial core file contains just the user and user common structures. Two values exist, one for batch and the other for interactive work; `udbgen`(8) and `udbsee`(1) present the values as `pcorelim[b]` for batch and `pcorelim[i]` for interactive.

`ue_pfdlimit` Per-process maximum open file limit. This represents the maximum number of file descriptors that a process belonging to this user can allocate. This limit is restricted by the value of `OPEN_MAX` (64) at the low end, and the system configurable value of `K_OPEN_MAX` at the high end. Two values exist, one for batch and the other for interactive work; `udbgen`(8) and `udbsee`(1) present these values as `pfdlimit[b]` for batch and `pfdlimit[i]` interactive.

`ue_mincomps` Minimum security compartments. The field is actually a bit list but is represented externally as a comma-separated list of compartments. See `ue_comparts` for more information.

`ue_jshmsegs` Number of shared memory segments a job can create. This field always present but is used only on CRAY T90 series systems supporting shared memory.

`ue_jshmsize` Per-job maximum shared memory allocation in units of 512 words. This field always present but is used only on CRAY T90 series systems supporting shared memory.

`ue_jsocbflim`
Per-job maximum amount of socket buffer space in units of 512 words. If this field is 0, an unlimited amount of socket buffer space is allowed.

The ue_cpu_quota and ue_cpu_quota_used fields hold values expressed in seconds * 10. Externally, udbgen(8) and udbsee(1) accept and display these fields in the form *vvv.v* (seconds and tenths of a second). The internal form is used so that tenth-of-second resolution can be maintained in a 32-bit field. The maximum value allowed in these fields is 4294967295, which limits the maximum CPU quota to 429,496,729.5 seconds.

Description of struct udbstat

```
struct udbstat {
    int       add;              number of add record requests
    int       dbhits;           number of data block rereads
    int       dbread;           number of data blocks read
    int       dbwrite;          number of data blocks written
    int       delete;           number of delete by name requests
    int       getnam;           number of get by name requests
    int       getudb;           number of sequential read requests
    int       getudbchain;      number of chain reads
    int       getuid;           number of get by uid requests
    int       hdread;           number of header block reads
    int       hdwrite;          number of header block writes
    int       lock;             number of lock requests
    int       maxuid;           highest value UID in the database
    int       nhread;           number of name hash blocks read
    int       nhwrite;          number of name hash blocks written
    int       rewrite;          number of rewrite requests
    int       uhread;           number of uid hash blocks read
    int       uhwrite;          number of uid hash blocks written
    int       unlock;           number of unlock requests
    int       version;          database version number
    int       maxrecs;          maximum number of records in the udb
};
```

Description of `struct udbdefault`

```
struct udbdefault {
    long      jcpulim[MAXUE_RCLASS];     default job CPU limit
    long      pcpulim[MAXUE_RCLASS];     default process CPU limit
    long      jmemlim[MAXUE_RCLASS];     default job memory limit
    long      pmemlim[MAXUE_RCLASS];     default process memory limit
    long      jsdslim[MAXUE_RCLASS];     default job SDS limit
    long      psdslim[MAXUE_RCLASS];     default process SDS limit
    long      jfilelim[MAXUE_RCLASS];    default job new file space allocation limit
    long      pfilelim[MAXUE_RCLASS];    default process new file space allocation limit
    unsigned char                        default tape limits
              jtapelim[MAXUE_RCLASS][MAXUE_TAPETYPE];
    int       nice[MAXUE_RCLASS];        default nice value
    int       jproclim[MAXUE_RCLASS];    default job process limit
    int       jpelimit[MAXUE_RCLASS];    default MPP PE limit
    int       jmpptime[MAXUE_RCLASS];    default MPP time limit
    int       jmppbarrier[MAXUE_RCLASS]; default MPP barrier limit
    int       pmpptime[MAXUE_RCLASS];    default per process MPP time limit
    int       pcorelim[MAXUE_RCLASS];    default per proc core file limit [512 words]
    int       pfdlimit[MAXUE_RCLASS];    default file descriptor limit
    int       jshmsegs[MAXUE_RCLASS];    default created shared memory segments limit
    int       jshmsize[MAXUE_RCLASS];    default job shared memory size limit
    int       jsocbflim[MAXUE_RCLASS];   default per job max socket buffer limit [512 words]
};
```

Description of `struct udbtmap`

```
struct udbtmap {
       struct {
    char      name[MAXUE_TNAME + 1];     tape name
       } mt_entry[MAXUE_TAPETYPE];
};
```

NOTES

The external representation of the records in the UDB is located in the files `libc/udb/uentrydb.c` and `libc/udb/libudb.h`. To save space in the files, the data is packed in `uentrydb.c`, using the structure defined in that file. In the extension files, data is tagged and compressed; all zero-valued fields are discarded. Transformation functions in the library convert between the file and `struct udb` representations.

In the previously described `struct udb`, the following fields are included only for compatibility with previous releases: `ue_shplimit`, `ue_shmlimit`, `ue_mask`, `ue_warnings`, `ue_reason`, `ue_age`, and `ue_limflags`. These fields are obsolete and are not referenced by the UDB.

WARNINGS

Successive calls to function getudbnam return a pointer to the same static memory space each time they are called; these calls overwrite the same data area. Use caution when working with more than one UDB structure at a time.

Most functions in this library leave the udb file open to assure reasonable performance for multiple calls. If it is important that the program in which the calls are made can be restarted, an endudb() call must be made to close the udb file after the access is complete.

RETURN VALUES

Successful calls to type int functions return with the value UDB_SUCCESS (0); unsuccessful calls return UDB_FAIL (–1). Structure pointer type requests return with a pointer to a structure if successful, or the pointer value UDB_NULL ((struct udb *) 0) if they fail (except for some options of getudbchain). In any failing case, the extern int udb_errno variable contains a reason code from the following list. The descriptions of some reason codes state that further information for the failure can be found in the errno file (see header errno.h).

Symbol	Description
UDBERR_BADPATH	Return value 1. Path name specified in the setudbpath( ) call is bad (see errno.h(3C)).
UDBERR_BADUFN	Return value 2. A UDB file name is illegal. UDB or UDBPUB is zero length, it ends with '/', or the full name is longer than UDBFNAMELEN.
UDBERR_CHANGED	Return value 3. Another user changed the database while getudbchain( ) was being used. The current position is lost.
UDBERR_CORRUPT	Return value 4. Something is wrong with the content of the database. The files must be regenerated.
UDBERR_CREATE	Return value 5. Error in creating the database (see errno.h(3C)).
UDBERR_DEADLK	Return value 6. Protected user database could not be locked, because a deadlock condition would have resulted.
UDBERR_END	Return value 7. No more records in database.
UDBERR_IDCHG	Return value 8. An update must be performed as a deleteudb( ) and addudb( ) sequence because the name or UID has changed.
UDBERR_INTERR	Return value 9. An internal program error occurred.
UDBERR_IOERR	Return value 10. An I/O error occurred (see errno.h(3C)).
UDBERR_NAMEINUSE	Return value 11. The user-name of the record to be created is already in use.

`UDBERR_NOLCK` Return value 12. Protected user database could not be locked, because some privileged process has the protected user database locked by a means other than the `lockudb()` call. If this occurs, an error exists in some non-kernel (user-level) system software.

`UDBERR_NOPOS` Return value 13. Undefined current position.

`UDBERR_NORW` Return value 14. Caller does not have read or write access on a protected database.

`UDBERR_NOSUCHUSER` Return value 15. No such record in the database.

`UDBERR_NOTLOCKED` Return value 16. User information cannot be rewritten because the database was not locked.

`UDBERR_UDBCHAIN` Return value 17. The call to `getudbchain()` has an unknown option code.

`UDBERR_VERSION` Return value 18. The software level of the database access functions linked with the caller is incompatible with the current version of the UDB.

`UDBERR_BADDEFER` Return value 19. The `DEFERTORESGRP` flag was set on more than four chained entries.

FILES

`/etc/udb`	User validation file containing user control limits
`/etc/udb.public`	Public version of the user database file
`/etc/udb_2/udb.index`	Public extention file index
`/etc/udb_2/udb.priva`	Private field extension file
`/etc/udb_2/udb.pubva`	Public field extension file
`/etc/passwd`	Traditional password file

Other path names can be used if `setudbpath` has been called.

SEE ALSO

`errno.h`(3C), `getpwent`(3C), `perror`(3C)

`chacid`(1), `guest`(1), `login`(1), `newacct`(1), `nice`(1), `quota`(1), `udbsee`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`acctid`(2), `chacid`(2), `getpermit`(2), `limits`(2), `mknod`(2), `mount`(2), `msgget`(2), `nice`(2), `plock`(2), `semget`(2), `setgid`(2), `setgroups`(2), `setuid`(2), `shmget`(2), `ulimit`(2), `umount`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`cpu`(4), `udb`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`accton`(8), `chroot`(8), `csaswitch`(8), `ddms`(8), `diskusg`(8), `tplabel`(8), `udbgen`(8), `xadmin`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

limits.h – Library header for integral type limits

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

TYPES

None

MACROS

The header limits.h defines the various machine-dependent numerical limits for Cray Research systems. Each of these limits expands into constant expressions suitable for use in a #if preprocessing directive. The macros and definitions are shown in the following table:

Macro	Standard	Definition
CHAR_BIT	ISO/ANSI	Number of bits for smallest object that is not a bit-field (byte).
_WORD_BIT	CRI	Number of bits in a word.
SCHAR_MIN	ISO/ANSI	Minimum value for an object of type signed char.
SCHAR_MAX	ISO/ANSI	Maximum value for an object of type signed char.
UCHAR_MAX	ISO/ANSI	Maximum value for an object of type unsigned char.
CHAR_MIN	ISO/ANSI	Minimum value for an object of type char.
CHAR_MAX	ISO/ANSI	Maximum value for an object of type char.
MB_LEN_MAX	ISO/ANSI	Maximum number of bytes in a multibyte character, for any supported locale.
SHRT_MIN	ISO/ANSI	Minimum value for an object of type short int.
SHRT_MAX	ISO/ANSI	Maximum value for an object of type short int.
USHRT_MAX	ISO/ANSI	Maximum value for an object of type unsigned short int.
INT_MIN	ISO/ANSI	Minimum value for an object of type int.
INT_MAX	ISO/ANSI	Maximum value for an object of type int.
UINT_MAX	ISO/ANSI	Maximum value for an object of type unsigned int.
LONG_MIN	ISO/ANSI	Minimum value for an object of type long int.
LONG_MAX	ISO/ANSI	Maximum value for an object of type long int.
ULONG_MAX	ISO/ANSI	Maximum value for an object of type unsigned long int.

The values in limits.h are as shown in the following table. For comparison, the "CRI Value" column in the table is followed by a column listing the minimum value (in magnitude) required by the standard.

Macro	CRI Value	Minimum for ISO/ANSI C
CHAR_BIT	8	8
_WORD_BIT†	64	–
SCHAR_MIN	–128	–127
SCHAR_MAX	127	127
UCHAR_MAX	255	255
CHAR_MIN	0	–
CHAR_MAX	255	–
MB_LEN_MAX	1	1
SHRT_MIN	–8388608 (24-bit A register) –2147483648 (32-bit A register)	–32767
SHRT_MAX	8388607 (24-bit A register) 2147483647 (32-bit A register)	32767
USHRT_MAX	16777215 (24-bit A register) 4294967295 (32-bit A register)	65535
INT_MIN	–35184372088832 (default) –9223372036854775808 (-h nofastmd)	–32767
INT_MAX	35184372088831 (default) 9223372036854775807 (-h nofastmd)	32767
UINT_MAX	18446744073709551615	65535
LONG_MIN	–9223372036854775808	–2147483647
LONG_MAX	9223372036854775807	2147483647
ULONG_MAX	18446744073709551615	4294967295

† _WORD_BIT is a CRI extension; not specified by the standard

See the *Cray Standard C Reference Manual*, Cray Research publication SR–2074, for information about the -h fastmd and -h nofastmd command line options. (The -h fastmd command-line option is the compiler default.)

FUNCTION DECLARATIONS

None

CAUTIONS

On CRI systems, comparisons between two integer values is done by subtraction. That is, the expression (`A > B`) is evaluated by performing the operation (`A - B`) and testing the sign of the result. If, however, `A` and `B` are signed variables with different signs and either `A` or `B` is greater than (2**62–1) or less than –(2**62), integer overflow can occur and the sign of the result may be incorrect. For this reason, it is not safe to use the values `LONG_MAX` or `LONG_MIN` as an arbitrary large number with the relational operators. Instead, pick a smaller number; `LONG_MAX`/2 is sufficiently small. `INT_MAX` in the absence of the `-h nofastmd` command-line option is also safe. Comparison of unsigned integer values is always safe.

SEE ALSO

`float.h`(3C), `values.h`(3C)

NAME

`loaded`, `loaded_data` – Tells whether soft external routine/data is loaded

SYNOPSIS

```
#include <infoblk.h>
int loaded ( );
int loaded_data ( );
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The `loaded` function tells whether a soft-referenced function has been loaded into a user program. The `loaded_data` function tells whether a soft-referenced data item has been loaded into a user program. Both take as arguments the address of the specified function or data item.

RETURN VALUES

The `loaded` and `loaded_data` functions return 1 if the specified function or data item has been loaded into the user program; if it has not been loaded, they return 0.

EXAMPLES

The following example shows how `loaded` and `loaded_data` execute:

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

#pragma _CRI soft data, func
extern int data;
extern int func(void);

main()
{
      if (loaded_data(&data))
            printf("data loaded; value %d\n", data);
      else
            printf("data NOT loaded\n");
      if (loaded(func))
            printf("func loaded; returns %d\n", func());
      else
            printf("func NOT loaded\n");
}
```

NAME

`locale` – Introduction to locale information functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The locale information functions and header file `locale.h` provide various means for setting and accessing a specific set of run-time environment variables that may vary with culture, geography, or other factors related to location. The set of locale information variables affects the following:

- Formatting of monetary information
- Representation of date and time
- Classification of characters
- Collation of characters and character strings
- Formatting of numeric values

The ANSI standard defines one structure type, `struct lconv`, which contains members related to the formatting of numeric values. These members are as follows:

`char *decimal_point`
The decimal-point used to format nonmonetary quantities.

`char *thousands_sep`
The characters used to separate groups of digits before the decimal-point character in formatted nonmonetary quantities.

`char *grouping`
The characters that indicate the size of each group of digits in formatted nonmonetary quantities.

The elements of `grouping` are interpreted according to the following:

`CHAR_MAX` No further grouping will be performed.

`0` The previous element will be used repeatedly for the remainder of the digits.

other The integer value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits before the current group.

`char *int_curr_symbol`
The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in ISO 4217 Codes for the Representation of Currency and Funds. The fourth character (immediately preceding the null character) is the character used to separate the international currency symbol from the monetary quantity.

`char *currency_symbol`
The local currency symbol applicable to the current locale.

`char *mon_decimal_point`
The decimal-point used to format monetary quantities.

`char *mon_thousands_sep`
The separator for groups of digits before the decimal-point in formatted monetary quantities.

`char *mon_grouping`
A string whose elements indicate the size of each group of digits in formatted monetary quantities.

The elements of `mon_grouping` are interpreted according to the following:

`CHAR_MAX` No further grouping will be performed.

`0` The previous element will be used repeatedly for the remainder of the digits.

other The integer value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits before the current group.

`char *positive_sign`
The string used to indicate a nonnegative-valued formatted monetary quantity.

`char *negative_sign`
The string used to indicate a negative-valued formatted monetary quantity.

`char int_frac_digits`
The number of fractional digits (those after the decimal-point) to be displayed in a internationally formatted monetary quantity.

`char frac_digits`
The number of fractional digits (those after the decimal-point) to be displayed in a formatted monetary quantity.

`char p_cs_precedes`
Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a nonnegative formatted monetary quantity.

`char p_sep_by_space`
Set to 1 or 0 if the `currency_symbol` respectively is or is not separated by a space from the value for a nonnegative formatted monetary quantity.

`char n_cs_precedes`
Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a negative formatted monetary quantity.

`char n_sep_by_space`
Set to 1 or 0 if the `currency_symbol` respectively is or is not separated by a space from the value for a negative formatted monetary quantity.

char p_sign_posn

Set to a value that indicates the positioning of the positive_sign for a nonnegative formatted monetary quantity. (See the note following the n_sign_posn descriptions.)

char n_sign_posn

Set to a value that indicates the positioning of the negative_sign for a negative formatted monetary quantity.

The value of p_sign_posn and n_sign_posn is interpreted according to the following:

0 Parentheses surround the quantity and currency_symbol.

1 The sign string precedes the quantity and currency_symbol.

2 The sign string succeeds the quantity and currency_symbol.

3 The sign string immediately precedes the currency_symbol.

4 The sign string immediately succeeds the currency_symbol.

The members of the structure with type char * are pointers to strings, any of which (except decimal_point) can point to "" to indicate that the value is not available in the current locale or is of 0 length. The members with type char are nonnegative numbers; to indicate that the value is not available in the current locale, any members can be CHAR_MAX.

The method by which users defined their own locales (described in previous releases on this man page) is no longer supported. It is no longer necessary because the localedef command provides a superset of this functionality. If you use the old method and try to compile a program to generate a locale, the program will not compile. However, existing binary files that create locales will work through the UNICOS 9.0 release. Any successfully generated locale files will continue to be accepted by setlocale(3C).

Associated Headers

<locale.h>

Associated Functions

Function	Description
iconv(3C), iconv_close(3C), iconv_open(3C)	Converts a sequence of characters from one codeset into another codeset
localeconv(3C)	Reports program's numeric formatting conventions
nl_langinfo(3C)	Points to language information.
setlocale(3C)	Selects program's locale

NOTES

The UNICOS C library functions that set or access these variables include the following: localeconv(3C), nl_langinfo(3C), printf(3C), scanf(3C), strcoll(3C), strfmon(3C), strftime(3C), strptime(3C), strxfrm(3C), wcscoll(3C), wcsxfrm(3C), and the character-handling functions (see character(3C)).

NAME

`localeconv` – Reports program's numeric formatting conventions

SYNOPSIS

```
#include <locale.h>
struct lconv *localeconv (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The `localeconv` function sets the components of an object with type `struct lconv` with values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale.

RETURN VALUES

The `localeconv` function returns a pointer to the filled-in object. The structure pointed to by the return value cannot be modified by the program, but may be overwritten by a subsequent call to `localeconv`. In addition, calls to `setlocale` with categories `LC_ALL`, `LC_MONETARY`, or `LC_NUMERIC` can overwrite the contents of the structure.

SEE ALSO

`locale`(3C), `locale.h`(3C), `setlocale`(3C)

NAME

locale.h – Library header for locale information functions

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The header file locale.h defines locale information functions.

Types

The types declared in locale.h are as follows:

Type	Description
struct lconv	Structure that contains members related to the formatting of numeric values. This conforms to the ISO/ANSI standard.

Macros

The macros defined in the header file locale.h are as follows (unless noted, macros conform to the ISO/ANSI standard):

Macro	Description
LC_ALL LC_COLLATE LC_CTYPE LC_MONETARY LC_NUMERIC LC_TIME	Each of these macros expands to an integral constant expression with distinct values, and is suitable for use as the first argument to the setlocale(3C) function.
LC_MESSAGES	Same as above. Conforms with POSIX P1003.2.
NULL	An implementation-defined null pointer constant, equal to 0 on Cray Research systems.

Function Declarations

The localeconv and setlocale functions are declared in the header file locale.h.

SEE ALSO

ctype.h(3C), locale(3C)

NAME

LOCKASGN – Identifies an integer variable intended for use as a lock

SYNOPSIS

CALL LOCKASGN(*name* [,*value*])

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

LOCKASGN identifies an integer variable that the program intends to use as a lock. The program must call the LOCKASGN routine for each lock variable before it is used with any other lock routines. The multitasking library gives a lock an initial state of off or cleared. A data statement can initialize the lock to the value in the optional argument, allowing the program to assign it in a routine. The first call assigns the lock, and further calls are ignored.

The following is a list of valid variables for this routine:

Argument	**Description**
name	Name of an integer variable to be used as a lock. The library stores an identifier into this variable; do not modify this variable after the call to LOCKASGN.
value	The initial integer value of the lock variable. An identifier should be stored into the variable only if it contains the value. If *value* is not specified, an identifier is stored into the variable unconditionally.

CAUTIONS

For SPARC systems, the *value* parameter is optional, and LOCKASGN is not predeclared (not intrinsic). Therefore, if a call is made to it with only the *name* parameter, LOCKASGN must be declared with an INTERFACE block in the calling module.

EXAMPLES

```
      PROGRAM MULTI
      INTEGER LKINPUT,LKOUTPUT,LKCALL
      REAL    INDATA(20000),OUTDATA(20000)
      COMMON  /CBINPUT/  LKINPUT,INDATA
      COMMON  /CBOUTPUT/ LKOUTPUT,OUTDATA
      COMMON  /MISC/     LKCALL
C     ...
      CALL LOCKASGN (LKINPUT)
      CALL LOCKASGN (LKOUTPUT)
      CALL LOCKASGN (LKCALL)
C     ...
      END

      SUBROUTINE SUB1
      COMMON /LOCK1/ LOCK1
      DATA LOCK1 /-1/
C     ...
      CALL LOCKASGN (LOCK1,-1)
C     ...
      END
```

NAME

lockf – Provides record locking on files

SYNOPSIS

```
#include <unistd.h>
int lockf (int fildes, int function, long size);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

AT&T extension

DESCRIPTION

The lockf function allows sections of a file to be locked with advisory or mandatory write locks, depending on the mode bits of the file (see chmod(2)). Locking calls from other processes that attempt to lock the locked file section either return an error value or are put to sleep until the resource becomes unlocked. All locks for a process are removed when the process terminates. (See fcntl(2) for more information about record locking.) The lockf function does not work on NFS files.

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

The *function* operand is a control value that specifies the action to be taken. Permissible values for *function* are defined in the header file 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 used.

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

The *size* operand is the number of contiguous bytes to be locked or unlocked. The resource to be locked starts at the current offset in the file; it extends forward for a positive size and backward for a negative size (the preceding bytes up to but not including the current offset). If *size* is 0, 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 to be locked, because such locks can exist past the end-of-file.

The sections locked with F_LOCK or F_TLOCK can, 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 are locked, 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 can, 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 that controls a locked resource is put to sleep by accessing another process’s locked resource. Thus, calls to lockf or fcntl scan for a deadlock before sleeping on a locked resource. An error return is made if sleeping on the locked resource would cause a deadlock.

Sleeping on a resource is interrupted with any signal. The alarm(2) system call can be used to provide a time-out facility in applications that require this facility.

If the lockf function fails, errno is set to one of the following values, defined in the header file errno.h:

Error Code	**Description**
EBADF	File descriptor *fildes* is not a valid open descriptor.
EACCES	*Cmd* is F_TLOCK or F_TEST, and the section is already locked by another process.
EDEADLK	*Cmd* is F_LOCK, and a deadlock would occur. Also the *cmd* is F_LOCK, F_TLOCK, or F_ULOCK, and the number of entries in the lock table exceeds the number allocated on the system.
ECOMM	File descriptor *fildes* is on a remote machine, and the link to that machine is no longer active.

WARNINGS

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

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

RETURN VALUES

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

SEE ALSO

alarm(2), chmod(2), close(2), creat(2), fcntl(2), intro(2), open(2), read(2), write(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

LOCKOFF – Clears a lock and returns control to the calling task

SYNOPSIS

CALL LOCKOFF(*lock*)

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

LOCKOFF clears a lock and returns control to the calling task. Clearing the lock can allow another task to resume execution, but this is transparent to the task calling LOCKOFF.

The following is a valid argument for this routine:

Argument	**Description**
lock	Name of an integer variable used as a lock.

EXAMPLES

```
      PROGRAM MULTI
      INTEGER LKOUTPUT
      REAL    OUTDATA(20000)
      COMMON  /CBOUTPUT/ LKOUTPUT,OUTDATA
C     ...
      CALL LOCKASGN (LKOUTPUT)
C     ...
C
      CALL LOCKON (LKOUTPUT)
      DO 100 I=1,20000
        OUTDATA(I)=MAX(OUTDATA(I),0)
 100  CONTINUE
      CALL LOCKOFF (LKOUTPUT)
C
C     ...
      END
```

SEE ALSO

LOCKON(3F), LOCKTEST(3F), multif(3F), NLOCKOFF(3F), NLOCKON(3F),

NAME

LOCKON – Sets a lock and returns control to the calling task

SYNOPSIS

CALL LOCKON(*lock*)

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

LOCKON sets a lock and returns control to the calling task. If the lock is already set when LOCKON is called, the task calling LOCKON waits until the lock is cleared by another task and then sets it. This means that placing LOCKON before a critical region ensures that the code in the region is executed only when the task has unique access to the lock. Calls to LOCKON cannot be nested.

The following is a valid argument for this routine:

Argument	**Description**
lock	Name of an integer variable used as a lock.

SEE ALSO

LOCKOFF(3F), LOCKTEST(3F), multif(3F), NLOCKOFF(3F), NLOCKON(3F)

NAME

LOCKREL – Releases the identifier assigned to a lock

SYNOPSIS

CALL LOCKREL(*name*)

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

LOCKREL releases the identifier assigned to a lock.

The following is a valid argument for this routine:

Argument	**Description**
name	Name of an integer variable used as a lock.

If the lock is set or a task is waiting for the lock when LOCKREL is called, an error results. This routine detects some errors that arise when a task is waiting for a lock that is never cleared. The lock variable can be reused following another call to LOCKASGN(3F).

EXAMPLES

```
      PROGRAM MULTI
      INTEGER LKOUTPUT
      REAL    INDATA(20000),OUTDATA(20000)
      COMMON  /CBOUTPUT/ LKOUTPUT,OUTDATA
C     ...
      CALL LOCKASGN (LKOUTPUT)
C     ...
C
      CALL LOCKON (LKOUTPUT)
      DO 100 I=1,20000
        OUTDATA(I)=MAX(OUTDATA(I),0)
 100  CONTINUE
      CALL LOCKOFF (LKOUTPUT)
C     ...
      CALL LOCKREL (LKOUTPUT)
C
      END
```

SEE ALSO

LOCKASGN(3F)

NAME

LOCKTEST – Tests a lock to determine its state (locked or unlocked)

SYNOPSIS

LOGICAL LOCKTEST
return = LOCKTEST(*lock*)

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

LOCKTEST tests a lock to determine its state. By using this function, a task can avoid blocking on a set lock.

The following is a list of valid variables for this routine:

Arguemnt	**Description**
return	A logical .TRUE. if the lock was originally set. A logical .FALSE. if the lock was originally clear. The lock variable's state is always set to locked upon return.
name	Name of an integer variable used as a lock.

Unlike a task using LOCKON(3F), the task does not wait if the lock is already locked. A task using LOCKTEST must always test the return value before continuing.

NOTES

LOCKTEST and *return* must be declared type LOGICAL in the calling module.

SEE ALSO

multif(3F), LOCKOFF(3F), LOCKON(3F), NLOCKOFF(3F), NLOCKON(3F)

NAME

`logb`, `logbf`, `logbl` – Returns the signed exponent of its argument

SYNOPSIS

CRAY T90 systems with IEEE floating-point hardware:

```
#include <fp.h>
```

```
double logb(double x);
float logbf(float x);
long double logbl(long double x);
```

Cray MPP systems:

```
#include <fp.h>
```

```
double logb(double x);
```

IMPLEMENTATION

Cray MPP systems (implemented as a macro)
CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

ANSI/IEEE Std 754-1985
X3/TR-17:199x

DESCRIPTION

The `logb` function or macro extracts the exponent of *x* as a signed integral value in the format of *x*. If *x* is subnormal, it is treated as though it were normalized; thus, for positive finite *x*, the following is true:

$$1 \leq x * \mathtt{FLT\_RADIX}^{-\mathrm{logb}(x)} < \mathtt{FLT\_RADIX}$$

RETURN VALUES

Each function or macro returns the signed exponent of its argument.

SEE ALSO

`float.h`(3C) for a description of the `FLT_RADIX` macro

Migrating to the CRAY T90 Series IEEE Floating Point, Cray Research publication SN–2194

NAME

`logname` – Returns the login name of the user

SYNOPSIS

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

```
char *logname (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

AT&T extension

DESCRIPTION

The `logname` function returns a pointer to the null-terminated login name of the user; it extracts the `$LOGNAME` variable from the user's environment.

CAUTIONS

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

FILES

`/etc/profile`	Systemwide shell start-up file

SEE ALSO

`env`(1), `login`(1), `sh`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`profile`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

`lsearch`, `lfind` – Performs a linear search and update

SYNOPSIS

```
#include <search.h>

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

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The `lsearch` function performs linear searches. It returns a pointer into a table that indicates where data may be found. If data does not occur, it is added at the end of the table. The *key* argument points to the data to be sought in the table. The *base* argument points to the first element in the table. The integer to which *nelp* points contains the current number of elements in the table. (This integer is incremented if the data is added to the table.) The value of *width* is the size in bytes of an element. You must supply the name of the comparison function, *compar* (for example, `strcmp`). It is called with two arguments that point to the elements being compared. If the elements are equal, the function must return 0; otherwise, it returns nonzero.

The `lfind` function is the same as `lsearch` except that if the data is not found, it is not added to the table. Instead, a null pointer is returned.

NOTES

The pointers to the key and the element at the base of the table may be pointers to any type.

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

The value required should be cast into type pointer-to-element.

CAUTIONS

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

RETURN VALUES

If the searched for data 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.

EXAMPLES

The following fragment reads in ≤ TABSIZE strings of length ≤ ELSIZE and stores them in a table, eliminating duplicates:

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

#define TABSIZE 50
#define ELSIZE 120

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

SEE ALSO

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

NAME

malloc, calloc, free, realloc, malloc_inplace, malloc_expand, malloc_extend, malloc_howbig, malloc_isvalid, malloc_space, malloc_brk, malloc_limit, malloc_check, malloc_stats, malloc_tron, malloc_troff, malloc_etrace, malloc_dtrace, mallopt, mallinfo, malloc_error – Memory management functions

SYNOPSIS

```
#include <stdlib.h> or #include <malloc.h>

void *malloc (size_t size);

void *calloc (size_t nmemb, size_t size);

void free (void *ptr);

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

#include <malloc.h>

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

size_t malloc_expand (void *ptr);

size_t malloc_extend (void *ptr);

size_t malloc_howbig (void *ptr);

int malloc_isvalid (void *ptr);

size_t malloc_space (long nbytes);

int malloc_brk (void *endds);

void malloc_limit (size_t thresh, size_t limit);

int malloc_check (int level);

void malloc_stats (int level);

void malloc_tron (void);

void malloc_troff (void);

void malloc_etrace (long funcs);

void malloc_dtrace (long funcs);

int mallopt (int cmd, int value);

struct mallinfo mallinfo (void);

extern long malloc_error;
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (`malloc`, `calloc`, `free`, and `realloc` only)

AT&T extension (`mallopt` and `mallinfo` only)

CRI extension (all others)

DESCRIPTION

The `malloc` function returns a pointer to a block of at least *size* bytes suitably aligned for any use. `malloc` calls `sbreak` (see `brk`(2)) to get more memory from the system when no suitable space is already free. The space returned is left uninitialized.

The `calloc` function allocates space for an array of *nmemb* objects, each of whose size is *size* bytes. The space is initialized to all bits 0.

The `free` function causes the block to which *ptr* points to be deallocated, that is, made available for further allocation. If *ptr* is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by a memory manager function, or if the space has already been deallocated by a call to `free` or `realloc`, `malloc_error` is set to indicate the error, and `free` returns.

The `realloc` function changes the size of the block to which *ptr* points to the size (in bytes) specified by *size*. The contents of the block are unchanged up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion of the block is indeterminate. If *ptr* is a null pointer, the `realloc` function behaves like the `malloc` function for the specified size. If *size* is 0 and *ptr* is not a null pointer, the block to which it points is freed. Otherwise, if *ptr* does not match a pointer earlier returned by a memory manager function, or if the space has been deallocated by a call to the `free` or `realloc` function, the `malloc_error` variable is set to indicate the error, and `realloc` returns a null pointer. If the space cannot be allocated, the block to which *ptr* points is unchanged.

The `malloc_inplace` function tries to change the size of the block to which *ptr* points to the size (in bytes) specified by *size*. However, if the size of the block cannot be changed without moving the block, the request fails and a null pointer is returned.

The `malloc_expand` function causes the memory block to which *ptr* points to grow as large as possible, without causing an `sbreak` or moving the block. It returns the new size of the block in bytes.

The `malloc_extend` function returns the expanded size of the block to which *ptr* points without actually doing the expansion. If the block is at the end of memory, a very large number is returned.

The `malloc_howbig` function returns the current size (in bytes) of the block to which *ptr* points (which may not be the same as the original *size* argument to `malloc`, et al.)

The malloc_isvalid function returns a nonzero value if *addr* points to a valid allocated block of memory used by the memory manager.

The malloc_space function tries to return up to *nbytes* bytes of memory to the system (only possible if the last block in the heap is free). If *nbytes* is –1, it returns as much memory as possible. If *nbytes* is 0, it returns the number of bytes of memory that could be freed.

The malloc_brk function extends the end of the heap to the address specified in *endds*. If *endds* is already contained within the heap, malloc_brk does nothing. Any new space created is turned into a free block. If the heap cannot be extended to the desired address, malloc_brk returns –1; otherwise, it returns 0.

The malloc_limit function controls the behavior of free when the last block in the heap is freed. If the *threshold* argument is nonzero, and the last block in the arena is free and larger than *threshold*, all but the last *limit* number of *nbytes* of that block are returned automatically to the system. The minimum positive value for *threshold* (512 Kwords) is silently enforced. Initially, *threshold* and *limit* are 0.

The malloc_check function checks the consistency of malloc's memory structure. If *level* is less than 0, malloc_check silently performs validation of the heap, and returns 0 if the heap is consistent, or nonzero if the heap has been corrupted. If *level* equals 0, malloc_check prints a message to stderr that describes the first inconsistency found. If *level* is greater than 0, malloc_check prints a line to stderr that describes each heap block in addition to checking the heap.

The malloc_stats function prints out memory manager statistics and heap block information to stdout. If *level* equals 0, malloc_stats reports the number of calls to each memory manager function, as well as summary statistics on the number and total size of the busy blocks, free blocks, and "spec" blocks (that is, blocks that are created by user calls to sbreak) in the heap. If *level* equals 1, malloc_stats prints a line with a * for each busy block, a . for each free block, and a @ for each "spec" block, in addition to the level 0 statistics. If *level* equals 2, malloc_stats prints a line describing each heap block, in addition to the level 0 statistics. The number of calls for each function are only available by linking with the libmalloc library; all of the other information is available in the default memory manager.

The malloc_tron, malloc_troff, malloc_etrace, and malloc_dtrace macros trace calls to the memory manager and to brk(2), sbrk(2), and sbreak(2); however, they are operational only by linking with the libmalloc library. When tracing is on, each memory manager function prints a line to stderr that shows the function called, its arguments, a one or two-level traceback, and the return value of the function. The malloc_tron and malloc_troff macros turn tracing on and off, respectively. Tracing is off by default. You can use the malloc_etrace and malloc_dtrace macros can be used to enable or disable, respectively, a given set of functions. For a list of the functions that can be traced, see the header file malloc.h. You may combine the function values using OR. The –1 constant refers to all functions.

The mallopt function allows you to set various options in the memory manager. The values for the *cmd* argument are defined in the header file malloc.h as follows (the values marked (libmalloc only) are operational only if the libmalloc library has been linked into the program):

Value	**Description**

M_TRACE (libmalloc only) If *value* ≥ 0, memory tracing is turned on, with the trace being written to file descriptor *value*. If *value* is less than 0, memory tracing is turned off.

M_ETRACE and M_DTRACE (libmalloc only) These use the malloc_etrace and malloc_dtrace macros, respectively, with *value* passed as an argument to them.

M_LIMIT and M_THRESH These set the free limit and threshold values, respectively, to *value* words (see malloc_limit).

M_BREAKSZ If *value* is greater than 0, any sbreak(2) calls from the memory manager are made in multiples of *value* words. If *value* equals 0, the heap is fixed to its current size, and malloc returns 0 rather than calling sbreak(2).

M_MEMCHK (libmalloc only) If *value* is 1, each call to a memory manager function checks the consistency of the heap; if the heap has been corrupted, it prints an error message to stderr and return an error status from the function called. If *value* is 2, the abort function is called rather than returning an error status (this flushes all open files and performs other cleanup actions before dumping core). If *value* is 3, an immediate core-dump is performed on detection of a corrupted heap. Setting *value* to 0 turns off heap consistency checking. If *value* is nonzero, the memory manager checks all calls to free, and prints an error message to stderr if an invalid pointer is passed as an argument.

M_LOWFIT If *value* is nonzero, the memory manager keeps the large block free lists sorted by address. This slows down the memory manager, but ensures that blocks are allocated from the lowest address possible (which keeps the heap as small as possible). This can be called anywhere in a program; the free lists will be sorted if they are in an unsorted state when mallopt is called. Setting the environment variable MEMLOWFIT to nonzero has the same effect.

M_INDEF If *value* is nonzero, calls to malloc and free initialize their blocks to the 'indef' pattern (this causes an operand range error if used as an address, or a floating-point exception if used as a floating-point number). You can also do this by setting the MEMINDEF environment variable to a nonzero number.

M_ABORT If *value* is nonzero, malloc prints an error message and calls abort if the program runs out of memory (that is, a call to sbreak(2) fails). Setting the environment variable MEMABORT to nonzero has the same effect as setting *value* to nonzero.

The mallinfo function provides information that describes space usage. It returns the structure mallinfo, which contains the following members:

```
    int arena;        /* total space in arena */
    int ordblks;      /* number of ordinary blocks */
    int smblks;       /* number of small blocks */
    int hblks;        /* number of holding blocks */
    int hblkhd;       /* space in holding block headers */
    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;     /* cost of enabling keep option (unused) */
```

The `malloc_error` variable is set if an error is encountered in the memory manager; it is never reset to 0. `malloc_error` may be set to the following values, defined in the header file `malloc.h`:

Value	**Description**
`ME_EXTEND`	Could not extend block in place
`ME_BADLEN`	Bad length supplied
`ME_NOMEM`	No memory available
`ME_BADADDR`	Address is outside bounds of heap
`ME_ISFREE`	A free block
`ME_NOTBLOCK`	Address is not a free/busy block
`ME_CORRUPT`	Corrupt memory arena
`ME_BREAK`	Arena truncated by user's `sbrk`(2)

NOTES

The `malloc` function uses a two-level allocation strategy for memory and time efficiency. Any requests to `malloc` larger than 64 bytes allocate a *large block*, which has a 2-word header. Free blocks also use the first 2 words of the block as free list pointers. All large blocks are on a doubly linked list, and there are 16 doubly linked free lists (hashed by size of the block). Requests to `malloc` smaller than 64 bytes allocate a large block of 4096+ bytes (called a *holding block*); from this block, many *small blocks* (which have a 1-word header) can be allocated and freed quickly. Different holding blocks are created when needed for different sizes of small blocks. Holding blocks are never freed, even if all of the small blocks within them have been freed.

A `free` of a large block causes any blocks immediately surrounding it to be coalesced into one free block. Each free list is unsorted, and the last block freed is put at the end of its corresponding free list (unless the `M_LOWFIT` option to `mallopt` is used).

The order of the algorithm that `realloc` (for large blocks) uses is as follows:

1. Coalesce any free block following the specified block.
2. Check if the block is at the end of memory, and use `sbreak`(2) to extend the block.
3. Check for a preceding free block, and slide the block lower in memory.
4. Use `malloc` to allocate a new block, and move the data to the new space.

ENVIRONMENT VARIABLES

You can use several environment variables to alter the behavior of the memory manager; their usage corresponds to the options for `mallopt`, as follows: The environment variables are:

Variable	**Description**
`MEMTRON=`*value*	(`libmalloc` only) If *value* is nonzero, the equivalent of `malloc_tron` is done.
`MEMCHK=`*value*	(`libmalloc` only) If *value* is nonzero, the equivalent of `mallopt(M_MEMCHK,` *value*`)` is done.
`MEMINDEF=`*value*	If *value* is nonzero, the equivalent of `mallopt(M_INDEF, 1)` is done.
`MEMABORT=`*value*	If *value* is nonzero, the equivalent of `mallopt(M_ABORT, 1)` is done.
`MEMLOWFIT=`*value*	If *value* is nonzero, the equivalent of `mallopt(M_LOWFIT, 1)` is done.

RETURN VALUES

The `malloc` and `calloc` functions return a pointer to the allocated space; otherwise, they return a null pointer (with `malloc_error` set).

The `free`, `malloc_limit`, and `malloc_stats` functions return no value.

The `realloc` and `malloc_inplace` functions return a pointer to the allocated space (which may have moved in the case of `realloc`); otherwise, they return a null pointer (with `malloc_error` set).

The `malloc_expand`, `malloc_extend`, and `malloc_howbig` functions return the size of the (possibly expanded) block; otherwise, they return 0 (with `malloc_error` set).

The `malloc_isvalid` function returns nonzero if pointing to a valid block; otherwise, it returns 0.

The `malloc_space` function returns the size of the space available to return to the system; otherwise, it returns 0.

The `malloc_brk` function returns –1 if the heap cannot be extended to the desired address; otherwise, it returns 0.

The `malloc_check` function returns nonzero if the heap is corrupt; otherwise, it returns 0.

The `malloc_tron`, `malloc_troff`, `malloc_etrace`, and `malloc_dtrace` macros return no value.

The `mallopt` function returns –1 if either *cmd* or *value* is invalid; otherwise it returns 0.

The `mallinfo` function returns a `mallinfo` structure, which describes the heap.

EXAMPLES

The following example turns memory tracing on:

```
malloc_tron();
```

The following example disables tracing for all functions but `malloc` and `free`:

```
malloc_dtrace(~(MF_MALLOC|MF_FREE));
```

The following example enables tracing for `realloc`:

```
malloc_etrace(MF_REALLOC);
```

The following example turns memory tracing off:

```
malloc_troff();
```

The following example links C and Fortran programs with `libmalloc`:

```
cc -oprog prog.c -lmalloc
cf77 -oprog prog.f -lmalloc
```

The following example runs programs with memory tracing and checking on:

```
env MEMTRON=1 MEMCHK=3 ./prog
```

SEE ALSO

`malloc.h`(3C)

`brk`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

malloc.h – Library header for memory allocation and management functions

IMPLEMENTATION

All Cray Research systems

STANDARDS

AT&T extension

TYPES

The types defined in header malloc.h are as follows:

Type	Standards	Description
struct mallinfo	AT&T	The structure type that is the type returned by the mallopt function. (See the description following this table.)
size_t	ISO/ANSI	The unsigned integral type of the result of the sizeof operator.

Structure mallinfo contains the following members:

```
int arena;        /* total space in arena */
int ordblks;      /* number of ordinary blocks */
int smblks;       /* number of small blocks */
int hblks;        /* number of holding blocks */
int hblkhd;       /* space in holding block headers */
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;     /* cost of enabling keep option */
```

MACROS

The header file malloc.h defines the following macros for use with the mallopt function (see mallopt(3C) for complete information):

```
M_MXFAST     M_NLBLKS     M_GRAIN      M_KEEP       M_TRACE
M_ETRACE     M_DTRACE     M_LIMIT      M_THRESH     M_BREAKSZ
M_MEMCHK     M_LOWFIT     M_INDEF      M_ABORT
```

The header file `malloc.h` defines the following function-like macros for use with the `libmalloc` debug library (see `malloc`(3C) for complete information):

```
malloc_tron     malloc_troff    malloc_dtrace   malloc_etrace
```

The header file `malloc.h` defines the following macros for use with the `malloc_etrace` and `malloc_dtrace` function-like macros (see `malloc`(3C) for complete information):

```
MF_MALLOC    MF_REALLOC   MF_FREE      MF_EXTEND    MF_INPLACE
MF_EXPAND    MF_HOWBIG    MF_CHECK     MF_ISVALID   MF_SPACE
MF_LIMIT     MF_SBREAK    MF__SBREAK   MF_BRK
```

The header file `malloc.h` defines the following macros, which describe the possible error values for the `malloc_error` variable (see `malloc`(3C) for complete information):

```
ME_EXTEND    ME_BADLEN    ME_NOMEM     ME_BADADDR   ME_ISFREE
ME_NOTBLOCK  ME_CORRUPT   ME_BREAK
```

FUNCTION DECLARATIONS

The following functions are declared in the header file `malloc.h`:

```
calloc          free            malloc          realloc
mallopt         mallinfo        malloc_check    malloc_inplace
malloc_expand   malloc_extend   malloc_howbig   malloc_isvalid
malloc_space    malloc_limit    malloc_stats    malloc_brk
```

OBJECT DECLARATIONS

The following object is declared in the header file `malloc.h`:

```
long malloc_error
```

NOTES

If only ISO/ANSI standard functions are used (i.e. `malloc`, `calloc`, `free`, `realloc`), the header file `stdlib.h` is preferred for use with Cray Standard C.

SEE ALSO

`stdlib.h`(3C)

NAME

`math` – Introduction to math functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The math functions provide various means for computing the results of common mathematical functions applied to specific arguments. The functions available include the common trigonometric and hyperbolic functions, exponential and logarithmic functions, and several others. Many more mathematical functions are available in the UNICOS math and scientific libraries, described in the λ8, and *Scientific Libraries Reference Manual*, Cray Research publication SR–2081.

Function Argument and Return Types

The arguments to and the values returned by these functions are all, with a few noted exceptions, floating types: `float`, `double`, `long double`, and `double complex`. In the presence of the function prototypes in the header file `math.h` or `complex.h`, if arguments of type integral are given where type `double` arguments are required, the compiler automatically promotes them to type `double`.

Double Complex and Long Double Functions

Math functions for `long double` and `double complex` values are provided. See `complex.h`(3C) and the *Cray Standard C Reference Manual*, Cray Research publication SR–2074, for more information.

Domain and Range Checking

For all functions, a *domain* error occurs if an input argument is outside the domain over which the mathematical function is defined. The description of each function describes the valid domain. Similarly, a *range* error occurs if the result of the function cannot be represented by the return type. The behavior of each of these mathematical functions when there is a domain or range error depends on the compilation mode; that is, the command-line options specified. Loops containing calls to these functions are candidates for vectorization, if compiled in extended mode.

When code containing calls to these functions is compiled by the Cray Standard C compiler in extended mode (the default), `errno` is not set on error and the functions do not return to the caller on error. If a domain error occurs, the program aborts with a run-time error. The reasons for this behavior being the default behavior are discussed in the next subsection, Fast Calling Sequence and Vector Functions. When calls to these functions are compiled in extended mode on CRAY T90 series systems with IEEE floating-point arithmetic, `errno` is not set on error. The functions do return to the caller on error; the return value for each function is documented on the corresponding man page in `libm` (see the *Intrinsic Procedures Reference Manual*, Cray Research publication SR–2138, or the online man page).

In strict conformance mode (specified by the cc(1) command-line option -hstdc), each function must execute as if it were a single operation, without generating any externally visible exceptions. This means that for those functions for which a domain or range error is possible, the function arguments are checked before the result is computed. If the value is such that it is outside the valid domain, the result is not computed; instead, errno is set to EDOM. If the value is such that the result of the function would overflow (the magnitude of the result is so large that it cannot be represented in an object of the specified type), the function returns the value of the macro HUGE_VAL with the same sign as the correct value of the function. If the value is such that the result of the function would overflow (that is, the magnitude of the result is so large that it cannot be represented in an object of the specified type), the function returns the value of the macro HUGE_VAL with the same sign as the correct value of the function. If the function returns a float, it returns the value of (+/-)HUGE_VALF. If the function returns a long double, it returns the value of (+/-)HUGE_VALL on Cray MPP systems and on CRAY T90 systems with IEEE arithmetic; on other Cray PVP machines, the function returns HUG_VAL for long double. The value of the macro ERANGE is stored in errno. On CRI systems, if the result underflows (the magnitude of the result is so small that it cannot be represented in an object of the specified type), the function returns 0; the integer expression errno does not acquire the value of the macro ERANGE, although it can in other implementations. (An exception to this rule is the function ldexp.)

In strict conformance mode, it is up to the calling function to set errno to 0 before the call and to check errno after the call to see if an error occurred. (See the prog_diag(3C) man page.)

Fast Calling Sequence and Vector Functions

Unfortunately, checking the arguments (and possibly setting errno) takes considerable time and prevents vectorization. For this reason, the Cray Standard C compiler offers a faster alternative. In extended mode (the default mode), argument checking is not done. The functions assume that the arguments are valid; if they are not, a program on any system (except the CRAY T90 system with IEEE floating-point arithmetic) aborts during computation, either because of a floating-point exception or because a low-level function detects the error. When calls to these functions are compiled in extended mode on CRAY T90 series systems with IEEE floating-point arithmetic, no error checking is done. The functions do return to the caller on a computation error; the return value for each function is documented on the corresponding man page in libm (see the *Intrinsic Procedures Reference Manual*, Cray Research publication SR–2138, or the online man page). In extended mode, if all the arguments are valid, the compiler generates code that uses the call-by-register calling sequence to call many of the standard math functions. Further, if the function call is in a vectorizable loop, the call will be made with vector arguments to vector versions of the functions. To compile in extended mode, do not specify the -hstdc option on the cc(1) command line (that is, extended mode is the default).

The slower mode of execution may be appropriate if you are not sure that the arguments are all valid. If you are sure about the validity of the arguments, the performance gain with the vector versions is significant.

ISO/ANSI Standard Reserved Names

All ISO/ANSI standard functions are reserved external names to the implementation. If you define an external with the same name as an ISO/ANSI library function, the behavior is undefined.

ASSOCIATED HEADERS

```
<complex.h>
<math.h>
```

ASSOCIATED FUNCTIONS

Function	Description
`acos`(3C)	Determines the arccosine of a `double` value (see `asin`(3C))
`acosl`(3C)	Determines the arccosine of a `long double` value (see `asin`(3C))
`asin`(3C)	Determines the arcsine of a `double` value
`asinl`(3C)	Determines the arcsine of a `long double` value (see `asin`(3C))
`atan`(3C)	Determines the arctangent of a `double` value (see `asin`(3C))
`atanl`(3C)	Determines the arctangent of a `long double` value (see `asin`(3C))
`atan2`(3C)	Determines the arctangent of a `double` value *x/y* (see `asin`(3C))
`atan2l`(3C)	Determines the arctangent of a `long double` *x/y* (see `asin`(3C))
`ccos`(3C)	Determines the cosine of a `double complex` value (see `sin`(3C))
`cos`(3C)	Determines the cosine of a `double` value (see `sin`(3C))
`cosl`(3C)	Determines the cosine of a `long double` value (see `sin`(3C))
`csin`(3C)	Determines the cosine of a `double complex` value (see `sin`(3C))
`hypot`(3C)	Determines the hypotenuse of a value (see `sqrt`(3C))
`sin`(3C)	Determines the sine of a `double` value
`sinl`(3C)	Determines the sine of a `long double` value (see `sin`(3C))
`tan`(3C)	Determines the tangent of a `double` value (see `sin`(3C))
`tanl`(3C)	Determines the tangent of a `long double` value (see `sin`(3C))

Hyperbolic Functions

Function	Description
`cosh`(3C)	Determines the hyperbolic cosine of a `double` value (see `sinh`(3C))
`coshl`(3C)	Determines the hyperbolic cosine of a `long double` value (see `sinh`(3C))
`sinh`(3C)	Determines the hyperbolic sine of a `double` value
`sinhl`(3C)	Determines the hyperbolic sine of a `long double` value (see `sinh`(3C))
`tanh`(3C)	Determines the hyperbolic tangent of a `double` value (see `sinh`(3C))
`tanhl`(3C)	Determines the hyperbolic tangent of a `long double` value (see `sinh`(3C))

Exponential and Logarithmic Functions

Function	Description
`cexp`(3C)	Determines the exponential for `double complex` values (see `exp`(3C))
`ldexp`(3C)	Multiplies a `double` floating-point number by an integral power
`ldexpl`(3C)	Multiplies a `long double` floating-point number by an integral power of 2 (see `frexp`(3C))
`exp`(3C)	Determines the exponential for `double` values
`expl`(3C)	Determines the exponential for `long double` values (see `exp`(3C))

`frexp`(3C)	Breaks a `double` floating-point number into a normalized fraction and an integral power of 2
`frexpl`(3C)	Breaks a `long double` floating-point number into a normalized fraction and an integral power of 2 (see `frexp`(3C))
`gamma`(3C)	Computes the log gamma function for double values
`log`(3C)	Determines the logarithm for `double` values (see `exp`(3C))
`logl`(3C)	Determines the logarithm for `long double` values (see `exp`(3C))
`clog`(3C)	Determines the logarithm for `double complex` values (see `exp`(3C))
`log10`(3C)	Determines the base 10 logarithm values for `double` (see `exp`(3C))
`log10l`(3C)	Determines the base 10 logarithm values for `long double` values (see `exp`(3C))
`modf`(3C)	Breaks the `double` argument *value* into integral and fractional parts (see `frexp`(3C))
`modfl`(3C)	Breaks the `long double` argument *value* into integral and fractional parts (see `frexp`(3C))
`pow`(3C)	Raises the specified `double` value to a given power
`powl`(3C))	Raises the specified `long double` value to a given power (see `pow`(3C))
`cpow`(3C)	Raises the specified `double complex` value to a given power (see `pow`(3C))
`sqrt`(3C)	Determines the square root of a `double` value
`sqrtl`(3C)	Determines the square root of a `long double` value (see `sqrt`(3C))
`csqrt`(3C)	Determines the square root of a `double complex` value (see `sqrt`(3C))

Nearest Integer, Absolute Value, and Remainder

Function	**Description**
`ceil`(3C)	Provides type `double` math functions for ceiling (see `floor`(3C))
`ceill`(3C)	Provides type `long double` math functions for ceiling (see `floor`(3C))
`fabs`(3C)	Computes the absolute value of a `double` floating-point number (see `floor`(3C))
`fabsl`(3C)	Computes the absolute value of a `long double` floating-point number (see `floor`(3C))
`cabs`(3C)	Computes the absolute value of a `double complex` floating-point number (see `floor`(3C))
`floor`(3C)	Provides type `double` math functions for floor
`floorl`(3C)	Provides type `long double` math functions for floor (see `floor`(3C))
`fmod`(3C)	Provides type `double` math functions for remainder (see `floor`(3C))
`fmodl`(3C)	Provides type `long double` math functions for remainder (see `floor`(3C))

Bessel Functions

Function	**Description**
`j0`(3C), `j1`(3C), `jn`(3C), `y0`(3C), `y1`(3C), `yn`(3C)	Return Bessel functions (see `bessel`(3C))

Statistical Functions

Function	**Description**
`erf`(3C)	Returns error function
`erfc`(3C)	Returns complementary error function (see `erf`(3C))

Complex Functions

Function	**Description**
`creal`(3C)	Computes the real part of the `double complex` number *x* (see `cimag`(3C))
`cimag`(3C)	Computes the imaginary part of the `double complex` number *x*
`conj`(3C)	Computes the conjugate of the `double complex` number *x* (see `cimag`(3C))

SEE ALSO

`complex.h`(3C), `prog_diag`(3C), `utilities`(3C) for integer arithmetic functions

Intrinsic Procedures Reference Manual, Cray Research publication SR–2138

Scientific Libraries Reference Manual, Cray Research publication SR–2081

Cray Standard C Reference Manual, Cray Research publication SR–2074, for discussion of complex arithmetic

NAME

`math.h` – Library header for math functions

IMPLEMENTATION

All Cray Research systems

STANDARDS

ANSI

TYPES

None

MACROS

The macros defined in header `math.h` are as follows. Unless noted as ISO/ANSI, all items are XPG4 compatible.

Macro	**Description**
`HUGE_VAL`, `HUGE_VALF`, `HUGE_VALL`	Expands to a positive `double` expression, not necessarily representable as a `float`; a positive `float` expression; and a positive `long double` expression, respectively. On all systems except CRAY T90 systems with IEEE arithmetic, `HUGE_VAL` is the same as `DBL_MAX`, which is the largest `double` value representable. On CRAY T90 systems with IEEE arithmetic, `HUGE_VAL` expands to positive infinity. On machines with IEEE arithmetic, `HUGE_VAL` is also defined, with the same value, in the IEEE floating-point header file, `fp.h`. ISO/ANSI standard.
`HUGE`	Symbolic constant whose value is the largest representable `double` value.
`M_E`	2.7182818284590452354
`M_LOG2E`	1.4426950408889634074
`M_LOG10E`	0.43429448190325182765
`M_LN2`	0.69314718055994530942
`M_LN10`	2.30258509299404568402
`M_PI_2`	Expands to the machine's best representation of *pi*/2 or 1.57079632679489661923.
`M_PI_4`	Expands to the machine's best representation of *pi*/4 or 0.78539816339744830962.
`M_1_PI`	Expands to the machine's best representation of 1/*pi* or 0.31830988618379067154.
`M_2_PI`	Expands to the machine's best representation of 2/*pi* or 0.63661977236758134308.
`M_2_SQRTPI`	Expands to the machine's best representation of 2/*sqrt pi* or 1.12837916709551257390.
`M_SQRT1_2`	Expands to the machine's best representation of *sqrt (1/2)* or 0.70710678118654752440.

When compiling in extended mode, header file `values.h` is included in `math.h`. Thus, all macros defined in `values.h` are available in addition to the previous list. See `values.h`(3C) for more information about these additional macros.

FUNCTION DECLARATIONS

Math functions take `double` or `long double` arguments and return `double` or `long double` values, respectively. Functions declared in header `math.h` are as follows:

`acos`(3C)
`acosf`(3C)
`acosl`(3C)
`asin`(3C)
`asinf`(3C)
`asinl`(3C)
`atan`(3C)
`atanf`(3C)
`atan2`(3C)
`atan2l`(3C)
`atof`(3C)
`ceil`(3C)
`ceilf`(3C)
`ceill`(3C)
`cos`(3C)
`cosf`(3C)
`cosh`(3C)
`coshf`(3C)
`coshl`(3C)
`cosl`(3C)
`erf`(3C)
`erfc`(3C)
`exp`(3C)
`expf`(3C)
`expl`(3C)
`fabs`(3C)
`fabsf`(3C)
`fabsl`(3C)
`floor`(3C)
`floorf`(3C)
`floorl`(3C)
`fmod`(3C)
`fmodf`(3C)
`fmodl`(3C)
`frexp`(3C)
`frexpf`(3C)
`frexpl`(3C)
`gamma`(3C)
`hypot`(3C)
`j0`(3C)
`j1`(3C)
`jn`(3C)
`ldexp`(3C)
`ldexpf`(3C)
`ldexpl`(3C)
`log`(3C)
`logf`(3C)
`log10`(3C)
`log10f`(3C)
`log10l`(3C)
`logl`(3C)
`modf`(3C)
`modff`(3C)
`modfl`(3C)
`pow`(3C)
`powf`(3C)
`powl`(3C)
`sin`(3C)
`sinf`(3C)
`sinh`(3C)
`sinhl`(3C)
`sinl`(3C)
`sqrt`(3C)
`sqrtf`(3C)
`sqrtl`(3C)
`tan`(3C)
`tanf`(3C)
`tanh`(3C)
`tanhf`(3C)
`tanhl`(3C)
`tanl`(3C)
`y0`(3C)
`y1`(3C)
`yn`(3C)

Function `atof`(3C) is declared in header file `math.h` only when compiling in extended mode. See `strtod`(3C) for more information about function `atof`(3C).

CAUTIONS

Some function declarations that were previously in `math.h` and `stdio.h`(3C) are now in `stdlib.h`(3C). Therefore, check to see that your code includes the proper header.

If you have selected strict ANSI conformance mode and, during compilation, the compiler complains about incompatible types for a function and its return value, first check the reference manual to determine if in fact the function is ANSI standard. If it is not and you still want to use it, you will have to explicitly declare it because it will not be declared in the header file you have included.

SEE ALSO

`stdio.h`(3C), `stdlib.h`(3C), `strtod`(3C), `values.h`(3C)

Scientific Libraries Reference Manual, Cray Research publication SR–2081

Intrinsic Procedures Reference Manual, Cray Research publication SR–2138

NAME

`mbtowc`, `mblen`, `wctomb` – Multibyte character handling

SYNOPSIS

```
#include <stdlib.h>
int mbtowc (wchar_t *pwc, const char *s, size_t n);
int mblen (const char *s, size_t n);
int wctomb (char *s, wchar_t wchar);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

If *s* is not a null pointer, the `mbtowc` function determines the number of bytes that comprise the multibyte character to which *s* points. It then determines the code for the value of type `wchar_t` that corresponds to that multibyte character. (The value of the code that corresponds to the null character is 0.) If the multibyte character is valid, and `pwc` is not a null pointer, `mbtowc` stores the code in the object to which *pwc* points. At most *n* bytes of the array to which *s* points are examined.

If *s* is not a null pointer, the `mblen` function determines the number of bytes comprising the multibyte character to which *s* points. Except that the shift state of the `mbtowc` function is not affected, it is equivalent to the following:

```
mbtowc((wchar_t *)0, s, n);
```

The `wctomb` function determines the number of bytes needed to represent the multibyte character that corresponds to the code that has the value *wchar* (including any change in shift state). It stores the multibyte character representation in the array object to which *s* points (if *s* is not a null pointer). At most `MB_CUR_MAX` characters are stored. If the value of *wchar* is 0, `wctomb` is left in the initial shift state.

NOTES

The `LC_CTYPE` category of the current locale affects the behavior of the multibyte character functions. For a state-dependent encoding, each function is placed into its initial state by a call for which its character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null pointer cause the internal state of the function to be altered as necessary. A call with *s* as a null pointer causes these functions to return a nonzero value if encodings have state dependency; otherwise, these functions return 0. Changing the `LC_CTYPE` category causes the shift state of these functions to be indeterminate.

RETURN VALUES

If *s* is a null pointer, `mblen` and `mbtowc` return a nonzero value if multibyte character encodings have state-dependent encodings; they return 0 if multibyte character encodings do not have state-dependent encodings.

If *s* is not a null pointer, `mblen` or `mbtowc` return 0 if *s* points to the null character. They return the number of bytes that comprise the multibyte character if the next *n* or fewer bytes form a valid multibyte character. They return –1 if they do not form a valid multibyte character.

The value returned is never greater than *n* or the value of the `MB_CUR_MAX` macro.

If *s* is a null pointer, `wctomb` returns a nonzero value if multibyte character encodings have state-dependent encodings. It returns 0 if multibyte character encodings do not have state-dependent encodings.

If *s* is not a null pointer, `wctomb` returns –1 if the value of *wchar* does not correspond to a valid multibyte character; otherwise, it returns the number of bytes that comprise the multibyte character that correspond to the value of *wchar*.

In the cases in which the preceding functions return –1, they also may set `errno` to the value `EILSEQ` (a byte/character sequence that is not valid is detected).

SEE ALSO

`locale.h`(3C), `mbstring`(3C)