NAME

mbstowcs, wcstombs – Multibyte string functions

SYNOPSIS

```
#include <stdlib.h>
size_t mbstowcs (wchar_t *pwcs, const char *s, size_t n);
size_t wcstombs (char *s, const wchar_t *pwcs, size_t n);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The mbstowcs function converts a sequence of multibyte characters that begins in the initial shift state from the array to which *s* points into a sequence of corresponding codes and stores not more than *n* codes into the array pointed to by *pwcs*. No multibyte characters that follow a null character (which is converted into a code with value 0) are examined or converted. Each multibyte character is converted as if by a call to mbtowc, except that the shift state of the mbtowc function is not affected. If *pwcs* is a null pointer, the mbstowcs function returns the number of elements required for the wide character code array.

No more than *n* elements are modified in the array to which *pwcs* points. If copying occurs between objects that overlap, the behavior is undefined.

The wcstombs function converts a sequence of codes that correspond to multibyte characters from the array to which *pwcs* points into a sequence of multibyte characters that begins in the initial shift state and stores these multibyte characters into the array to which *s* points, stopping if a multibyte character would exceed the limit of *n* total bytes or if a null character is stored. Each code is converted as if by a call to the wctomb(3C) function, except that the shift state of the wctomb function is not affected. If *s* is a null pointer, the wcstombs function returns the number of bytes required for the character array.

No more than *n* bytes are modified in the array to which *s* points. If copying takes place between objects that overlap, the behavior is undefined.

The LC_CTYPE category of the current locale affects the behavior of the multibyte string functions.

RETURN VALUES

If a multibyte character that is not valid is encountered, mbstowcs returns (size_t) –1, and may set errno to EILSEQ (a character sequence that is not valid is detected). Otherwise, mbstowcs returns the number of array elements modified, not including a terminating 0 code, if any.

On encountering a code that does not correspond to a valid multibyte character, `wcstombs` returns `(size_t)-1`, and it may set `errno` to `EILSEQ`. Otherwise, `wcstombs` returns the number of bytes modified, not including a terminating null character, if any.

SEE ALSO

`locale.h`(3C), `mbchar`(3C)

NAME

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

SYNOPSIS

```
#include <string.h>
void *memchr(const void *s, int c, size_t n);
int memcmp(const void *s1, const void *s2, size_t n);
void *memcpy(void *s1, const void *s2, size_t n);
void *memccpy(void *s1, const void *s2, int c, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
void *memset(void *s, int c, size_t n);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (except memccpy)
XPG4 (memccpy only)

DESCRIPTION

The memchr function locates the first occurrence of *c* (converted to an unsigned char) in the initial *n* characters (each interpreted as unsigned char) of the object pointed to by *s*.

The memcmp function compares the first *n* characters of the object pointed to by *s1* to the first *n* characters of the object pointed to by *s2*.

The memcpy function copies *n* characters from the object pointed to by *s2* into the object pointed to by *s1*. If copying takes place between objects that overlap, the behavior is undefined.

The memccpy function copies characters from memory area *s2* into *s1*, stopping after the first occurrence of character *c* has been copied, or after *n* characters have been copied, whichever comes first.

The memmove function copies *n* characters from the object pointed to by *s2* into the object pointed to by *s1*. Copying takes place as if the *n* characters from the object pointed to by *s2* are first copied into a temporary array of *n* characters that does not overlap the objects pointed to by *s1* and *s2*, and then the *n* characters from the temporary array are copied into the object pointed to by *s1*.

The memset function copies the value of *c* (converted to an unsigned char) into each of the first *n* characters of the object pointed to by *s*.

RETURN VALUES

The `memchr` function returns a pointer to the located character, or a null pointer if the character does not occur in the object.

The `memcmp` function returns an integer that is greater than, equal to, or less than 0, according to whether the object pointed to by *s1* is greater than, equal to, or less than the object pointed to by *s2*.

The `memcpy` and `memmove` functions return the value of *s1*.

Function `memccpy` returns a pointer to the character after the copy of *c* in *s1*, or a null pointer if *c* was not found in the first *n* characters of *s2*.

The `memset` function returns the value of *s*.

SEE ALSO

`bstring`(3C)

NAME

`memory.h` – Library header for string-handling functions

IMPLEMENTATION

All Cray Research systems

STANDARDS

AT&T extension

DESCRIPTION

Header `memory.h` is identical to header `string.h`; it is included only for compatibility with prior use.

SEE ALSO

`string.h`(3C)

NAME

memwcpy, memwset, memstride, memwchr, memwcmp – Performs word-oriented memory operations

SYNOPSIS

```
#include <string.h>
void *memwcpy(long *s1, long *s2, int n);
long *memwset(long *s, long w, int n [, int str]);
long *memstride(long *s1, int str1, long *s2, int str2, int n);
long *memwchr(long *s, long w, int n [, int str]);
int memwcmp(long *s1, long *s2, int n);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The memwcpy function copies *n* words in the memory area addressed by *s2* to the memory area addressed by *s1*. It handles overlapping moves correctly.

The memwset function sets the first *n* words in memory area *s* to the value of word *w*; it returns *s*. If the optional *str* argument is used, it sets every *str*'th word to the value of *w*.

The memstride function copies *n* words from *s2* (with a stride of *str2*) to *s1* (with a stride of *str1*); it returns *s1*.

The memwchr function returns a pointer to the first occurrence of the word *w* in the first *n* words of *s*; it returns 0 if *w* is not found in the first *n* words of *s*. If the optional *str* argument is used, it compares *w* with every *str*'th word of *s*.

The memwcmp function compares the first *n* words of its arguments; it returns 0 if they are identical, or the index of the first detected difference. The index is one-based.

NOTES

For user convenience, all these functions are declared in the optional <string.h> header file. The memwcpy, memstride, and memwcmp functions are declared as fast in-line functions in <string.h>.

NAME

message – Introduction to UNICOS message system functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

These functions provide means for accessing basic system resources affecting the UNICOS message system.

ASSOCIATED HEADERS

<nl_types.h>

ASSOCIATED FUNCTIONS

Function	Description
catclose(3C)	Closes a message catalog (see catopen(3C))
catgetmsg(3C)	Reads a message from a message catalog
catgets(3C)	Gets message from a message catalog
catmsgfmt(3C)	Formats an error message
catopen(3C)	Opens a message catalog

SEE ALSO

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

nl_types(5) for a description of header nl_types.h in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

Cray Message System Programmer's Guide, Cray Research publication SG–2121

NAME

mktemp – Makes a unique file name

SYNOPSIS

```
#include <stdlib.h>
char *mktemp (char *template);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

AT&T extension

DESCRIPTION

The mktemp function replaces the contents of the string to which *template* points with a unique file name, and it returns the address of *template*. The string in *template* should look like a file name with six trailing X's; mktemp replaces the X's with a letter and the current process ID. The letter is chosen so that the resulting name does not duplicate that of an existing file.

NOTES

You may run out of letters. If mktemp cannot create a unique name, it assigns the null string to *template*.

FORTRAN EXTENSIONS

You also can all the mktemp function from Fortran programs, as follows:

```
CHARACTER*n template
INTEGER*8 MKTEMP, I
I = MKTEMP(template)
```

The *template* argument also may be an integer variable. In this case, the data must be packed 8 characters per word and terminated with a null (0) byte.

SEE ALSO

tmpfile(3C), tmpnam(3C)

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

NAME

mktime – Converts local time to calendar time

SYNOPSIS

```
#include <time.h>
time_t mktime (struct tm *timeptr);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The mktime function converts the broken-down time, expressed as local time, in the structure pointed to by *timeptr*, into a calendar time value with the same encoding as that of the values returned by the time(3C) function. The original values of the tm_wday and tm_yday components of the structure are ignored, and the original values of the other components are not restricted to the ranges indicated in the time.h entry. On successful completion, the values of the tm_wday and tm_yday components of the structure are set appropriately, and the other components are set to represent the specified calendar time, but their values are forced to the ranges indicated above; the final value of tm_mday is not set until tm_mon and tm_year are determined.

NOTES

A positive or zero value for tm_isdst causes the mktime function to presume initially that daylight saving time, is (+) or is not (0) in effect for the specified time. A negative value for tm_isdst causes the mktime function to attempt to determine whether daylight saving time is in effect for the specified time.

RETURN VALUES

The mktime function returns the specified calendar time encoded as a value of type time_t. If the calendar time cannot be represented, the function returns the value (time_t)-1.

SEE ALSO

time(3C)

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

NAME

mldlist – Obtains the list of mandatory access control (MAC) labels currently represented in a multilevel directory

SYNOPSIS

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

int mldlist(char *path, mls_t *labels, int count);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The mldlist routine scans the directory structure indicated by the multilevel symbolic link file *path*. If the label list buffer pointer specified by *labels* is a non-null pointer, mldlist merely counts the labels in the multilevel directory and returns that number.

The labels placed in the array to which *labels* points are of the opaque type, mls_t. This data type is created by using mls_create and destroyed by using mls_free. Before discarding the *labels* array, the user should destroy all labels in it by using mls_free.

If *path* is not a multilevel symbolic link, but is a directory or a normal symbolic link to a directory, mldlist returns the label of the directory specified in *path*. This allows mldlist to be used on wildcard directories as well as multilevel directories.

NOTES

The ability to obtain an accurate list of labels depends on MAC access to all subdirectories in the multilevel directory structure. The list of labels returned always represents the total list of labels to which the calling process has MAC read access.

WARNINGS

This routine calls stat(2), lstat(2), and readlink(2), among other system calls. As a result, the routine can sleep or hang if a needed file system resource is unavailable.

RETURN VALUES

If *path* specifies a multilevel symbolic link file, and the target of the symbolic link is an accessible directory, mldlist returns the number of labels represented in the multilevel directory structure.

If *path* is a normal symbolic link to a directory, or *path* is a directory itself, `mldlist` returns a 1 (in this case, the number of labels represented is 1).

If *path* is a not a symbolic link or *path* is not a directory, or access is denied to *path* or its symbolic link target for some reason, `mldlist` returns a –1.

SEE ALSO

`mldname`(3C), `mldwalk`(3C), `mls_create`(3C), `mls_free`(3C), `pathname`(3C)

NAME

mldname – Expands a multilevel symbolic link reference at an arbitrary mandatory access control (MAC) label

SYNOPSIS

```
#include <sys/types.h>
#include <dirent.h>
#include <sys/mac.h>

char *mldname(char *path, mls_t label);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The mldname routine determines the actual path name to which a multilevel symbolic link redirects a path name search at an arbitrary MAC label. The routine takes a path name (*path*) and a MAC label (*label*) as its arguments. If the file to which *path* points is a multilevel symbolic link, mldname computes the labeled subdirectory name based on the MAC label provided in *label* and appends it to the contents of the multilevel symbolic link file. If the file to which *path* points is not a multilevel symbolic link, but is either a directory or a symbolic link to a directory, mldname simply returns the path name *path*.

This routine allocates space for the path name it returns by using malloc and copies the result into that space. It then returns a pointer to the newly composed path name. The allocated space can be freed by using free.

WARNINGS

This routine calls stat(2), lstat(2), and readlink(2), among other system calls. As a result, the routine can sleep or hang if a needed file system resource is unavailable.

RETURN VALUES

This routine returns a pointer to a path name. If an error occurs, it returns a null pointer.

SEE ALSO

free(3C), malloc(3C), mldlist(3C), mldwalk(3C), mls_create(3C), mls_free(3C), opendir(3C), pathname(3C)

NAME

mldwalk – Walks the labeled subdirectories of a multilevel directory (MLD)

SYNOPSIS

```
#include <ftw.h>
#include <sys/stat.h>
#include <sys/secstat.h>
#include <sys/mac.h>

int mldwalk(char *path, int(*fn)(char*, struct stat*, struct secstat*,
int));
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The mldwalk routine traverses the labeled subdirectories of the multilevel directory specified by the multilevel symbolic link file, *path*. For each entry found in a label subdirectory, mldwalk calls the user-supplied function *fn* with a character pointer that points to the full path name of the entry, a stat structure that contains the status of the file corresponding to the entry, a secstat structure that contains the security status of the file corresponding to the entry, and an integer with the following possible values (found in ftw.h):

Value	Description
FTW_F	Nondirectory
FTW_D	Directory
FTW_NS	Object for which stat(2) or secstat(2) system call could not be successfully executed

Unlike the ftw routine, mldwalk does not walk down the directory tree, but walks across the labeled subdirectories and visits each entry in each labeled subdirectory. Therefore, it does not concern itself with the FTW_DNR value (which is a directory that can not be read).

When walking a multilevel directory, mldwalk silently skips all labeled subdirectories it cannot open for reading.

If the file specified in *path* is a directory or a normal symbolic link to a directory rather than a multilevel symbolic link, mldwalk visits each file in the directory and returns.

WARNINGS

This routine calls `stat`(2), `lstat`(2), and `readlink`(2), among other system calls. As a result, the routine can sleep or hang if a needed file system resource is unavailable.

RETURN VALUES

If `mldwalk` exhausts its traversal, it returns a 0. If *fn* returns a nonzero value, `mldwalk` stops its traversal and returns whatever value was returned from *fn*. If an error occurs, `mldwalk` returns –1.

SEE ALSO

`ftw`(3C) `mldlist`(3C), `mldname`(3C), `pathname`(3C)

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

NAME

mls_create – Creates an opaque security label structure

SYNOPSIS

```
#include <sys/types.h>
#include <sys/secparm.h>
#include <sys/mac.h>

mls_t mls_create(int level, long comparts);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The mls_create routine allocates and creates an opaque security label for use with the security label comparison routines (for example, mac_equal). *level* is the desired level and *comparts* is the desired compartment bit mask.

RETURN VALUES

On successful completion, the routine allocates space for and returns the desired security label. Otherwise, no space is allocated, a null pointer is returned, and *errno* is set to indicate the error.

ERRORS

mls_create fails if the following error condition occurs:

Error Code	Description
ENOMEM	The label to be returned required more memory than was allowed for the calling process.

SEE ALSO

mls_extract(3C), mls_free(3C)

NAME

mls_dominate – Performs a security label domination test

SYNOPSIS

```
#include <sys/types.h>
#include <sys/secparm.h>
#include <sys/mac.h>

int mls_create(mls_t mac_pl, mls_t mac_p2);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The mls_dominate routine determines whether *mac_p1* dominates *mac_p2*.

NOTES

Dominance includes equivalence. Therefore, if one label equals another, each shall dominate the other. The two labels may not dominate each other (that is, the labels are disjoint).

RETURN VALUES

A value of –1 is returned and *errno* is set to indicate the error. Otherwise, the mls_dominate function returns a 1 if *mac_pl* dominates *mac_p2*, or a 0 if *mac_pl* does not dominate *mac_p2*.

ERRORS

The mls_dominate routine fails if the following error condition occurs:

Error Code	Description
EINVAL	At least one of the labels, *mac_p1* or *mac*_p2, is not a valid security label.

SEE ALSO

mls_create, mls_equal(3C), mls_free(3C), mls_glb(3C), mls_lub(3C)

NAME

mls_equal – Performs a security label equality test

SYNOPSIS

```
#include <sys/types.h>
#include <sys/secparm.h>
#include <sys/mac.h>

int mls_equal(mls_t mac_p1, mls_t mac_p2);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The mls_equal routine determines whether *mac_p1* equals *mac_p2*.

RETURN VALUES

The mls_equal routine returns a 1 if *mac_p1* is equal to *mac_p2*, or 0 if *mac_p1* does not equal *mac_p2*. If an error occurs, a value of –1 is returned, and *errno* is set to indicate the error.

ERRORS

mls_equal fails if the following error condition occurs:

Error Code	Description
EINVAL	At least one of the labels, *mac_p1* or *mac*_p2, is not a valid security label.

SEE ALSO

mls_create(3C), mls_dominate(3C), mls_free(3C), mls_glb(3C), mls_lub(3C)

NAME

mls_export – Converts internal security label to text representation

SYNOPSIS

```
#include <sys/types.h>
#include <sys/secparm.h>
#include <sys/mac.h>

char *mls_export(mls_t mac_p1);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The mls_export routine converts the internal representation of the security label stored in *mac_p1* into its text representation. The routine allocates space for the text representation, copies the text representation into that space, and returns a character pointer to that space.

The format of the text security label is as follows:

level,*compartment*[,*compartment*[...]]

level is the name that represents the appropriate level; *compartment* is the name that represents the appropriate compartment. If the label does not have any compartments specified, the text none is used.

RETURN VALUES

On successful completion, the mls_export routine allocates storage for and returns the text representation of the label. Otherwise, no storage space is allocated, a null pointer is returned, and *errno* is set to indicate the error.

ERRORS

mls_export fails if the following error condition occurs:

Error Code	Description
ENOMEM	The label to be returned required more memory than was allowed for the calling process.

SEE ALSO

mls_import(3C)

NAME

`mls_extract` – Extracts label from an opaque security label structure

SYNOPSIS

```
#include <sys/types.h>
#include <sys/secparm.h>
#include <sys/mac.h>

void mls_extract(mls_t macp1, int *level, long *comparts);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `mls_extract` routine extracts the level and compartment information from the opaque security label specified by *macp1*; the label information is stored in *level* and *comparts*.

RETURN VALUES

The `mls_extract` routine does not return a value.

SEE ALSO

`mls_create`(3C), `mls_free`(3C)

NAME

`mls_free` – Frees security label storage space

SYNOPSIS

```
#include <sys/types.h>
#include <sys/secparm.h>
#include <sys/mac.h>

void mls_free(mls_t mac_p1);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `mls_free` routine frees memory previously allocated by calls made to any security label function that allocates memory on the caller's behalf (for example `mls_create`). The *mac_p1* argument is the security label to be freed.

RETURN VALUES

The `mls_free` function does not return a value.

SEE ALSO

`mls_create`(3C), `mls_extract`(3C)

NAME

mls_glb – Computes the greatest lower bound

SYNOPSIS

```
#include <sys/types.h>
#include <sys/secparm.h>
#include <sys/mac.h>

mls_t mls_glb(mls_t mac_p1, mls_t mac_p2);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The mls_glb routine allocates space for and returns the security label (if it exists) that is dominated by both the labels specified by *mac_p1* and *mac_p2*, and dominates all other security labels that are dominated by both *mac_p1* and *mac_p2*.

RETURN VALUES

On successful completion, this routine allocates space for and returns the newly allocated bounding security label. Otherwise, no space is allocated, a null pointer is returned, and *errno* is set to indicate the error.

ERRORS

mls_glb fails if the following error conditions occur:

Error Code	Description
ENOMEM	The label to be returned required more memory than was allowed for the calling process.
EINVAL	The bounding security label does not exist, or *mac_p1* and/or *mac_p2* has a wildcard label.

SEE ALSO

mls_create(3C), mls_dominate(3C), mls_free(3C), mls_lub(3C)

NAME

mls_import – Converts text security label to internal representation

SYNOPSIS

```
#include <sys/types.h>
#include <sys/secparm.h>
#include <sys/mac.h>

mls_t mls_import(char *text);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The mls_import routine converts the text representation of the security label specified by *text* into its internal representation. When mls_import is called, the routine allocates storage space for the security label, that may be freed with a call to mls_free(3C).

The format of the text security label is as follows:

level[,*compartment*[,*compartment*[...]]]

Where *level* is the name or numeric value that represents the appropriate level; *compartment* is the name or numeric value that represents the appropriate compartment. If no compartments are specified or the text string none or 0 is used, and the compartment bit mask is set to 0.

RETURN VALUES

On successful completion, the mls_import routine allocates storage for and returns the security label. Otherwise, no storage space is allocated, a null pointer is returned, and *errno* is set to indicate the error.

ERRORS

mls_import fails if the following error conditions occur:

Error Code	Description
ENOMEM	The label to be returned required more memory than was allowed for the calling process.
EINVAL	The string *text* is not a valid text representation of a security label.

SEE ALSO

mls_export(3C), mls_extract(3C), mls_free(3C)

NAME

mls_lub – Computes the least upper bound

SYNOPSIS

```
#include <sys/types.h>
#include <sys/secparm.h>
#include <sys/mac.h>

mls_t mls_lub(mls_t mac_p1, mls_t mac_p2);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The mls_lub routine allocates space for and returns the security label (if it exists) that is dominated by both the security labels specified by *mac_p1* and *mac_p2*, and is dominated by all other security labels that dominate both *mac_p1* and *mac_p2*.

RETURN VALUES

On successful completion, this function allocates space for and returns the newly allocated bounding security label. Otherwise, no space is allocated, a null pointer is returned, and *errno* is set to indicate the error.

Error Code	Description
ENOMEM	The label to be returned required more memory than was allowed for the calling process.
EINVAL	The bounding security label does not exist, or *mac_p1* and/or *mac_p2* has a wildcard label.

SEE ALSO

mls_create(3C), mls_dominate(3C), mls_free(3C), mls_glb(3C),

NAME

MTIMESX, MTIMESCN, MTIMESUP – Returns multitasking overlap time

SYNOPSIS

```
CALL MTIMESX(overlap)
ncpus = MTIMESCN()
irtc = MTIMESUP()
```

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

MTIMESX, MTIMESCN, and MTIMESUP all return fields in the structure that is made known to the system by the mtimes(2) system call. The structure contains execution timing information about multitasking programs.

The following is a list of valid arguments:

Argument	Description
ncpus	Integer that specifies the number of physical CPUs connected at that instant.
irtc	Integer real-time clock value when mtimes structure was last updated by the operating system.
rarray	Real array to hold the timing values returned by MTIMESX.
overlap	Address of the real array to hold the overlap time array of the mtimes structure. The length of this array must be the number of CPUs on the target machine. The elements of the array denote how much CPU time was accumulated by the program with a particular number of CPUs executing. For example, overlap(3) denotes the amount of CPU time accumulated when three CPUs were executing in the program, overlap(8) the amount accumulated when eight CPUs were executing in the program, and so on.

MTIMESX fills the array whose address is passed with the overlap time array of the mtimes structure. Summing the elements of the array filled in by MTIMESX yields total execution time for a multitasking program.

MTIMESCN returns the field of the mtimes structure that denotes how many physical CPUs are connected to the program at that point. The contents of the field may change at any time.

MTIMESUP returns the field of the mtimes structure that denotes the real-time clock value at which the mtimes structure was last updated by the operating system. The contents of the field may change at any time.

MTIMESCN and MTIMESUP are useful for getting accurate timings of small multitasking codes. By using them together, as shown in the following example, you can ensure that your program has as many CPUs as it requires and that none of the CPUs you are using is interrupted by the operating system during the execution of the loop.

NOTES

This routine is available on SPARC systems, so that user codes do not need to be rewritten, but it has no effect.

EXAMPLES

```
 886  CONTINUE
      CPUS = MTIMESCN( )
C       if (not all CPUs here) loop
      IF (CPUS. NE. 8) GO TO 886
      BEFOREUP = MTIMESUP( )

C       ...(work to be timed)...

      AFTERUP = MTIMESUP( )
C       if (interrupted) loop
      IF (AFTERUP .NE. BEFOREUP) GO TO 886
```

SEE ALSO

SECOND(3F), TSECND(3F)

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

NAME

MTTIMES – Prints CPU timing information to stdout

SYNOPSIS

```
CALL  MTTIMES
```

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

The MTTIMES subroutine prints CPU timing information to stdout. MTTIMES can tell you how long multiple CPUs were concurrently active on the job. The average is not a speedup factor, but rather an indication of average overlap. If the amount of time busy-waiting is small relative to the total job time, the average may be close to the actual speedup. This information is the same as that available from the ja(1) command.

The following is a sample of the output from MTTIMES:

```
     CPU Utilization
CPUS |  Time(sec)|   Total
1x   |  0.551=   |  0.551
2x   |  14.203=  |  28.407
3x   |  35.926=  |  107.778
4x   |  3.342=   |  13.368
_____|___________|________
2.78x|  54.022=  |  150.103
```

NOTES

This routine is available on SPARC systems, so that user codes do not need to be rewritten, but it has no effect.

NOTES

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

NAME

multic – Introduction to multitasking functions in C

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

These C functions provide means for accessing basic system resources affecting multitasking.

ASSOCIATED HEADERS

<tfork.h>

ASSOCIATED FUNCTIONS

Function	Description
t_exit	Exits a multitasking process
tfork	Creates a multitasking sibling
t_fork	Creates a multitasking sibling (see tfork)
t_gettid	Returns the TID for specified process (see tid)
t_id	Returns the PID of the caller (see tid)
t_lock	Blocks until the lock is free (see tlock)
t_nlock	Sets a nested lock (see tlock)
t_nunlock	Releases a nested lock (see tlock)
t_testlock	Tests the lock and locks it if necessary (see tlock)
t_tid	Returns the TID of the called function (see tid)
t_unlock	Releases the lock (see tlock)

SEE ALSO

file(3C), message(3C), password(3C), terminal(3C), multif(3F), (all introductory pages to other operating system service functions)

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

CF77 Optimization Guide, Cray Research publication SG–3773

NAME

multif – Introduction to multitasking routines

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

Multitasking routines create and synchronize parallel tasks within programs. The information in this man page describes these routines on Cray PVP systems and SPARC systems.

Macrotasking Environment Variables

The following environment variables are available to tune macrotasked applications on Cray PVP systems. These environment variables control TSKTUNE tuning keywords, which allow you to tune an application for macrotasking without recompiling or relinking your code. For more information about these keywords, see the TSKTUNE man page.

Variable	Description
MP_DBACTIVE	Specifies the number of additional macrotasks that can be readied for execution before an additional logical CPU is acquired; this is called the *activation deadband value*. The value of MP_DBACTIVE can be any positive integer value. The initial value is 0.
MP_DBRELEAS	Specifies the number of logical CPUs retained by the job if there are more CPUs than macrotasks; this is called the *release deadband value*. Any CPUs in excess of this number are released to the system. The initial value is set to 1 less than the number of physical CPUs available on the system. Setting MP_DBRELEAS to less than this value may cause an excessive number of CPUs to be deleted and acquired, and a correspondingly long list of CPUs in the log file. The value of MP_DBRELEAS can be any positive integer value.
MP_MAXCPU	Specifies the maximum number of logical CPUs allowed for macrotasking. The default is the number of physical CPUs on the system. The maximum value is 63.
MP_PRIORITY	Specifies the scheduling priority for macrotasks. Legal values are 0 to 63, 0 being the lowest priority. The default is 31.
MP_STACKSZW	Specifies the initial stack size (in words) for macrotasks.
MP_STACKINW	Specifies the stack increment (in words) for macrotasks.

The MP_STACKSZW variable is supported on SPARC systems. The other environment variables have no effect.

Task Subroutines

The following are the task routines:

Routine	Description
TSKSTART	Initiates a task
TSKTEST	Indicates if a task exists
TSKTUNE	Modifies tuning parameters within the library scheduler

TSKWAIT	Waits for a task to complete execution
TSKVALUE	Retrieves the user identifier specified in the task control array
TSKLIST	Lists the status of each existing task

Lock Routines

Lock routines protect critical regions of code and shared memory. The following are the lock routines:

Routine	Description
LOCKASGN	Identifies an integer variable to be used as a lock
LOCKREL	Releases the identifier assigned to a lock
LOCKON	Sets a lock
LOCKOFF	Clears a lock
LOCKTEST	Returns the state of a lock
NLOCKON	Sets a nested lock and returns control to the calling task
NLOCKOFF	Clears a nested lock and returns control to the calling task
ISELFADD, ICRITADD	Perform *ivar=ivar+ivalue* under protection of ISELFADD hardware semaphore
ISELFMUL, ICRITMUL	Perform *ivar=ivar*ivalue* under protection of hardware semaphore
ISELFSCH	Performs *ivar=ivar*+1 under protection of hardware semaphore
XSELFADD, XCRIADD	Performs *xvar=xvar+xvalue* under protection of hardware semaphore
XSELFMUL, XCRITMUL	Perform *xvar=xvar+xvalue* under protection of hardware semaphore

Event Routines

Event routines signal and synchronize between tasks. The following are the event routines:

Routine	Description
EVASGN	Identifies a variable to be used as an event
EVCLEAR	Clears an event
EVREL	Releases the identifier assigned to a task
EVPOST	Posts an event
EVTEST	Returns the state of an event
EVWAIT	Suspends task execution until an event is posted

Multitasking History Trace Buffer Routines

The user-level routines for the multitasking history trace buffer can be called from a user program to control what is recorded in the buffer and to dump the contents of the buffer to a file. The following are the multitasking history trace buffer routines:

Routine	Description
BUFTUNE	Modifies parameters used to control which multitasking actions are recorded in the history trace buffer
BUFPRINT	Writes a formatted dump of the history trace buffer to a dataset
BUFDUMP	Writes an unformatted dump of the history trace buffer to a file

BUFUSER	Adds entries to the history trace buffer

These routines are present on SPARC systems, but they have no effect.

Barrier Routines

A barrier is a synchronization point in an application, beyond which no task will proceed until a specified number of tasks have reached the barrier. The following are the barrier routines:

Routine	Description
BARASGN	Identifies an integer variable to use as a barrier
BARREL	Releases the identifier assigned to a barrier
BARSYNC	Registers the arrival of a task at the barrier

Timing Routines

Timing routines return a variety of timing information that is helpful when evaluating and tuning multitasked programs. The following are the timing routines:

Routine	Description
MTIMESX, MTIMESUP, MTIMESCN	Return multitasked overlap time
TSECND	Returns user CPU time in seconds
MTTIMES	Prints CPU timing information

The TSECND routine is available for use on SPARC systems. The other routines are available on SPARC platforms so that user codes do not need to be rewritten, but they have no effect.

NAME

ndbm, dbm_open, dbm_close, dbm_fetch, dbm_store, dbm_delete, dbm_firstkey, dbm_nextkey, dbm_error, dbm_clearerr – Database subroutines

SYNOPSIS

```
#include <ndbm.h>

typedef struct {
    void *dptr;
    size_t desize;
} datum;

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

void dbm_close(DBM *db);

datum dbm_fetch(DBM *db, datum key);

int dbm_store(DBM *db, datum key);

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

int dbm_delete(DBM *db, datum key);

datum dbm_firstkey(DBM *db);

datum dbm_nextkey(DBM *db);

int dbm_error(DBM *db);

int dbm_error(DBM *db);

int dbm_clearerr(DBM *db);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

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

Keys and contents are described by the datum typedef:

```
typedef struct {
  void *dptr;
  size_t dsize;
} datum;
```

A datum specifies data of dsize bytes pointed to by `dptr`. Arbitrary binary data, as well as normal ASCII strings, are allowed.

The database is stored in two files. One file is a directory that contains a bit map and has `.dir` as its suffix. The second file contains all data and has `.pag` as its suffix.

Before a database can be accessed, it must be opened by `dbm_open`. This will open and/or create the `file.dir` and `file.pag` files, depending on the flags parameter (see the `open`(2) man page). The mode argument is the same as the third argument in `open`(2).

Once open, the data stored under a key is accessed by `dbm_fetch` and data is placed under a key by `dbm_store`. The flags field can be either `DBM_INSERT` or `DBM_REPLACE`. `DBM_INSERT` will insert only new entries into the database and will not change an existing entry with the same key. `DBM_REPLACE` will replace an existing entry if it has the same key. A key (and its associated contents) is deleted by `dbm_delete`. A linear pass through all keys in a a database may be made in a random order by use of `dbm_firstkey` and `dbm_nextkey`. `dbm_firstkey` will return the first key in the database. `dbm_nextkey` will return the next key in the database. The following code will traverse the database:

```
for (key = dbm_firstkey(db);
     key.dptr != NULL;
     key = dbm_nextkey(db))
```

`dbm_error` returns nonzero when an error has occurred while reading or writing the database. `dbm_clearerr` resets the error condition on the named database.

NOTES

The `.pag` file is designed to contain holes in files. Holes will make this file appear larger than its actual contents.

`dptr` pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 1024 bytes). All key/content pairs that hash together must fit on a single block. `dbm_store` will return an error in the event that a disk block fills with inseparable data.

`dbm_delete` does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by `dbm_firstkey()` and `dbm_nextkey()` rely on the `ndbm` hashing function.

The ndbm functions, except dbm_error, provide interlocks per database to provide a level of thread safe use. Although concurrent updating and reading of a database may lead to unpredictable or unexpected behavior.

RETURN VALUES

All functions that return an int indicate errors with negative values. A zero return indicates that there are no errors. Routines that return a datum indicate errors with a null (0) dptr. If dbm_store called with a flags value of DBM_INSERT and finds an existing entry with the same key, it returns 1.

SEE ALSO

dbm(3C)

NAME

network – Introduction to the network access functions

IMPLEMENTATION

All Cray PVP systems

DESCRIPTION

This subsection describes the functions that make up the network library, the UNICOS network information service facility (NIS), the remote procedure call (RPC) library, and the external data representation (XDR) library.

Associated Headers

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

```
<sys/socket.h>
<sys/types.h>
<rpcsvc/ypclnt.h>
<netdb.h>
<netinet/in.h>
<net/route.h>
<net/if.h>
```

Associated Functions

Function	Description
authkerb_getucred	Converts a Kerberos encryption-based authentication received in a Remote Procedure Call (RPC) request into a traditional UNIX-style authentication (see kerberos_rpc(3C)). Server routine.
authkerb_seccreate	Returns an authentication handle that enables the use of the Kerberos authentication system (see kerberos_rpc(3C)). Client routine.
bindresvport	Binds a socket to a prvileged IP port
dn_comp	Domain name service resolver functions (see resolver)
dn_expand	Domain name service resolver functions (see resolver)
dn_skipname	Domain name service resolver functions (see resolver)
endhostent	Closes /etc/hosts file (see gethost)
endnetent	Closes /etc/networks file (see getnet)
endnetinfo	Closes /etc/networks file (see getnetinfo)
endprotoent	Closes /etc/protocols file (see getprot)
endrpcent	Closes /etc/rpc file (see getrpcent)
endservent	Closes /etc/services file (see getserv)
endtosent	Closes /etc/iptos file (see gettos)
getdomainname	Gets or sets name of current domain (see getdomain)
gethostbyaddr	Searches for host address (see gethost)

gethostbyname	Searches for host name (see gethost)
gethostent	Gets network host entry (see gethost)
gethostinfo	Gets network host and service entry by host name or host address
gethostinfo	Gets network host and service entry by host name or host address
gethostlookup	Gets network host entry (see gethost)
getnetbyaddr	Searches for network entry by address (see getnet)
getnetbyname	Searches for network entry by name (see getnet)
getnetent	Gets network entry (see getnet)
getnetibyaddr	Searches for network entry by address (see getnetinfo)
getnetibyname	Searches for network entry by name (see getnetinfo)
getnetinfo	Gets network entry
getprotobyname	Searches for protocol name (see getprot)
getprotobynumber	Searches for protocol number (see getprot)
getprotoent	Reads entry in /etc/protocol file (see getprot)
getrpcbyname	Gets remote procedure call entry
getrpcbynumber	Gets remote procedure call entry
getrpcent	Gets remote procedure call entry
getservbyname	Searches for service name (see getserv)
getservbyport	Searches for port number (see getserv)
getservent	Gets service entry (see getserv)
gettosbyname	Searches for Type Of Service name (see gettos)
gettosent	Reads next entry in Type Of Service database (see gettos)
herror	Produces host lookup error messages
hostalias	Domain name service resolver functions (see resolver)
htonl	Converts values between host and network byte order (see byteorder)
htonl	Converts values between host and network byte order (see byteorder)
htons	Converts values between host and network byte order (see byteorder)
htons	Converts values between host and network byte order (see byteorder)
inet_addr	Interprets dot notation and returns Internet address (see inet)
inet_addr	Interprets dot notation and returns Internet address (see inet)
inet_lnaof	Separates Internet host addresses and returns local network address (see inet)
inet_lnaof	Separates Internet host addresses and returns local network address (see inet)
inet_makeaddr	Constructs Internet address (see inet)
inet_netof	Separates Internet host addresses and returns network number (see inet)
inet_netof	Separates Internet host addresses and returns network number (see inet)
inet_network	Interprets dot notation and returns Internet number (see inet)
inet_network	Interprets dot notation and returns Internet number (see inet)
inet_ntoa	Interprets Internet address and returns dot notation (see inet)
inet_ntoa	Interprets Internet address and returns dot notation (see inet)
inet_subnetmaskof	Returns subnet of the Internet address (see inet)
inet_subnetof	Returns subnet mask of the Internet address (see inet)

iso_addr	Manipulates ISO/OSI address
iso_ntoa	Manipulates ISO/OSI address (see iso_addr)
kerberos_rpc(3C)	Make remote procedure calls using Kerberos authentication.
nqeapi(3)	General Network Queuing Environment (NQE) functions for formatted lists
nqe_get_policy_list(3)	Queries the Network Load Balancer(NLB) to retrieve a formatted list of hosts that match a specified policy
nqe_get_request_ids(3)	Returns a list of all Network Queuing System (NQS) request IDs known to a specified NLB server
nqe_get_request_info(3)	Returns a list of all information known about a specific NQS request ID from a specified NLB server
ntohl	Converts values between host and network byte order (see byteorder)
ntohl	Converts values between host and network byte order (see byteorder)
ntohs	Converts values between host and network byte order (see byteorder)
ntohs	Converts values between host and network byte order (see byteorder)
parsetos	Gets network Type Of Service information (see gettos)
rcmd	Returns a stream to a remote command
rcmdexec	Returns a stream to a remote command
res_init	Domain name service resolver functions (see resolver)
res_mkquery	Domain name service resolver functions (see resolver)
res_query	Domain name service resolver functions (see resolver)
res_querydomain	Domain name service resolver functions (see resolver)
res_search	Domain name service resolver functions (see resolver)
res_send	Domain name service resolver functions (see resolver)
resolver	Domain name service resolver functions
rexec	Returns a stream to a remote command
rpc	Makes a remote procedure call
rresvport	Returns a descriptor to a socket (see rcmd)
ruserok	Authenticates remote users (see rcmd)
setdomainname	Gets or sets name of current domain (see getdomain)
sethostent	Opens/rewinds /etc/hosts file (see gethost)
setnetent	Opens/rewinds /etc/networks file (see getnet)
setnetinfo	Opens/rewinds /etc/networks file (see getnetinfo)
setprotoent	Opens/rewinds /etc/protocols file (see getprot)
setrpcent	Gets remote procedure call entry
setservent	Opens/rewinds /etc/services file (see getserv)
settosent	Opens/rewinds /etc/iptos file (see gettos)
svc_kerb_reg	Performs registration tasks that are required before Kerberos encryption-based authentication requests are processed (see kerberos_rpc(3C)). Server routine.
xdr	Achieves machine-independent data transformation
yp_all	Network information service (NIS) client interface (see ypclnt)
yp_bind	NIS client interface (see ypclnt)
ypclnt	NIS client interface

yperr_string	NIS client interface (see ypclnt)
yperr_string	NIS client interface (see ypclnt)
yp_first	NIS client interface (see ypclnt)
yp_get_default_domain	NIS client interface (see ypclnt)
yp_get_default_domain	NIS client interface (see ypclnt)
yp_master	NIS client interface (see ypclnt)
yp_match	NIS client interface (see ypclnt)
yp_next	NIS client interface (see ypclnt)
yp_order	NIS client interface (see ypclnt)
ypprot_err	NIS client interface (see ypclnt)
yp_unbind	NIS client interface (see ypclnt)

SEE ALSO

errno.h(3C), intro(3C), perror(3C)

dup(2), intro(2), open(2), pipe(2), read(2), recv(2), send(2), socket(2), write(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

hosts(5), icmp(4P), inet(4P), ip(4P), iptos(5), networks(5), protocols(5), services(5), tcp(4P), udp(4P) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

nextafter, nextafterf, nextafterl – Returns the next value in the direction of the second argument

SYNOPSIS

CRAY T90 systems with IEEE hardware:

```
#include <fp.h>

double nextafter (double x, double y);
float nextafterf (float x, float y);
long double nextafterl (long double x, long double y);
```

Cray MPP systems:

```
#include <fp.h>

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

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 nextafter function and macro and the nextafterf and nextafterl functions determine the next representable value, in the type of the function, after *x* in the direction of *y*. If *x*==*y*, the functions return *y*.

RETURN VALUES

These functions return the next representable value after *x* in the direction of *y*.

It is sometimes desirable to find the next representation after a value in the direction of a previously computed value, either smaller or larger. The nextafter functions have a second floating-point argument so that the program will not have to include floating-point tests for determining the direction in such situations.

For the case *x*==*y*, the IEEE standard recommends that *x* be returned. This specification differs so that nextafter(-0.0,+0.0) returns +0.0, and nextafter(+0.0,-0.0) returns -0.0.

SEE ALSO

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

NAME

`nlimit` – Provides an interface to setting or obtaining resource limit values

SYNOPSIS

```
#include <errno.h>
#include <sys/category.h>
#include <sys/resource.h>

int nlimit (int id, struct resclim *rptr);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The `nlimit` library interface provides a means to establish or view resource limit information from the kernel based on the following arguments:

Argument	Description
id	The *pid*, *sid*, or *uid* corresponding to the `resclim` field `resc_category`. A 0 indicates the current *pid*, *sid*, or *uid*.
rptr	A pointer to the `resclim` structure. The `resclim` structure to which *rptr* points includes the following members (for a complete list, see `/usr/include/sys/resource.h`):

```
struct resclim {
  int   resc_resource;  /* One of:  L_CPU                                  */
  int   resc_category;  /* One of:  C_PROC, C_SESS, C_UID, C_SESSPROCS  */
  int   resc_type;      /* One of:  L_T_HARD, L_T_SOFT    */
  int   resc_action;    /* One of:  L_A_TERMINATE, L_A_CHECKPOINT       */
  int   resc_used;      /* Current amount of resource used              */
  int   resc_value[R_NLIMTYPES];
                        /* Current resource limit value for */
                        /* L_T_HARD AND L_T_SOFT            */
};
```

The `resclim` structure contains fields used to establish or view resource limits. The `nlimit` function sets up the structure according to the parameters passed to it and calls `setlim(2)` to change limit values, or `getlim(2)` to obtain information about limit values.

Obtaining Resource Limits

You must set the `resclim` structure fields `resc_resource`, `resc_category`, and `resc_type` to return limit values. The `resc_resource` field represents the resource to be queried. Currently, only CPU resources are supported; therefore, the value of `resc_resource` must be `L_CPU`. The `resc_category` identifies the category of resource that will be queried. Acceptable values are `C_PROC`, `C_SESS`, `C_UID`, and `C_SESSPROCS`, as follows:

Value	Description
`C_PROC`	Returns process limits
`C_SESS`	Returns session limits
`C_UID`	Returns user limits
`C_SESSPROCS`	Returns default process limits for the session

The `resclim` field `resc_category` determines whether the *id* argument is a *pid*, *sid*, or *uid*. The `resc_category` of `C_SESSPROCS` requires a *sid*.

You must set the `resc_type` field to `NULL` to return its limits.

If the call succeeds, `nlimit` fills in the missing information in the `resclim` structure, including the following fields:

Field	Description
`resc_action`	Returns a value of `L_A_TERMINATE` or `L_A_CHECKPOINT`. When a hard limit is reached, this value determines whether the process is checkpointed before termination.
`resc_used`	Returns the amount of resource currently accumulated at the time of the call. For `L_CPU`, this value is the amount of CPU seconds accumulated.
`resc_value[R_NLIMTYPES]`	Returns two values. The `resc_value[L_T_HARD]` field is the hard resource limit, and `resc_value[L_T_SOFT]` is the soft resource limit.

Setting Resource Limits

To set a limit value, all `resclim` fields must be set to either a value or a null. To set a value to be unlimited, use `CPUUNLIM`. To set a value to be null, use `NULL`.

The following list describes each of the fields in the `resclim` structure and their acceptable values:

Field	Description
`resc_resource`	Represents the resource for which a limit will be established. Currently, only CPU resources are supported; therefore, the value of `resc_resource` must be `L_CPU`.
`resc_category`	Identifies the category of resource that will be set. The `resc_category` determines whether the *id* argument is a *pid*, *sid*, or *uid*. Acceptable values are `C_PROC`, `C_SESS`, `C_UID`, and `C_SESSPROCS`. The `resc_category` of `C_SESSPROCS` requires a *sid*. A short description follows:

`C_PROC`	Sets process limits
`C_SESS`	Sets session limits
`C_UID`	Sets user limits
`C_SESSPROCS`	Sets default process limits for the session

`resc_type` Identifies the type of limit that will be set. Acceptable values are: `L_T_HARD` and `L_T_SOFT`.

`resc_action` When a hard limit is reached, this value determines whether the process is checkpointed before termination. Acceptable values are `NULL`, `L_A_TERMINATE`, or `L_A_CHECKPOINT`. If you set the `resc_action` field to `L_A_TERMINATE` or `L_A_CHECKPOINT`, the `resc_type` must be `L_T_HARD`.

`resc_used` Not used with the `nlimit` call when setting limits; the only acceptable value is `NULL`.

`resc_value[R_NLIMTYPES]`
To set hard limits, set the field `resc_type` to `L_T_HARD` and place a value in `resc_value[L_T_HARD]`. To set soft limits, set the field `resc_type` to `L_T_SOFT` and place a value in `resc_value[L_T_SOFT]`. The values in `resc_value[R_NLIMTYPES]` for `resc_resource L_CPU` must be in seconds. You can set only one of `resc_value[L_T_HARD]` or `resc_value[L_T_SOFT]` with each `nlimit` call.

The `nlimit` function fails and no information is updated in the `resclim` structure or no resource limits are set if one or more of the following error conditions occur:

Error Code	Description
EFAULT	The address specified for *rptr* was invalid.
EINVAL	One of the arguments contains a value that is not valid.
EPERM	The user ID of the requesting process is not that of a super user.
EPERM	An attempt was made to change a limit on a system process; this is not allowed.
ESRCH	No processes were found that matched the request.

NOTES

A functional description of `nlimit` is in `nlimit`(1).

RETURN VALUES

On successful completion, a value of 0 indicates that the call succeeded, and the `resclim` structure was filled in with appropriate returned values. Otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

EXAMPLES

The following example shows the execution of the `nlimit` function:

```
#include <stdio.h>
#include <errno.h>
#include <sys/resource.h>
#include <sys/category.h>

   ...

      int retn;
```

```
   struct resclim r;
   struct resclim *rptr;
   rptr = &r;

...

   /*
    *  Set up fields to return current process limits
    */
   rptr->resc_resource        = L_CPU;
   rptr->resc_category        = C_PROC;
   rptr->resc_type            = NULL;
   retn = nlimit(0, rptr);
   if (retn == -1) {
         fprintf(stderr,"nlimit failed with errno %d\n",errno);
   }

...

   /*
    *  Set current process hard limit to 400 seconds and the hard action
    *  to checkpoint.
    */
   rptr->resc_resource        = L_CPU;
   rptr->resc_category        = C_PROC;
   rptr->resc_type            = L_T_HARD;
   rptr->resc_action          = L_A_CHECKPOINT;
   rptr->resc_value[L_T_HARD] = 400;
   retn = nlimit(0, rptr);
   if (retn == -1) {
         fprintf(stderr,"nlimit failed with errno %d\n",errno);
   }
```

SEE ALSO

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

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

`NLIMIT`(3F) in the

NAME

`nlist` – Gets entries from name list

SYNOPSIS

```
#include <nlist.h>
int nlist (char *filename, struct nlist *nl);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

AT&T extension

DESCRIPTION

The `nlist` function examines the name list in the executable file whose name is pointed to by *filename*, selectively extracts a list of values, and puts them in the array of `nlist` structures to which *nl* points. The name list *nl* consists of an array of structures that contains names of variables, types, and values. The list is terminated with a null name; that is, a null pointer is in the name position of the structure. Each variable name is looked up in the name list of the file. If the name is found, the remaining fields in its `nlist` are filled with the corresponding values from the symbol table. If the name is not found, the fields are set to 0. For a discussion of the symbol table structure, see `relo`(5).

This function is useful for examining the system name list kept in the `/unicos` file. In this way, programs can obtain system addresses that are current.

NOTES

The `nlist` structure in header `nlist.h` does not correspond exactly to the actual symbol table structure (compare the `nlist` structure with the `gse` structure in header `symbol.h`). The `nlist` structure is used primarily for convenience and compatibility.

RETURN VALUES

If the file cannot be read or if it does not contain a valid name list, all type entries are set to `0`.

On error, `nlist` returns –1; otherwise, it returns 0.

SEE ALSO

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

NAME

nl_langinfo – Points to language information

SYNOPSIS

```
#include <langinfo.h>
char *nl_langinfo (nl_item item);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The nl_langinfo function returns a pointer to a string containing information relevant to the particular language or cultural area defined in the program's locale. The manifest constant names and values of the *item* argument are defined in the langinfo.h header file. For example, the following returns a pointer to the string Dom if the identified language is Portuguese, and Sun if the identified language is English:

```
nl_langinfo (ABDAY_1)
```

Following are the currently defined constants. Unless otherwise noted, all are in the LC_TIME category.

Constant	Meaning
CODESET	Codeset name. LC_TIME category.
D_T_FMT	String for formatting date and time
D_FMT	Date format string
T_FMT	Time format string
T_FMT_AMPM	a.m. or p.m. time format string
AM_STR	Ante meridian affix
PM_STR	Post meridian affix
DAY_1	Name of the first day of the week (for example, Sunday)
DAY_2	Name of the second day of the week
DAY_3	Name of the third day of the week
DAY_4	Name of the fourth day of the week
DAY_5	Name of the fifth day of the week
DAY_6	Name of the sixth day of the week
DAY_7	Name of the seventh day of the week
ABDAY_1	Abbreviated name of the first day of the week
ABDAY_2	Abbreviated name of the second day of the week
ABDAY_3	Abbreviated name of the third day of the week
ABDAY_4	Abbreviated name of the fourth day of the week

ABDAY_5	Abbreviated name of the fifth day of the week
ABDAY_6	Abbreviated name of the sixth day of the week
ABDAY_7	Abbreviated name of the seventh day of the week
MON_1	Name of the first month of the year
MON_2	Name of the second month
MON_3	Name of the third month
MON_4	Name of the fourth month
MON_5	Name of the fifth month
MON_6	Name of the sixth month
MON_7	Name of the seventh month
MON_8	Name of the eighth month
MON_9	Name of the ninth month
MON_10	Name of the tenth month
MON_11	Name of the eleventh month
MON_12	Name of the twelfth month
ABMON_1	Abbreviated name of the first month
ABMON_2	Abbreviated name of the second month
ABMON_3	Abbreviated name of the third month
ABMON_4	Abbreviated name of the fourth month
ABMON_5	Abbreviated name of the fifth month
ABMON_6	Abbreviated name of the sixth month
ABMON_7	Abbreviated name of the seventh month
ABMON_8	Abbreviated name of the eighth month
ABMON_9	Abbreviated name of the ninth month
ABMON_10	Abbreviated name of the tenth month
ABMON_11	Abbreviated name of the eleventh month
ABMON_12	Abbreviated name of the twelfth month
ERA	Era description segment
ERA_D_FMT	Era date format string
ERA_D_T_FMT	Era date and time format string
ERA_T_FMT	Era time format string
ALT_DIGITS	Alternative symbols for digits
RADIXCHAR	Radix character. LC_NUMERIC category.
THOUSEP	Separator for thousands. LC_NUMERIC category.
YESEXPR	Affirmative response expression. LC_MESSAGES category.
NOEXPR	Negative response expression. LC_MESSAGES category.
YESSTR	Affirmative response for yes/no queries. LC_MESSAGES category.
NOSTR	Negative response for yes/no queries. LC_MESSAGES category.
CRNCYSTR	Currency symbol, preceded by – if the symbol should appear before the value, + if the symbol should appear after the value, or . if the symbol should replace the radix character. LC_MONETARY category.

RETURN VALUES

In a locale where language information data is not defined, `nl_langinfo` returns a pointer to the corresponding string in the POSIX locale. In all locales, `nl_langinfo` returns a pointer to an empty string if *item* contains a setting that is not valid.

SEE ALSO

`localeconv`(3C), `setlocale`(3C)

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

NAME

NLOCKOFF – Clears a nested lock and returns control to the calling task

SYNOPSIS

CALL NLOCKOFF(*name*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

NLOCKOFF clears a nested lock and returns control to the calling task. When NLOCKOFF is called, the nesting level is decremented. If this is the last active nest level, the task clears the lock. Clearing the lock may allow another task to resume execution, but this is transparent to the task calling NLOCKOFF. NLOCKOFF must always be called to clear a lock that has been set by NLOCKON(3F).

The following is a valid argument for this routine:

Argument	Description
name	Name of a 2-word integer array; the first word is the lock, and the second word is the nesting level.

SEE ALSO

NLOCKON(3F)

NAME

NLOCKON – Sets a nested lock and returns control to the calling task

SYNOPSIS

CALL NLOCKON(*name*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

NLOCKON sets a nested lock and returns control to the calling task. If the lock is already set when NLOCKON is called, the task ID is checked. If this task already holds the lock, the nesting level is incremented, and control returns to the calling task. If this task does not hold the lock, the task is suspended until another task clears the lock. This task then sets the lock when it next resumes execution of user code. This means that placing NLOCKON before a critical region ensures that the code in the region is executed only when the task has unique access to the lock. NLOCKON should be used instead of LOCKON(3F) for codes where critical regions may be nested, usually across subroutine boundaries. NLOCKOFF(3F) must always be called to clear a lock that has been set by the NLOCKON routine.

The following is a valid argument for this routine:

Argument	Description
name	Name of a 2-word integer array; the first word is the lock, and the second word is the nesting level.

CAUTIONS

The LOCKTEST(3F) routine cannot be used on nested locks. The same lock cannot be used for calls to the LOCKON(3F) and NLOCKON(3F) routines. These situations result in job aborts.

EXAMPLES

```
      PROOGRAM MULTI
      INTEGER LOCKWD(2)
      INTEGER REALDATA(1000)
      COMMON /MULTITST/ LOCKWD, REALDATA
C     ...
      CALL LOCKASGN(LOCKWD)
C     ...
      CALL NLOCKON(LOCKWD)
      DO 100 I=1,1000
        IF(REALDATA(I).GE.0)
          CALL FIXIT(I)
        ELSE
          REALDATA(I)=0
        ENDIF
 100  CONTINUE
      CALL NLOCKOFF(LOCKWD)
C     ...
      DO 200 I=1,1000
       CALL FIXIT(I)
 200  CONTINUE
C     ...
      END

      SUBROUTINE FIXIT(X)
      INTEGER X
      CALL NLOCKON(LOCKWD)
      REALDATA(X)=MAX(REALDATA(X),50)
      CALL NLOCKOFF(LOCKWD)
      RETURN
      END
```

SEE ALSO

LOCKON(3F), NLOCKOFF(3F)

NAME

numeric_lim – Introduction to numerical limits headers

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The numerical limits headers provide simple macros that expand to numerical limits and parameters, many of which are machine-specific values. Many of the macros specify maximum and minimum values for data types. Using these macros gives you the correct values. You do not need to know the specific value. Use of these macros also greatly increases the portability of your program.

ASSOCIATED HEADERS

<limits.h>
<float.h>
<values.h>

ASSOCIATED FUNCTIONS

Functions that use the values in these headers are primarily in the Mathematics and the General Utility sections. See math(3C) and utilities(3C) for a list of functions.

SEE ALSO

math(3C), utilities(3C)

NAME

_pack, _unpack – Packs or unpacks 8-bit bytes to/from Cray 64-bit words

SYNOPSIS

```
#include <stdlib.h>
long  _pack (const long *up, char *cp, long bc, long tc);
long _unpack (const char *cp, long *up, long bc, long tc);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

These vectorized functions pack or unpack 8-bit bytes to/from Cray 64-bit words. They can be used, for example, to pack lines from a line buffer to a packed buffer, or unpack lines from a packed buffer to a line buffer. A line buffer contains one byte per word, and a packed buffer contains 8 bytes per word. Arguments are as follows:

up Pointer to unpacked data. When packing, bytes are retrieved from this buffer, one right-justified byte per word. When unpacking, bytes are placed in this buffer, one right-justified byte per word.

cp Pointer to packed data. When packing, bytes are placed in this buffer, 8 bytes per word. When unpacking, bytes are retrieved from this buffer, 8 bytes per word. Packing and unpacking need not start or end on a word boundary.

bc Byte count. When packing, this is the number of bytes to pack from *up* to *cp*, excluding a termination character, if specified. When unpacking, this is the maximum number of bytes to unpack from *cp* to *up*. A termination character, if specified and if encountered, terminates unpacking before the byte count is exhausted.

tc Termination character (an integer). Integer value corresponding to a termination character that will terminate unpacking or which will be appended to the end of the packed bytes. This is an optional parameter. If it is omitted or if its value is –1, unpacking will be terminated only after *bc* bytes are unpacked and packing will not append any characters.

RETURN VALUES

No processing takes place and a –1 is returned if any of the following conditions are true:

- *bc* is less than 0.
- *tc* is invalid (it must be in the range 0 through UCHAR_MAX or 1).

- The routine is called with fewer than three arguments.

If the preceding conditions are not true, processing takes place and the number of bytes packed or unpacked is returned. When unpacking, the termination character, if specified and if encountered, is not unpacked nor is it counted as a unpacked byte. When packing, the termination character, if specified, is packed and is counted as a packed byte.

EXAMPLES

Example 1: This example unpacks bytes from `char_buffer` to `line_buffer`. The `_unpack()` routine unpacks 80 bytes or until a new-line character (`'0`) is encountered, whichever occurs first. The new-line character, if it is encountered, will not be unpacked. The variable `nb` will contain the number of bytes actually unpacked

```
int  nb;
long *line_buffer;
unsigned char *char_buffer;

nb = _unpack(char_buffer, line_buffer, 80, '\n');
```

Example 2: This example packs 80 bytes from `words` to `chars`. No termination character is appended to the end of the bytes.

```
int  nb;
long words[80];
unsigned char chars[80];

nb = _pack(words, chars, 80, -1);
```

NAME

password – Introduction to password and security functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The password and security functions provide means for accessing basic system resources affecting passwords and system security.

ASSOCIATED HEADERS

<grp.h>
<pwd.h>
<rpc/netdb.h>
<stdlib.h>
<sys/sitesec.h>
<sys/types.h> (see sys_types.h)
<udb.h>

ASSOCIATED FUNCTIONS

acid2nam
: Maps IDs to names (see id2nam)

acidnamfree
: Maps IDs to names (see id2nam)

addudb
: Library of user database access functions (see libudb)

deleteudb
: Library of user database access functions (see libudb)

endgrent
: Gets group file entry (see getgrent)

endpwent
: Gets password file entry (see getpwent)

endrpcent
: Gets remote procedure call (RPC) entry (see getrpcent)

endudb
: Library of user database access functions (see libudb)

`fgetgrent`
Gets group file entry (see `getgrent`)

`fgetpwent`
Gets password file entry (see `getpwent`)

`getgrent`
Gets group file entry

`getgrgid`
Gets group file entry (see `getgrent`)

`getgrnam`
Gets group file entry (see `getgrent`)

`getpass`
Reads a password

`getpwent`
Gets password file entry

`getpw` Gets name from UID

`getpwnam`
Gets password file entry (see `getpwent`)

`getpwuid`
Gets password file entry (see `getpwent`)

`getrpcbyname`
Gets remote procedure call (RPC) entry (see `getrpcent`)

`getrpcbynumber`
Gets remote procedure call (RPC) entry (see `getrpcent`)

`getsysudb`
Library of user database access functions (see `libudb`)

`gettrustedudb`
Library of user database access functions (see `libudb`) `getudbchain` Library of user database access functions (see `libudb`) `getudb` Library of user database access functions (see `libudb`)

`getudbnam`
Library of user database access functions (see `libudb`)

`getudbstat`
Library of user database access functions (see `libudb`)

`getudbuid`
Library of user database access functions (see `libudb`)

`gid2nam`
Maps IDs to names (see `id2nam`)

`gidnamfree`
Maps IDs to names (see `id2nam`)

`initgroups`
Initializes group access list

`lockudb`
Library of user database access functions (see `libudb`)

`nam2acid`
Maps IDs to names (see `id2nam`)

`nam2gid`
Maps IDs to names (see `id2nam`) `nam2uid` Maps IDs to names (see `id2nam`)

`putpwent`
Writes password file entry

`rewriteudb`
Library of user database access functions (see `libudb`) `secbits` Returns a bit pattern representing names of security compartments, categories, flags, or permission names (see `secnames`)

`secnames`
Returns a list of security compartments, categories, flags, or permission names

`secnums`
Returns numeric value of given security level or class (see `secnames`)

`secwords`
Returns security level or class given its corresponding numeric value (see `secnames`)

`setdomainname`
Sets name of current domain (see `getdomainname`)

`setgrent`
Gets group file entry (see `getgrent`)

`setpwent`
Gets password file entry (see `getpwent`)

`setrpcent`
Gets remote procedure call (RPC) entry (see `getrpcent`)

`setudb`
Library of user database access functions (see `libudb`)

`setudbpath`
Library of user database access functions (see `libudb`)

`udbisopen`
Library of user database access functions (see `libudb`)

`uid2nam`
: Maps IDs to names (see `id2nam`)

`unlockudb`
: Library of user database access functions (see `libudb`)

`zeroudbstat`
: Library of user database access functions (see `libudb`(3C))

SEE ALSO

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

UNICOS File Formats and Special Files Reference Manual, Cray Research publication SR–2014

NAME

pathname – Computes a true path name from a specified path

SYNOPSIS

```
#include <sys/types.h>
#include <pathlib.h>
#include <errno.h>

char *pathname(char *path, mls_t label, unsigned flags, unsigned *pathinfo)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The pathname routine translates a path name specified in *path* that may contain symbolic link references, multilevel directory (MLD) references, and . or .. references into a true path to the specified file.

The *label* argument allows the caller to specify the security label to be used when resolving MLD references. If a null security label is passed (that is, (mls_t) 0), the security label of the current process is used. If a nonnull security label is passed, the specified security label is used instead of the current process security label. This can be useful when the caller must handle requests at a different security label.

The *flags* argument allows the caller to control the nature of the expansion. The following flags can be combined in any call to pathname:

Flag	Description
PN_ABSOLUTE	If this flag is ON, pathname always resolves the specified path to an absolute path, regardless of whether the specified path is absolute or relative. If this flag is OFF, pathname does not try to produce an absolute path. If the path passed in is absolute or some component of the path is a symbolic link that causes the path to become absolute, pathname produces an absolute path even if PN_ABSOLUTE is OFF. If the result of the translation is relative, pathname produces a relative path if PN_ABSOLUTE is OFF.
PN_FULLMLD	If this flag is ON and the final component of the resulting path is a multilevel symbolic link, pathname expands the last component out to the labeled subdirectory. If this flag is OFF under the same conditions, pathname only resolves the path to the root of the MLD. For example, if /tmp is a multilevel symbolic link to /tmp.mld and pathname is called to resolve /tmp with the PN_FULLMLD flag and a label with a zero compartment set and zero level, pathname returns /tmp.mld/000. Under the same circumstances, if the PN_FULLMLD flag is not used, the path returned is /tmp.mld.

PN_NOFOLLOW If this flag is ON and the final component of the resulting path is a symbolic link or a multilevel symbolic link, pathname does not expand the final component. If this flag is OFF, pathname follows all symbolic links and multilevel symbolic links it encounters when resolving *path*.
For example, if /usr/symlink is a symbolic link to /usr/target and pathname is called to resolve /usr/symlink with the PN_NOFOLLOW flag, the path returned is /usr/symlink. Without the PN_NOFOLLOW flag, the path returned is /usr/target.

PN_KEEPERR If this flag is ON and pathname encounters some kind of error while resolving *path*, pathname returns a buffer containing the path that produced the error. If this flag is OFF, pathname returns a null pointer on error.
It is possible for pathname to return a null pointer even if this flag is ON. This can result from dynamic memory exhaustion or corruption within the calling program. The caller must handle a null return even when the PN_KEEPERR flag is specified.

The *pathinfo* argument provides a pointer to a space into which pathname can place flags that describe conditions encountered while translating *path*. If a null pointer is passed for *pathinfo*, these flags are not returned to the caller. The following flags can be set in the location pointed to by *pathinfo* on return from pathname:

Flag	Description
PI_MLSLINK	The final component of *path* resolves to a multilevel symbolic link.
PI_NOTTHERE	The file specified by the final component of *path* does not actually exist, but the path translation was successful up to the last component.
PI_ERROR	An error occurred during translation of *path*. The value of errno describes the error.

WARNINGS

The pathname routine calls stat(2), lstat(2), and readlink(2), among other system calls. As a result, it may sleep or hang if a needed file system resource is unavailable.

RETURN VALUES

If pathname succeeds in translating the provided path, it returns a pointer to a buffer that is allocated by pathname using malloc(3C) and contains the translated path. This buffer can be released by the caller using free(3C). Each call to pathname allocates a new buffer and does not affect the contents of any previously returned buffer.

If pathname fails to translate the specified path, it normally returns a null pointer. If the PN_KEEPERR flag is specified and pathname fails to translate the specified path, pathname returns a pointer to a buffer containing the name of the file that caused the translation to fail.

If pathname is unable to allocate a buffer, it returns a null pointer even if PN_KEEPERR is set, so a null return must be handled by all callers.

Regardless of the setting of the PN_KEEPERR flag, if pathname fails, it sets the PI_ERROR flag in the location specified by *pathinfo* and sets the global variable errno to indicate the error.

FORTRAN EXTENSIONS

None.

EXAMPLES

The following example shows several different ways that pathname can be called to resolve paths. The program runs through all arguments on the command line, resolving each as a pathname in three different ways and printing each result. Failures fall through to the next example to demonstrate the different ways that failure may be handled by pathname.

```
#include <stdio.h>

#include <sys/types.h>
#include <pathlib.h>
#include <errno.h>

main(argc,argv)
int   argc;
char  *argv[];
{
      char        *result;
      unsigned    info;
      int         errs;
      int         i;

      for (i = 1; i < argc; ++i) {
            /*
             * Obtain the translated pathname as a relative path
             * with full MLD expansion and print it.
             */
            printf("Relative resolution with full MLDs\n");
            fflush(stdout);

            result = pathname(argv[i], (mls_t)0, PN_FULLMLD, &info);
            if (result == (char *)0) {
                  fprintf(stderr, "pathname failed for '%s' ", argv[i]);
                  perror("");
                  errs = 1;
            } else {
                  printf("%s\n", result);
                  free(result);
            }
```

```
/*
 * Obtain the pathname as an absolute path with full
 * MLD expansion and print it.
 */

printf("Absolute resolution with full MLDs\n");
fflush(stdout);

result = pathname(argv[i], (mls_t)0, PN_ABSOLUTE | PN_FULLMLD, &info);
if (result == (char *)0) {
      fprintf(stderr, "pathname failed for '%s' ", argv[i]);
      perror("");
      errs = 1;
} else {
      printf("%s\n", result);
      free(result);
}

/*
 * Obtain the pathname as an absolute path with full
 * MLD expansion, preserving the failed pathname on failure
 * and print the result.
 */
printf("Absolute resolution, full MLDs, keeping error path\n");
result = pathname(argv[i], (mls_t)0,
                  PN_KEEPERR | PN_ABSOLUTE | PN_FULLMLD, &info);
/*
 * First check for a null return, in case pathname(3) had
 * trouble allocating its return buffer.
 */
if (result == (char *)0) {
      fprintf(stderr, "pathname failed for '%s' ", argv[i]);
      perror("");
      errs = 1;
      continue;
}

/*
 * Now check for an error indication in the 'info' flags.
 */
if ((info & PI_ERROR) != 0) {
      fprintf(stderr, "pathname failed for '%s' ", argv[i]);
      perror("");
      errs = 1;
```

```
            } else {
                  printf("%s\n", result);
            }
            free(result);
      }
      exit(errs);
}
```

SEE ALSO

`errno.h`(3C), `free`(3C), `malloc`(3C), `mldname`(3C), `mls_create`(3C), `mls_free`(3C)

`lstat`(2), `readlink`(2), and `stat`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

perror, sys_errlist, sys_nerr – Generates system error messages

SYNOPSIS

```
#include <stdio.h>
void  perror (const char *s);
#include <errno.h>
extern char *sys_errlist[  ];
extern int sys_nerr;
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (perror only)
AT&T extension (sys_errlist and sys_nerr)

DESCRIPTION

The perror function produces on the standard error output a message that describes the last error encountered during a call to a system or library function. The argument string *s* is printed first, then a colon and a blank, followed by the message and a newline character. To be of most use, the argument string should include the name of the program incurring the error. The error number is taken from errno, which is set when errors occur but not cleared when error-free calls are made.

To simplify variant formatting of messages, sys_errlist, an array of message strings, is provided; you can use errno as an index in this table to get the message string without the newline character.

Argument sys_nerr is the largest message number provided for in the table plus 1; it should be checked, because new error codes may be added to the system before they are added to the table.

SEE ALSO

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

Cray Message System Programmer's Guide, Cray Research publication SG–2121

NAME

popen, pclose – Initiates a pipe to or from a process

SYNOPSIS

```
#include <stdio.h>
FILE *popen (const char *command, const char *mode);
int pclose (FILE *stream);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX

DESCRIPTION

The arguments to popen are pointers to null-terminated strings that contain, respectively, a shell command line and an I/O mode, either "r" for reading or "w" for writing. The popen function creates a pipe between the calling program and the command to be executed. The value returned is a stream pointer. If the I/O mode is "w", you can write to the standard input of the command by writing to the file *stream*; if the I/O mode is "r", you can read from the standard output of the command by reading from the file *stream*.

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

Because open files are shared, you can use a *mode* "r" command as an input filter and a *mode* "w" command as an output filter.

If the shell cannot be executed, the status returned by pclose is the same as if the shell terminated using _exit(127).

CAUTIONS

If the original process, and the process opened with popen, concurrently read or write a common file, neither should use buffered I/O, because the buffering gets all mixed up. To forestall problems with an output filter, flush the buffer carefully (that is, with fflush(3C)).

RETURN VALUES

If files or processes cannot be created, or if the shell cannot be accessed. the popen function returns a null pointer. Otherwise, it returns a stream pointer as described previously.

On successful return, `pclose` returns the termination status of the shell that ran the command; otherwise, `pclose` returns `-1`, and sets `errno` to indicate the error.

SEE ALSO

`fclose`(3C), `fflush`(3C), `fopen`(3C), `system`(3C)

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

NAME

pow, powf, powl, cpow – Raises the specified value to a given power

SYNOPSIS

```
#include <math.h>
#include <complex.h> (for function cpow only)
double pow (double x, double y);
float powf (float x, float y);
long double powl (long double x, long double y);
double complex cpow (double complex x, double complex y);
```

IMPLEMENTATION

All Cray Research systems (pow, cpow only)
Cray MPP systems (powf only)
Cray PVP systems (powl only)

STANDARDS

ISO/ANSI (pow only)
CRI extension (all others)

DESCRIPTION

The pow, powf, powl, and cpow functions compute *x* raised to the power *y* for double, float, long double, and double complex numbers, respectively. A domain error occurs if *x* is negative and *y* is not an integral value. A domain error also occurs if the result cannot be represented when *x* is 0 and *y* is less than or equal to 0. A range error may occur.

When code containing calls to these functions is compiled by the Cray Standard C compiler in extended mode, domain checking is not done, errno is not set on error, and the functions do not return to the caller on error. If an error occurs, the program aborts, producing a traceback and a core file. On CRAY T90 systems with IEEE floating-point arithmetic only, in extended mode, errno is not set, but the functions do return to the caller on error. For more information, see the corresponding libm man page (for example, POW(3M)).

Specifying the cc(1) command-line option -h stdc (signifying strict conformance mode) or -h matherr=errno causes these functions to perform domain and range checking, set errno on error, and return to the caller on error.

In strict conformance mode, vectorization is inhibited for loops containing calls to any of these functions. Vectorization is not inhibited in extended mode.

RETURN VALUES

The pow, powf, powl, and cpow functions return the value of *x* raised to the power *y*.

When a program is compiled with -hstdc or -hmatherror=errno on Cray MPP systems and CRAY T90 systems with IEEE arithmetic, under certain error conditions the functions perform as follows:

- pow(*x*,*NaN*) returns NaN, and errno is set to EDOM.
- pow(*NaN*,*y*) returns NaN, and errno is set to EDOM.
- powl(*x*,*NaN*) returns NaN, and errno is set to EDOM.
- powl(*NaN*,*y*) returns NaN, and errno is set to EDOM.
- cpow(*x*,*y*), where either the real or imaginary part of *x* or *y* is NaN, returns NaN+NaN*1.0*i*, and errno is set to EDOM.

SEE ALSO

errno.h(3C)

cc(1) in the *Cray Standard C Reference Manual*, Cray Research publication SR–2074

power(3M) in the *Intrinsic Procedures Reference Manual*, Cray Research publication SR–2138

NAME

printf, fprintf, sprintf, snprintf – Prints formatted output

SYNOPSIS

```
#include <stdio.h>
int printf (const char *format, ... );
int fprintf (FILE *stream, const char *format, ... );
int sprintf (char *s, const char *format, ... );
int snprintf (char * restrict s, size_t n const char * restrict format,
... );
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The `printf` function places output on the standard output stream `stdout` and returns the number of characters transmitted or a negative value if an output error was encountered. The `fprintf` function is equivalent to `printf`(3C) with output written to the stream to which *stream* points instead of `stdout`.

The `sprintf` function is equivalent to `printf`, except that the argument *s* specifies an array into which the generated output is written, rather than to `stdout`. You must ensure enough storage space is available. A null character is written at the end of the characters written; it is not counted as part of the returned sum. If copying occurs between objects that overlap, the behavior is undefined.

The `snprintf` function is equivalent to `fprintf`, except that argument *s* specifies an array into which the generated output is to be written, rather than to a stream. If *n* is zero, nothing is written, and *s* may be a null pointer. Otherwise, output characters beyond the *n-1st* are discarded rather than being written to the array, and a null character is written at the end of the characters actually written into the array. If copying takes place between objects that overlap, the behavior is undefined.

Function `printf` converts, formats, and prints its arguments under the control of *format*. The *format* is a multibyte character string that begins and ends in its initial shift state. It contains two types of objects: ordinary multibyte characters (not %), which are simply copied to the output stream; *conversion specifications*, each of which results in the fetch of 0 or more arguments. The results are undefined if insufficient arguments for the format exist. If the format is exhausted while arguments remain, the excess arguments are evaluated, but otherwise ignored.

Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than to the next unused argument. In this case, the % symbol is replaced by the %n$ symbol, where n is a decimal integer in the range [1, NL_ARGMAX], giving the position of the argument in the argument list. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages.

In format strings containing the %n$ symbol, numbered arguments in the argument list can be referenced from the format string as many times as required.

Each conversion specification is introduced by either the % or the %n$ symbol. After the % or %n$, the following appear in sequence:

1. Zero or more *flags*, which modify the meaning of the conversion specification.
2. An optional decimal digit string that specifies a minimum *field width*. If the converted value has fewer characters than the field width, it is padded on the left (or right, if the left-adjustment flag (-) has been given) to the field width. The field width takes the form of an asterisk * (described later) or a decimal integer.
3. A *precision* that gives the minimum number of digits to appear for the d, i, o, u, x, or X conversions; the number of digits to appear after the decimal point for the e and f conversions; the maximum number of significant digits for the g conversion; or the maximum number of characters to be printed from a string in the s conversion. The precision takes the form of a period (.), followed by either an asterisk * (described later) or an optional decimal integer; if you specify only the period, the precision is taken as 0. If a precision appears with any other conversion specifier, the behavior is undefined.
4. An optional h specifying that a following d, i, o, u, x, or X conversion specifier applies to a short int or unsigned short int argument (the argument will have been promoted according to the integral promotions, and its value is converted to short int or unsigned short int before printing); an optional h specifying that a following n conversion specifier applies to a pointer to a short int argument; an optional l (ell) specifying that a following d, i, o, u, x, or X conversion specifier applies to a long int or unsigned long int argument; an optional ll (ell ell) specifying that the following d, i, o, u, x, or X conversion specifier applies to a long long int or unsigned long long int argument; an optional ll, specifying that a following n conversion specifier applies to a pointer or a long int argument; an optional l specifying that a following n conversion specifier applies to a pointer to a long int argument; or an optional L specifying that a following e, E, f, g, or G conversion specifier applies to a long double argument. If an h, l, or L appears with any other conversion specifier, the behavior is undefined.
5. A character that indicates the type of conversion to be applied.

A field width or precision may be indicated by an asterisk (*) instead of a digit string. In this case, an integer argument supplies the field width or precision. The argument that is actually converted is not fetched until the conversion letter is seen, so the argument specifying field width or precision must appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a - flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

In format strings containing the %n$ symbol, a field width or precision may be indicated by the sequence %n$, where m is a decimal integer in the range [1, NL_ARGMAX] giving the position in the argument list (after the format argument) of an integer argument containing the field width or precision. The following is an example:

```
printf ("%1$d:%2$.*$d:%4$.*3$d0, hour, min, precision, sec);
```

The *format* can contain either numbered argument specifications (that is, %n$, and *m$, or unnumbered argument specifications, that is, % and *), but usually not both. The only exception to this is that %% can be mixed with the %n$ form. The results of mixing numbered and unnumbered argument specifications in a *format* string are undefined. When numbered argument specifications are used, specifyng the *n*th argument requires that all the leading arguments, from the first to the (*n–1*th), are specified in the format string.

The flag characters and their meanings are as follows:

Flag	Description
’	The integer portion of the result of a decimal conversion (%i, %d, %u, %f, %g, or %G) are formatted with thousands’ grouping characters. For other conversions, the behavior is undefined. The nonmonetary grouping character is used.
-	Conversion result is left-justified within the field.
+	A signed conversion result always begins with a + or - sign.
space	If the first character of a signed conversion is not a sign, a space is prefixed to the result. This implies that if the space and + flags both appear, the space flag is ignored.
#	Specifies that the value will be converted to an alternative form. For o conversion, it increases the precision to force the first digit of the result to be a 0. For x (X) conversion, a nonzero result has 0x (0X) prefixed to it. For e, E, f, g, and G conversions, the result always contains a decimal point, even if no digits follow the point (usually, a decimal point appears in the result of these conversions only if a digit follows it). For g and G conversions, trailing 0’s are not removed from the result. For other conversions, the behavior is undefined.

For d, i, o, u, x, X, e, E, f, g, and G conversions, leading 0’s (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the 0 and - flags both appear, the 0 flag is ignored. For d, i, o, u, x, and X conversions, if a precision is specified, the 0 flag is ignored. For other conversions, the behavior is undefined.

The conversion characters and their meanings are as follows:

Character	Description
d,i	The integer argument is converted to signed decimal in the style [-]*dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading 0’s. The default precision is 1. The result of converting a 0 value with a precision of 0 is no characters.

o, u, x, X, B The `unsigned int` argument is converted to unsigned octal (`o`), unsigned decimal (`u`), unsigned hexadecimal notation (`x` or `X`), or unsigned binary notation (`B`) in the style *dddd*; the letters `abcdef` are used for `x` conversion, and the letters `ABCDEF` are used for `X` conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading 0's. The default precision is 1. The result of converting a 0 value with a precision of 0 is no characters.

f The `double` argument is converted to decimal notation in the style [-]*ddd.ddd*, in which the number of digits after the decimal point is equal to the precision specification. If the precision is missing, 6 digits are output; if the precision is explicitly 0 and the # flag is not specified, no decimal point appears. If a decimal point character appears, at least 1 digit appears before it. The value is rounded to be the appropriate number of digits.

e, E The `double` argument is converted in the style [-]*d.ddd*e±*dd*, in which a 1 digit is before the decimal point, and the number of digits after it is equal to the precision; when the precision is missing, it is assumed to be 6; if the precision is 0 and the # flag is not specified, no decimal point appears. The value is rounded to the appropriate number of digits. The `E` format code produces a number with `E` instead of `e` introducing the exponent. The exponent always contains at least 2 digits. If the value is 0, the exponent is 0.

g, G The `double` argument is printed in style `f` or `e` (or in style `E` in the case of a `G` format code), with the precision specifying the number of significant digits. If the precision is 0, it is taken as 1. The style used depends on the value converted: style `e` is used only if the exponent resulting from the conversion is less than −4 or greater than or equal to the precision. Trailing 0's are removed from the fractional portion of the result; a decimal point appears only if it is followed by a digit.

c The `int` argument is converted to an `unsigned char`, and the resulting character is written.

s The argument is taken to be a string (character pointer), and characters from the string are printed until a null character (`\0`) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed.

p The argument is a pointer to `void`. The value of the pointer is converted to a sequence of printable characters, in an implementation-defined manner. On Cray Research systems, the conversion is the same as the `o` conversion.

n The argument is a pointer to an integer into which is written the number of characters written to the output stream so far by this call to `fprintf`. No argument is converted.

C The `wchar_t` argument is converted to an array of bytes representing a character, and the resulting character is written. If the precision is specified, its effect is undefined. The conversion is the same as the expected the `wctomb()` function.

S The argument must be a pointer to an array of type `wchar_t`. Wide character codes from the array up to, but not including any terminating null wide-character code, are converted to a sequence of bytes, and the resulting bytes written. If the precision is specified no more than that, many bytes are written and only complete characters are written. If the precision is not specified, or is greater than the size of the array of converted bytes, the array of wide characters must be terminated by a null wide character. The conversion is the same as that expected from the `wcstombs()` function.

% This flag prints a %; no argument is converted.

If any argument is, or points to, a union or an aggregate (except for an array of character type using `%s` conversion, or a pointer using `%p` conversion), the behavior is undefined.

If a conversion specification is not valid, the behavior is undefined. In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by `printf` and `fprintf` are printed as if `putc`(3C) had been called.

For machines with IEEE arithmetic, the `e`, `E`, `f`, `g`, and `G` formats print infinity as `Inf` and "not-a-number" as `NaN`.

RETURN VALUES

The `printf` and `fprintf` functions return the number of characters transmitted, or a negative value if an output error occurred.

The `sprintf` function returns the number of characters written in the array, not counting the terminating null character.

EXAMPLES

To print a date and time in the form ‘‘Sunday, July 3, 10:02’’, where *weekday* and *month* are pointers to null-terminated strings, enter the following command line:

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

To print *pi* to 5 decimal places, enter the following command line:

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

SEE ALSO

`ecvt`(3C), `putc`(3C), `scanf`(3C), `stdio.h`(3C), `vprintf`(3C)

NAME

priv_clear_file – Clears all privilege sets in a file privilege state

SYNOPSIS

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

int priv_clear_file(priv_file_t *privstate);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_clear_file routine clears all privilege sets in the file privilege state to which *privstate* points.

RETURN VALUES

A return value of 0 indicates success. A return value of –1 indicates that an error has occurred, and an error code is stored in *errno*. If the return value is –1, the contents of the privilege state to which *privstate* points is not affected.

ERRORS

priv_clear_file fails if the following error condition occurs:

Error Code	Description
EINVAL	An illegal or undefined value was supplied for *privstate*.

NAME

priv_clear_proc – Clears all privilege sets in a process privilege state

SYNOPSIS

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

int priv_clear_proc(priv_proc_t *privstate);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_clear_proc routine clears all privilege sets in the process privilege state to which *privstate* points.

RETURN VALUES

A return value of 0 indicates success. A return value of –1 indicates that an error has occurred, and an error code is stored in *errno*. If the return value is –1, the contents of the privilege state to which *privstate* points is not affected.

ERRORS

priv_clear_proc fails if the following error condition occurs:

Error Code	Description
EINVAL	An illegal undefined value was supplied for *privstate*.

SEE ALSO

priv_init_proc(3C)

NAME

priv_dup_file – Creates a copy of a file privilege state

SYNOPSIS

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

priv_file_t *priv_dup_file(priv_file_t *source);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_dup_file routine creates a copy of the file privilege state to which *source* points. This routine allocates any memory necessary to hold the new file privilege state and returns a pointer to that privilege state. Once duplicated, an operation on either privilege state does not affect the other.

RETURN VALUES

If successful, returns a pointer to the new file privilege state. A return value of null indicates that an error has occurred, and an error code is stored in *errno*.

ERRORS

priv_dup_file fails if any of the following error conditions occur:

Error Code	Description
EINVAL	An illegal or undefined value was supplied for *source*.
ENOMEM	Insufficient memory was available to allocate the new file privilege state.

NAME

priv_dup_proc – Creates a copy of a process privilege state

SYNOPSIS

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

priv_proc_t *priv_dup_proc(priv_proc_t *source);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_dup_proc routine creates a copy of the process privilege state to which *source* points. This routine allocates any memory necessary to hold the new process privilege state and returns a pointer to that privilege state. Once duplicated, an operation on either privilege state does not affect the other.

RETURN VALUES

If successful, priv_dup_proc returns a pointer to the new process privilege state. A return value of null indicates that an error has occurred, and an error code is stored in *errno*.

ERRORS

priv_dup_proc fails if any of the following error conditions occur:

Error Code	Description
EINVAL	An illegal or undefined value was supplied for *source*.
ENOMEM	Insufficient memory was available to allocate the new process.

NAME

priv_free_file – Deallocates file privilege state space

SYNOPSIS

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

priv_free_file(priv_file_t *privstate);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_free_file routine deallocates space associated with the file privilege state to which *privstate* points.

RETURN VALUES

A return value of 0 indicates success. A return value of –1 indicates that an error has occurred, and an error code is stored in *errno*.

ERRORS

priv_free_file fails if the following error condition occurs:

Error Code	Description
EINVAL	An illegal or undefined value was supplied for *privstate*.

SEE ALSO

priv_init_file(3C)

NAME

priv_free_proc – Deallocates process privilege state space

SYNOPSIS

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

int priv_free_proc(priv_proc_t *privstate);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_free_proc routine deallocates the space associated with the process privilege state to which *privstate* points.

RETURN VALUES

A return value of 0 indicates success. A return value of –1 indicates that an error has occurred, and an error code is stored in *errno*.

ERRORS

priv_free_proc fails if the following error condition occurs:

Error Code	Description
EINVAL	An illegal or undefined value was supplied for *privstate*.

SEE ALSO

priv_init_proc(3C)

NAME

priv_get_fd – Gets the privilege state of a file

SYNOPSIS

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

priv_file_t *priv_get_fd(int fdes);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_get_fd routine uses the fgetpal(2) system call to get the privilege state of the file identified by the file descriptor *fdes*. This function allocates any memory necessary to hold the file privilege state and returns a pointer to that privilege state.

The caller must have MAC read access to the file or have PRIV_MAC_READ in its effective privilege set.

RETURN VALUES

If successful, priv_get_fd returns a pointer to the file privilege state. A return value of null indicates that an error has occurred, and an error code is stored in *errno*.

ERRORS

priv_get_fd fails if any of the following error conditions occur:

Error Code	Description
EBADF	An illegal or undefined value was specified for *fdes*.
EACCES	The caller does not have MAC read access to the file.
ENOMEM	Insufficient memory was available to allocate the file privilege state.

SEE ALSO

priv_get_file(3C), priv_set_fd(3C), priv_set_file(3C)

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

NAME

priv_get_file – Gets the privilege state of a file

SYNOPSIS

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

priv_file_t *priv_get_file(char *path);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_get_file routine uses the getpal(2) system call to get the privilege state of the file identified by *path*. This function allocates any memory necessary to hold the file privilege state and returns a pointer to that privilege state.

The caller must have MAC read access to the file or have PRIV_MAC_READ in its effective privilege set.

RETURN VALUES

If successful, priv_get_file returns a pointer to the file privilege state. A return value of null indicates that an error has occurred, and an error code is stored in *errno*.

ERROR

priv_get_file fails if any of the following error conditions occur:

Error Code	Description
EFAULT	The *path* argument points outside the process address space.
EACCES	Search permission is denied for a component of the *path* prefix or the caller does not have MAC read access to the file.
ENOENT	A component of the specified *path* does not exist.
ENOTDIR	A component of the *path* prefix is not a directory.
ENAMETOOLONG	The length of the *path* argument exceeds PATH_MAX, or a path name component is longer than NAME_MAX while POSIX_NO_TRUNC is in effect.
ENOMEM	Insufficient memory was available to allocate the file privilege state.

SEE ALSO

priv_get_fd(3C), priv_set_fd(3C), priv_set_file(3C)

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

NAME

priv_get_file_flag – Indicates the existence of a privilege in a file privilege set

SYNOPSIS

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

int priv_get_file_flag(priv_file_t *privstate, priv_value_t priv,
priv_fflag_t flag, priv_flag_value_t *value_p);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_get_file_flag routine indicates whether the privilege identified by *priv* exists in the file privilege set, identified by *flag*, of the file privilege state to which *privstate* points. A value that indicates whether the privilege exists is placed in the location to which *value_p* points.

If the value placed in the location to which *value_p* points is PRIV_SET, the specified privilege exists in the privilege set. If the privilege does not exist, the value placed in the location to which *value_p* points is PRIV_CLEAR.

The *priv* argument is the privilege identifier (for example, PRIV_MAC_READ). The *flag* argument is a privilege set identifier. PRIV_ALLOWED, PRIV_FORCED, and PRIV_SETEFF identify the file's allowed, forced, and set-effective privilege sets, respectively.

RETURN VALUES

A return value of 0 indicates success. A return value of –1 indicates that an error has occurred, and an error code is stored in *errno*. If the return value is –1, the contents of the location to which *value* points is not affected.

ERRORS

priv_get_file_flag fails if the following error condition occurs:

Error Code	Description
EINVAL	An illegal or undefined value was supplied for *privstate*, *priv*, *value_p*, or *flag*.

SEE ALSO

priv_set_file_flag(3C)

NAME

priv_get_proc – Gets the privilege state of the calling process

SYNOPSIS

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

priv_proc_t *priv_get_proc();
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_get_proc routine uses the getppriv(2) system call to get the privilege state of the calling process. This routine allocates any memory necessary to hold the process privilege state and returns a pointer to that privilege state.

RETURN VALUES

If successful, priv_get_proc returns a pointer to the process privilege state. A return value of null indicates that an error has occurred, and an error code is stored in *errno*.

ERRORS

priv_get_proc fails if the following error condition occurs:

Error Code	Description
ENOMEM	Insufficient memory was available to allocate the privilege state.

SEE ALSO

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

NAME

priv_get_proc_flag – Indicates the existence of a privilege in a process privilege state

SYNOPSIS

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

int priv_get_proc_flag(priv_proc_t *privstate, priv_value_t priv,
priv_pflag_t flag, priv_flag_value_t *value_p);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_proc_flag routine indicates whether the privilege identified by *priv* exists in the process privilege set, identified by *flag*, of the process privilege state to which *privstate* points. A value that indicates whether the privilege exists is placed in the location to which *value_p* points.

If the value placed in the location to which *value_p* points is PRIV_SET, then the specified privilege exists in the privilege set. If the privilege does not exist, the value placed in the location to which *value_p* points is PRIV_CLEAR.

The *priv* argument is the privilege identifier (for example, PRIV_MAC_READ). The *flag* argument is a privilege set identifier. PRIV_PERMITTED and PRIV_EFFECTIVE identify the process permitted and effective privilege sets, respectively.

RETURN VALUES

A return value of 0 indicates success. A return value of –1 indicates that an error has occurred, and an error code is stored in *errno*. If the return value is –1, the contents of the location to which *value_p* points is not modified.

ERRORS

priv_get_proc_flag fails if the following error condition occurs:

Error Code	Description
EINVAL	An illegal or undefined value was supplied for *privstate*, *priv*, *value_p*, or *flag*.

SEE ALSO

priv_set_proc_flag(3C)

NAME

priv_init_file – Allocates space to hold a file privilege state

SYNOPSIS

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

priv_file_t *priv_init_file();
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_init_file routine allocates space to hold a file privilege state and returns a pointer to that privilege state. The allocated space is cleared.

RETURN VALUES

If successful, priv_init_file returns a pointer to the allocated space. A return value of null indicates that an error has occurred, and an error code is stored in *errno*.

ERRORS

priv_init_file fails if the following error condition occurs:

Error Code	Description
ENOMEM	Insufficient memory was available to allocate the file privilege state.

SEE ALSO

priv_free_file(3C)

NAME

priv_init_proc – Allocates space to hold a process privilege state

SYNOPSIS

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

priv_proc_t *priv_init_proc();
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_init_proc routine allocates space to hold a process privilege state and returns a pointer to that privilege state. The allocated space is cleared.

RETURN VALUES

If successful, priv_init_proc returns a pointer to the allocated space. A return value of null indicates that an error occurred and an error code is stored in *errno*.

ERRORS

priv_init_proc fails if the following error condition occurs:

Error Code	Description
ENOMEM	Insufficient memory was available to allocate the process privilege state.

SEE ALSO

priv_free_proc(3C)

NAME

priv_set_fd – Sets the privilege state of a file

SYNOPSIS

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

int priv_set_fd(int fdes, priv_file_t *privstate);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_set_fd routine uses the fsetpal(2) system call to set the privilege state of the file identified by the file descriptor *fdes* to the file privilege state to which *privstate* points.

The calling process must have PRIV_SETFPRIV in its effective privilege set, each privilege whose state is being altered must exist in the permitted privilege set of the calling process, and the caller must either own the file or have the privilege PRIV_FOWNER in its effective privilege set.

This routine retrieves the file's current privilege assignment list (PAL) records, combines the records with the supplied privilege state, and passes the result to fsetpal(2). This routine does not modify the PAL records of a file. The caller must have MAC read and write access to the file, or have PRIV_MAC_READ and PRIV_MAC_WRITE in its effective privilege set.

RETURN VALUES

A return value of 0 indicates success. A return value of –1 indicates that an error has occurred and an error code is stored in *errno*. If the return value is –1, the privilege state of the file is not affected.

ERRORS

priv_set_fd fails if any of the following error conditions occur:

Error Code	Description
EINVAL	An illegal or undefined value was supplied for *privstate*.
EPERM	The calling process does not have PRIV_SETFPRIV in its effective privilege set, is not the file's owner, or is attempting to change the state of a privilege that it does not have in its permitted privilege set.
EROFS	The named file resides on a read-only file system.
EBADF	An illegal or undefined value was specified for *fdes*.
EACCES	The caller does not have MAC read or MAC write access to the file.

SEE ALSO

`priv_get_fd`(3C), `priv_get_file`(3C), `priv_set_file`(3C)

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

NAME

priv_set_file – Sets the privilege state of a file

SYNOPSIS

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

int priv_set_file(char *path, priv_file_t *privstate);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_set_file routine uses the setpal system call to set the privilege state of the file identified by *path* to the file privilege state to which *privstate* points.

The calling process must have PRIV_SETFPRIV in its effective privilege set, each privilege whose state is being altered must exist in the calling process' permitted privilege set, and the caller must either own the file or have the PRIV_FOWNER privilege in its effective privilege set.

This routine retrieves the file's current privilege assignment list (PAL) category records, combines the records with the supplied privilege state, and passes the result to setpal(2). This routine does not modify the PAL category records of a file. The caller must have MAC read and write access to the file, or have PRIV_MAC_READ and PRIV_MAC_WRITE in its effective privilege set.

RETURN VALUES

A return value of 0 indicates success. A return value of –1 indicates that an error has occurred, and an error code is stored in *errno*. If the return value is –1, the privilege state of the file is not affected.

ERRORS

priv_set_file fails if any of the following error conditions occur:

Error Code	Description
EFAULT	The *path* argument points outside the process address space.
EINVAL	An illegal or undefined value was supplied for *privstate*.
EACCES	Search permission is denied for a component of the *path* prefix.
ENOENT	A component of the specified *path* does not exist.
ENOTDIR	A component of the *path* prefix is not a directory.
ENAMETOOLONG	The length of the *path* argument exceeds PATH_MAX, or a path name component is longer than NAME_MAX while POSIX_NO_TRUNC is in effect.

EPERM	The calling process does not have PRIV_SETFPRIV in its effective privilege set, is not the file's owner, or is attempting to change the state of a privilege that it does not have in its permitted privilege set.
EROFS	The specified file resides on a read-only file system.
EACCES	The caller does not have both MAC read and MAC write access to the file.

SEE ALSO

priv_get_fd(3C), priv_get_file(3C), priv_set_fd(3C)

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

NAME

priv_set_file_flag – Adds or removes privileges of a file privilege set

SYNOPSIS

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

int priv_set_file_flag(priv_file_t *privstate, priv_fflag_t flag, int npriv,
priv_value_t *privs, priv_flag_value_t value);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_set_file_flag routine adds (or removes) the privileges specified in the *privs* array to (or from) the file privilege set, identified by *flag*, of the file privilege state to which *privstate* points.

To add the specified privileges, *value* must be PRIV_SET. To remove the specified privileges, *value* must be PRIV_CLEAR.

Each element in the *privs* array is a privilege identifier (for example, PRIV_MAC_READ). The *flag* argument is a privilege set identifier. The PRIV_ALLOWED, PRIV_FORCED, and PRIV_SETEFF flags identify the file's allowed, forced, and set-effective privilege sets, respectively.

RETURN VALUES

A return value of 0 indicates success. A return value of –1 indicates that an error has occurred, and an error code is stored in *errno*. If the return value is –1, the contents of the specified privilege set is not affected.

ERRORS

priv_set_file_flag fails if the following error condition occurs:

Error Code	Description
EINVAL	An illegal or undefined value was supplied for *privstate*, *flag*, *npriv*, *privs*, or *value*.

SEE ALSO

priv_get_file_flag(3C)

NAME

priv_set_proc – Sets the privilege state of the calling process

SYNOPSIS

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

int priv_set_proc(priv_proc_t *privstate);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_set_proc routine uses the setppriv(2) system call to set the privilege state of the calling process to the process privilege state to which *privstate* points. The function returns an error if an attempt is made to modify the state of any privilege that is not in the calling process' permitted privilege set.

RETURN VALUES

A return value of 0 indicates success. A return value of –1 indicates that an error has occurred, and an error code is stored in *errno*. If the return value is –1, the privilege state of the calling process is not affected.

ERRORS

priv_set_proc fails if any of the following error conditions occur:

Error Code	Description
EINVAL	An illegal or undefined value was supplied for *privstate*.
EPERM	The calling process attempted to modify the state of a privilege that was not in its permitted privilege set.

SEE ALSO

priv_get_proc(3C)

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

NAME

priv_set_proc_flag – Adds or removes privileges of a process privilege state

SYNOPSIS

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

int  priv_set_proc_flag(priv_proc_t *privstate, priv_pflag_t flag, int npriv,
priv_value_t *privs, priv_flag_value_t value);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_set_proc_flag routine adds (or removes) the privileges specified in array *privs* to (or from) the process privilege set identified by *flag* of the process privilege state to which *privstate* points. The number of privileges specified in the array is *npriv*.

To add the specified privileges, *value* must be PRIV_SET. To remove the specified privileges, *value* must be PRIV_CLEAR.

Each element in the *privs* array is a privilege identifier (for example, PRIV_MAC_READ). The *flag* argument is a privilege set identifier. PRIV_PERMITTED and PRIV_EFFECTIVE identify the process permitted and effective privilege sets, respectively.

RETURN VALUES

A return value of 0 indicates success. A return value of –1 indicates that an error has occurred, and an error code is stored in *errno*. If the return value is –1, the contents of the specified privilege set is not affected.

ERRORS

priv_set_proc_flag fails if the following error condition occurs:

Error Code	Description
EINVAL	An illegal or undefined value was supplied for *privstate*, *flag*, *npriv*, *privs*, or *value*.

SEE ALSO

priv_get_proc_flag(3C)

NAME

prog_diag – Introduction to program diagnostics and error handling functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The program diagnostics and error handling functions provide various means for aiding the programmer in diagnosing programming errors; detecting, reporting, and diagnosing run-time errors; and measuring program performance. These functions differ from other debugging and performance measuring facilities in that these are executed within the program at run time.

errno

The value of errno is 0 at program startup, but is never set to 0 by any library function. A library function may set errno to a positive value to indicate the type of error. All those library functions that set errno on error are so documented in that manual entry. If the calling function wishes to check for errors, it is the caller's responsibility to set errno to 0 before the call and then check after the call.

errno is a macro that expands to an expression that can be used anywhere a simple variable can be used. In strict ANSI terminology, it is a modifiable lvalue. If a program defines an identifier with the name errno, the behavior is undefined.

By default, Standard C library math functions do not support errno to provide significantly better performance. You must specify the -hstdc option or the -h matherr=errno option on the cc command line to force the math functions to support errno.

Function sys_errlist provides an array of message strings; function sys_nerr provides the largest message number in system table (see perror).

Error Messages

If errno has been set, the associated error message can be formatted with either the perror or the strerror function.

Program Termination

The exit(2) system call allows a process to return its error status to its parent process.

ASSOCIATED HEADERS

<assert.h>
<errno.h>

ASSOCIATED FUNCTIONS

Function	Description
assert	Verifies program assertion
FLOWMARK	Provides a more detailed flowtrace
STKSTAT	Collects stack statistics
STACKSZ	Reports stack statistics (see STKSTAT)
tracebk	Prints a traceback

SEE ALSO

clock(3C), perror(3C), rtclock(3C)

performance(7) (available only online)

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

Scientific Libraries Reference Manual, Cray Research publication SR–2081

Intrinsic Procedures Reference Manual, Cray Research publication SR–2138

NAME

pthread_create, pthread_detach, pthread_join, pthread_exit, pthread_self, pthread_equal, pthread_once, pthread_attr_init, pthread_attr_destroy, pthread_attr_setdetachstate, pthread_attr_getdetachstate, pthread_attr_setstacksize, pthread_attr_getstacksize, pthread_attr_setstackaddr, pthread_attr_getstackaddr – Thread management

SYNOPSIS

```
#include <pthread.h>

int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void
*(*start_routine)(void *), void *arg);

int pthread_detach (pthread_t thread);

int pthread_join (pthread_t thread, void **value_ptr);

void pthread_exit (void *value_ptr);

pthread_t pthread_self (void);

int pthread_equal (pthread_t t1, pthread_t t2);

pthread_once_t once_control = PTHREAD_ONCE_INIT;

int pthread_once (pthread_once_t *once_control, void (*init_routine)(void));

int pthread_attr_init (pthread_attr_t *attr);

int pthread_attr_destroy (pthread_attr_t *attr);

int pthread_attr_setdetachstate (pthread_attr_t *attr, int detachstate);

int pthread_attr_getdetachstate (const pthread_attr_t *attr,
int *detachstate);

int pthread_attr_setstacksize (pthread_attr_t *attr, size_t stacksize);

int pthread_attr_getstacksize (const pthread_attr_t *attr,
size_t *stacksize);

int pthread_attr_setstackaddr (pthread_attr_t *attr, void *stackaddr);

int pthread_attr_getstackaddr (const pthread_attr_t *attr,
void **stackaddr);
```

IMPLEMENTATION

Cray PVP systems systems

STANDARDS

PThreads

DESCRIPTION

The `pthread_create` function creates a new thread, with attributes (see below) specified by *attr*, within a process. If *attr* is a null value, the default attributes are used. Upon successful completion, `pthread_create` stores the ID of the created thread in the location referenced by *thread*.

The thread begins execution in *start_routine* with *arg* as its sole argument. If *start_routine* returns, its effect is as though `pthread_exit` had been called using the return value of *start_routine* as the exit status. The new thread inherits its signal mask from the creating thread.

The `pthread_join` function suspends execution of the calling thread until the target *thread* terminates, unless the target thread has already terminated. On return from a successful `pthread_join` call with a non-null *value_ptr* argument, the value passed to `pthread_exit` by the terminating thread is made available in the location referenced by *value_ptr*. When a `pthread_join` function returns successfully, the target thread has been terminated. An attempt to call `pthread_join` on a detached target thread returns an error. Only one thread can call `pthread_join` for a given thread.

The `pthread_exit` function terminates the calling thread and makes the value *value_ptr* available to any successful join with the terminating thread. Any cancellation cleanup handlers that have been pushed and not yet popped shall be popped in the reverse order that they were pushed and then executed. After all cancellation cleanup handlers have been executed, if the thread has any thread-specific data, any destructor functions previously specified are called.

The `pthread_exit` function cannot be called from a cancellation cleanup handler or destructor function that was invoked as a result of either an implicit or explicit call to `pthread_exit`.

The `pthread_self` routine returns the thread ID of the caller.

The `pthread_equal` function compares the thread IDs *t1* and *t2*.

The `pthread_detach` function detaches the calling thread. Storage for the thread *thread* is reclaimed when that thread terminates.

The first call to `pthread_once` by any thread in a process, with a given *once_control*, calls the *init_routine* with no arguments. Subsequent calls to `pthread_once` with the same *once_control* do not call the *init_routine*. On return from `pthread_once`, *init_routine* is guaranteed to be completed. The *once_control* parameter determines whether the associated initialization routine has been called.

The behavior of `pthread_once` is not as described here if *once_control* has automatic storage duration or is not initialized by `PTHREAD_ONCE_INIT`.

The `pthread_attr_init` function initializes a thread attributes object *attr* with the default value for all of the individual attributes used by a given implementation.

The resulting thread attributes object (possibly modified by setting individual attribute values), when used by `pthread_create`, defines the attributes of the thread created. A single attributes object can be used in multiple simultaneous calls to `pthread_create`.

Once an attributes object is no longer needed, `pthread_attr_destroy` should be called to ensure that any system resources are released. A thread attributes object can be destroyed while threads that were created with that attributes object are executing. After calling `pthread_attr_destroy`, the thread attributes object cannot be used as an argument to `pthread_create`.

The *detachstate* attribute controls whether the thread is created in a detached state. If the thread is created detached, then the ID of the newly created thread cannot be specified to the `pthread_join` function.

The `pthread_attr_setdetachstate` and `pthread_attr_getdetachstate` functions set and get the *detachstate* attribute, respectively, in the *attr* object. The *detachstate* attribute is set to either `PTHREAD_CREATE_DETACHED` or `PTHREAD_CREATE_JOINABLE`. A value of `PTHREAD_CREATE_DETACHED` causes all threads created with *attr* to be in the detached state; using a value of `PTHREAD_CREATE_JOINABLE` causes all threads created with *attr* to be in the joinable state. The default value of the *detachstate* attribute is `PTHREAD_CREATE_JOINABLE`.

The `pthread_attr_setstacksize` and `pthread_attr_getstacksize` functions set and get the *stacksize* attribute, respectively, in the *attr* object. The `pthread_attr_setstackaddr` and `pthread_attr_getstackaddr` functions set and get the *stackaddr* attribute, respectively, in the *attr* object. Since neither of these attributes are supported, these functions return `ENOSYS` if called.

RETURN VALUES

The `pthread_exit` function does not return a value.

The `pthread_equal` function returns a nonzero value if *t1* and *t2* are equal; otherwise, 0 is returned.

If successful, all the other functions return 0. Otherwise, an error number is returned to indicate the error.

ERRORS

If any of the following conditions occur, the `pthread_create` function returns the corresponding error numbers:

`EAGAIN` The system lacked the necessary resources to create another thread, or the system-imposed limit on the total number of threads in a process was exceeded.

`EINVAL` The specified attributes are invalid.

If any of the following conditions occur, the `pthread_join` and `pthread_detach` functions return the corresponding error number:

`EINVAL` The specified thread is detached.

`ESRCH` No thread could be found corresponding to that specified by the given thread ID.

If any of the following conditions occur, the `pthread_join` function returns the corresponding error number:

`EDEADLK` The value of *thread* specifies the calling thread.

If any of the following conditions occur, the `pthread_attr_setdetachstate` function returns the corresponding error number:

`EINVAL` The value of *detachstate* was invalid.

If any of the following conditions occur, the `pthread_attr_init` function returns the corresponding error number:

`ENOMEM` Insufficient memory exists to create the thread attributes object.

If any of the following conditions occur, the `pthread_attr_getstacksize`, `pthread_attr_setstacksize`, `pthread_attr_getstackaddr`, and `pthread_attr_setstackaddr` functions return the corresponding error number:

`ENOSYS` The *stacksize* or *stackaddr* attributes are not defined.

SEE ALSO

`pthread_atfork`(3C), `pthread_cancel`(3C), `pthread_cond`(3C), `pthread_kill`(3C), `pthread_mutex`(3C), `pthread_spec`(3C)

NAME

pthread_atfork – Register fork handlers

SYNOPSIS

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

int pthread_atfork (void (*prepare)(void), void (*parent)(void), void
(*child)(void));
```

IMPLEMENTATION

Cray PVP systems systems

STANDARDS

PThreads

DESCRIPTION

The pthread_atfork function shall declare fork handlers to be called before and after fork, in the context of the thread that called fork. The *prepare* fork handler shall be called before fork processing commences. The *parent* fork handler shall be called after fork processing completes in the parent process. The *child* fork handler shall be called after fork processing completes in the child process. If no handling is desired at one or more of these three points, the corresponding fork handler address(es) may be set to null.

The order of calls to pthread_atfork is significant. The *parent* and *child* fork handlers shall be called in the order in which they were established by calls to pthread_atfork. The *prepare* fork handlers shall be called in the opposite order.

RETURN VALUES

Upon successful completion, the pthread_atfork function shall return 0. Otherwise, an error number is returned to indicate the error.

ERRORS

If any of the following conditions occur, the pthread_atfork function shall return the corresponding error number:

ENOMEM Insufficient table space exists to record the fork handler addresses.

SEE ALSO

fork(2) in the

NAME

pthread_condattr_init, pthread_condattr_destroy, pthread_cond_init, pthread_cond_destroy, pthread_cond_signal, pthread_cond_broadcast, pthread_cond_wait, pthread_cond_timedwait – Condition variables

SYNOPSIS

```
#include <pthread.h>

int pthread_condattr_init (pthread_condattr_t *attr);

int pthread_condattr_destroy (pthread_condattr_t *attr);

int pthread_cond_init (pthread_cond_t *cond,
 const pthread_condattr_t *attr);

int pthread_cond_destroy (pthread_cond_t *cond);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int pthread_cond_signal (pthread_cond_t *cond);

int pthread_cond_broadcast (pthread_cond_t *cond);

int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex);

int pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mutex,
const struct timespec *abstime);
```

IMPLEMENTATION

Cray PVP systems systems

STANDARDS

PThreads

DESCRIPTION

The function pthread_condattr_init initializes *attr*, a *condition variable attributes object* (CVAO), using the default value for all attributes. Currently there are no supported attributes. Attempting to initialize an already initialized CVAO results in undefined behavior.

The pthread_condattr_destroy function destroys the specified CVAO. This allows the system to reclaim any resources used by the CVAO.

The pthread_cond_init function initializes the condition variable referenced by *cond* with attributes referenced by *attr*. If *attr* is a null value, the default condition variable attributes are used; this is the same as passing the address of a default CVAO. Upon successful initialization, the state of the condition variable becomes initialized. Attempting to initialize an already initialized condition variable results in undefined behavior.

After a CVAO initializes one or more condition variables, any function affecting the CVAO (including destruction) has no effect on any previously initialized condition variables.

The `pthread_cond_destroy` function destroys the given condition variable specified by *cond*; the object becomes, in effect, uninitialized. A destroyed condition variable object can be reinitialized using `pthread_cond_init`; the results of otherwise referencing the object after it has been destroyed are undefined.

Attempting to destroy a condition variable upon which other threads are currently blocked results in undefined behavior.

In cases for which default condition variable attributes are appropriate, the macro `PTHREAD_COND_INITIALIZER` can initialize condition variables that are statically allocated. This is equivalent to dynamic initialization by a call to `pthread_cond_init` with parameter *attr* specified as a null value, except that no error checks are performed.

The `pthread_cond_signal` call unblocks at least one of the threads that are blocked on the specified condition variable *cond* (if any threads are blocked on *cond*).

The `pthread_cond_broadcast` call unblocks all threads currently blocked on the specified condition variable *cond*.

If more than one thread is blocked on a condition variable, the scheduling policy determines the order in which threads are unblocked. When each thread unblocked as a result of a `pthread_cond_signal` or the `pthread_cond_broadcast` call returns from its call to `pthread_cond_wait` or `pthread_cond_timedwait`, the thread owns the mutex with which it called `pthread_cond_wait` or `pthread_cond_timedwait`. The unblocked thread(s) contend for the mutex as if each had called `pthread_mutex_lock`.

The `pthread_cond_signal` or `pthread_cond_broadcast` functions may be called by a thread, whether or not it currently owns the mutex that threads calling `pthread_cond_wait` or `pthread_cond_timedwait` have associated with the condition variable during their waits; however, if predictable scheduling behavior is required, that mutex is locked by the thread calling `pthread_cond_signal` or `pthread_cond_broadcast`.

The `pthread_cond_signal` and `pthread_cond_broadcast` functions have no effect if there are no threads currently blocked on *cond*.

The `pthread_cond_wait` and `pthread_cond_timedwait` functions are used to block on a condition variable. They are called with *mutex* locked by the thread, or undefined behavior will result.

These functions atomically release *mutex* and cause the calling thread to block on the condition variable *cond*. Atomically here means "atomically with respect to access by another thread to the mutex and then the condition variable." That is, if another thread can acquire the mutex after it is released by an about-to-block thread, a subsequent call to `pthread_cond_signal` or `pthread_cond_broadcast` in that thread behaves as if it were issued after the other thread has blocked. Upon successful return, the mutex is locked and is owned by the calling thread.

When condition variables are used, there is always a boolean predicate involving a shared variable for each condition wait, which is true if the `pthread_cond_timedwait` functions may occur. Since the return from `pthread_cond_wait` or `pthread_cond_timedwait` implies nothing about this predicate's value, the predicate should be reevaluated upon such return.

More than one mutex should not be used for concurrent `pthread_cond_wait` or `pthread_cond_timedwait` operations on the same condition variable, as this can result in improper behavior from these functions.

The `pthread_cond_timedwait` function is the same as `pthread_cond_wait`, except that an error is returned if the absolute time specified by *abstime* passes (that is, if system time equals or exceeds *abstime*) before the condition *cond* is signaled or broadcast, or if the absolute time specified by *abstime* has already been passed at the time of the call. When such time-outs occur, `pthread_cond_timedwait` nonetheless releases and reacquires the mutex referenced by *mutex*.

RETURN VALUES

If successful, all of these functions return 0. Otherwise, an error number is returned to indicate the error.

MESSAGES

If any of the following conditions occur, the `pthread_condattr_init` function returns the corresponding error number:

`ENOMEM` Insufficient memory exists to initialize the condition variable attributes object.

If any of the following conditions occur, the `pthread_cond_init` function returns the corresponding error number:

`EAGAIN` The system lacked the necessary resources (other than memory) to initialize another condition variable.

`ENOMEM` Insufficient memory exists to initialize the condition variable.

If any of the following conditions occur, the `pthread_cond_timedwait` function returns the corresponding error number.

`ETIMEDOUT` The time specified by *abstime* to `pthread_cond_timedwait` has passed.

SEE ALSO

`pthread`(3C), `pthread_spec`(3C), `pthread_mutex`(3C)

NAME

pthread_mutexattr_init, pthread_mutexattr_destroy, pthread_mutexattr_setkind_np, pthread_mutexattr_getkind_np, pthread_mutex_init, pthread_mutex_destroy, pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock – Mutual exclusion

SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_init (pthread_mutexattr_t*attr);

int pthread_mutexattr_destroy (pthread_mutexattr_t*attr);

int pthread_mutexattr_setkind_np (pthread_mutexattr_t*attr, int kind);

int pthread_mutexattr_getkind_np (const pthread_mutexattr_t *attr);

int pthread_mutex_init (pthread_mutex_t*mutex,
const pthread_mutexattr_t*attr);

int pthread_mutex_destroy (pthread_mutex_t*mutex);

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_lock (pthread_mutex_t*mutex);

int pthread_mutex_trylock (pthread_mutex_t*mutex);

int pthread_mutex_unlock (pthread_mutex_t*mutex);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

PThreads

DESCRIPTION

The pthread_mutexattr_init function initializes the specified mutex attributes object to the default values.

The pthread_mutexattr_destroy function should be called when the attributes object is no longer needed in order to release any system resources associated with the object. The attributes object can be destroyed even when there are active mutexes created with that attributes object. Once destroyed, the attributes object cannot be used.

The pthread_mutexattr_setkind_np and pthread_mutexattr_getkind_np functions set or get the mutex type. The mutex type can be MUTEX_FAST_NP, MUTEX_NONRECURSIVE_NP, or MUTEX_RECURSIVE_NP. A mutex with type MUTEX_FAST_NP is a simple mutex lock which does no error checking. A mutex with type MUTEX_NONRECURSIVE_NP provides additional error checking. Both of these mutexes will block if the owner of the mutex attempts to lock the mutex a second time. If, instead, attempts to lock the mutex should be nested, then the mutex can be initialized as a MUTEX_RECURSIVE_NP mutex. The default mute type is MUTEX_FAST_NP.

The pthread_mutex_init function initializes the mutex referenced by *mutex* with attributes specified by *attr*. If *attr* is a null value, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object. On successful initialization, the state of the mutex becomes initialized and unlocked. Attempting to initialize an already initialized mutex results in undefined behavior.

The pthread_mutex_destroy function destroys the mutex object referenced by *mutex*; the mutex object becomes effectively uninitialized. A destroyed mutex object can be reinitialized using pthread_mutex_init; the results of otherwise referencing the object after it has been destroyed are undefined.

It is invalid to destroy a locked mutex.

For cases in which default mutex attributes are appropriate, the macro PTHREAD_MUTEX_INITIALIZER can be used to initialize mutexes that are statically allocated. This is equivalent to dynamic initialization by a call to pthread_mutex_init, with parameter *attr* specified as a null value, except that no error checks are performed.

The *mutex* object is locked by a call to pthread_mutex_lock. If the mutex is already locked, the calling thread blocks until the mutex becomes available. The operation returns with *mutex* in the locked state, with the calling thread as its owner.

The function pthread_mutex_trylock is identical to pthread_mutex_lock except that, if *mutex* is currently locked (by any thread including the current thread), the call returns immediately.

The pthread_mutex_unlock function is called by the owner of *mutex* to release it. It is invalid to call pthread_mutex_unlock from a thread that is not the owner of the mutex. Calling pthread_mutex_unlock when *mutex* is unlocked is also invalid. If there are threads blocked on *mutex* when pthread_mutex_unlock is called, the mutex becomes available, and the scheduling policy is used to determine which thread shall acquire the mutex.

RETURN VALUES

If successful, the pthread_mutexattr_init, pthread_mutexattr_destroy, and pthread_mutexattr_setkind_np functions return 0. Otherwise, an error number indicates the error.

The pthread_mutexattr_getkind_np function returns the mutex type of the specified attributes object.

If successful, the `pthread_mutex_init` and `pthread_mutex_destroy` functions return 0. Otherwise, an error number indicates the error.

If successful, the `pthread_mutex_lock` and `pthread_mutex_unlock` functions return 0. Otherwise, an error number is returned to indicate the error.

The function `pthread_mutex_trylock` returns 0 if a lock on *mutex* is acquired. Otherwise, an error number indicates the error.

MESSAGES

If any of the following conditions occur, the `pthread_mutexattr_init` function returns the corresponding error number:

`ENOMEM` The system lacked the necessary resources (other than memory) to initialize the mutex attributes object.

If any of the following conditions occur, the `pthread_mutexattr_setkind_np` function returns the corresponding error number:

`EINVAL` The mutex type specified by *kind* is invalid.

If any of the following conditions occur, the `pthread_mutex_init` function returns the corresponding error number:

`EAGAIN` The system lacked the necessary resources (other than memory) to initialize another mutex.

`ENOMEM` Insufficient memory exists to initialize the mutex.

If the following condition occurs, the `pthread_mutex_trylock` function returns the corresponding error number:

`EINVAL` The mutex could not be acquired because it was already locked.

SEE ALSO

`pthread`(3C), `pthread_cond`(3C), `pthread_spec`(3C)

NAME

pthread_key_create, pthread_key_delete, pthread_setspecific, pthread_getspecific – Thread-specific data

SYNOPSIS

```
#include <pthread.h>
int pthread_key_create (pthread_key_t *key, void (*destructor) (void *));
int pthread_key_delete (pthread_key_t key);
int pthread_setspecific (pthread_key_t key, const void *value);
void *pthread_getspecific (pthread_key_t key);
```

IMPLEMENTATION

Cray PVP systems systems

STANDARDS

PThreads

DESCRIPTION

The pthread_key_create function creates a thread-specific data key visible to all threads in the process. Key values provided by pthread_key_create are opaque objects used to locate thread-specific data. Although the same key value may be used by different threads, the values bound to the key by pthread_setspecific are maintained on a per-thread basis and persist for the life of the calling thread.

When a key is created, it is given a null value in all active threads. When a thread is created, all defined keys in the new thread are given null values.

An optional destructor function may be associated with each key value. At thread exit, if a key value has a non-null destructor pointer, and the thread has a non-null value associated with that key, the function pointed to is called with the current associated value as its sole argument. The order of destructor calls is unspecified if more than one destructor exists for a thread when it exits.

The pthread_key_delete function deletes a thread-specific data key previously returned by pthread_key_create. The thread-specific data values associated with *key* need not be null when pthread_key_delete is called. The application must free storage or perform cleanup actions for data structures related to the deleted key or associated thread-specific data in any threads; this cleanup can be done either before or after pthread_key_delete is called. The specified *key* cannot be used following the call to pthread_key delete. No destructor functions are invoked by pthread_key_delete.

The pthread_setspecific function associates a thread-specific value with a *key* obtained via a previous call to pthread_key_create. Different threads may bind different values to the same key. The pthread_setspecific function cannot be called from a destructor function, as this may result in lost storage or infinite loops.

The pthread_getspecific function returns the value currently bound to the specified *key* on behalf of the calling thread.

RETURN VALUES

The function pthread_getspecific returns the thread-specific data value for *key*. If *key* has no thread-specific data value, a null value is returned.

If successful, the pthread_setspecific, pthread_key_create, and pthread_key_delete functions return 0. Otherwise, an error number indicates the error.

MESSAGES

If any of the following conditions occur, the pthread_key_create function returns the corresponding error number:

EAGAIN The system lacked the necessary resources to create another thread-specific data key, or the system-imposed limit on the total number of keys per process, PTHREAD_KEYS_MAX, has been exceeded.

ENOMEM Insufficient memory exists to create the key.

If any of the following conditions occur, the pthread_key_delete function returns the corresponding error number:

EINVAL The *key* value is invalid.

If any of the following conditions occur, the pthread_setspecific function returns the corresponding error number:

ENOMEM Insufficient memory exists to associate the value with the *key*.

EINVAL The *key* value is invalid.

SEE ALSO

pthread(3C), pthread_cond(3C), pthread_mutex(3C)

NAME

putc, putchar, fputc, putc_unlocked, putchar_unlocked, putw, fputwc, putwc, putwchar – Puts a character or word on a stream

SYNOPSIS

```
#include <stdio.h>
int fputc (int c, FILE *stream);
int putc (int c, FILE *stream);
int putchar (int c);
int putc_unlocked (int c, FILE *stream);
int putchar_unlocked (int c);
int putw (int w, FILE*stream);
#include <wchar.h>
wint_t fputwc (wint_t wc, FILE *stream);
wint_t putwc (wint_t wc, FILE *stream);
wint_t putwchar (wint_t wc);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (fputc, putc, and putchar only)
POSIX (putc_unlocked and putchar_unlocked only)
XPG4 (putw, fputwc, ungetwc, and putwchar only)

DESCRIPTION

The fputc function writes the character specified by c (converted to an unsigned char) to the output stream pointed to by stream, at the position indicated by the associated file position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream.

The fputc function behaves like putc, but it runs more slowly than putc, and it takes less space per invocation.

The putc function is equivalent to fputc, except that if it is implemented as a macro, it may evaluate stream more than once, so the argument should never be an expression with side effects. In particular,

```
putc(c,*f++)
```

does not work as expected; use fputc instead.

The putchar function is equivalent to putc with the second argument stdout.

The putc_unlocked and putchar_unlocked functions provide functionality equivalent to the putc and putchar 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 putw function writes the word (that is, type int) *w* to the output *stream* (at the position at which the file pointer, if defined, is pointing). The size of a word is the size of a type int and varies from machine to machine. The putw function neither assumes nor causes special alignment in the file.

The fputwc function writes the character corresponding to the wide-character code *wc* to the output stream to which *stream* points, at the position indicated by the associated file-position indicator for the stream (if defined), and it advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened while writing the character, the shift state of the output file is left in an undefined state. The st_ctime and st_mtime fields of the file are marked for update between the successful execution of fputwc and the next successful completion of a call to fflush(2) or fclose(2) on the same stream or a call to exit or abort.

The putwc function is equivalent to fputwc, except that if it is implemented as a macro it may evaluate *stream* more than once, so the argument should never be an expression with side effects. Therefore, you should not use this function; use the fputwc() function instead.

The function call putwchar(*wc*) is equivalent to putwc(*wc, stdout*).

CAUTIONS

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

NOTES

The macro version of the putc 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.

RETURN VALUES

If successful, these functions each return the value they have written. If a write error occurs, the error indicator for the stream is set and the functions return EOF (WEOF for fputwc). The latter occurs if the file *stream* is not open for writing, or if the output file cannot be created. Because EOF is a valid integer, use the ferror(3C) function to detect putw errors.

SEE ALSO

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

NAME

putenv – Changes or adds value to the environment

SYNOPSIS

```
#include <stdlib.h>
int putenv (const char *string);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The putenv function makes the value of the environment variable name equal to *value* by altering an existing variable or creating a new one. In either case, the string to which *string* points becomes part of the environment, so altering the string changes the environment. The *string* argument points to a string of the form "name=value" that contains no embedded blanks. The space that *string* uses is no longer used after a new string-defining name is passed to putenv.

NOTES

On Cray MPP systems, each processing element (PE) gets a separate copy of the environment; therefore, alterations to the environment by using putenv on a single PE will not be reflected on other PEs.

The putenv function manipulates the environment to which sh(1)points, and it can be used in conjunction with the getenv(3C) function; however, *envp* (the third argument to main) is not changed.

This function uses malloc(3C) to enlarge the environment.

After putenv is called, environmental variables are not necessarily in alphabetical order.

Calling putenv with an automatic variable as the argument, then exiting the calling function while *string* is still part of the environment is a potential error.

RETURN VALUES

If putenv could not obtain enough space (using malloc(3C)) for an expanded environment, it returns a nonzero value; otherwise, it returns 0.

FORTRAN EXTENSIONS

You also may call the putenv function from Fortran programs, as follows:

```
INTEGER*8 PUTENV, I
CHARACTER *n string
I = PUTENV(string)
```

Argument *string* can be either a Fortran character variable or an integer variable of the form *name=value*. If you use an integer variable, the data must be packed 8 characters per word and terminated with a null (0) byte.

Fortran function PUTENV allocates space and copies *string* to that space. Therefore, altering *string* after calling PUTENV does not change the environment.

SEE ALSO

getenv(3C), malloc(3C), setenv(3C)

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

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

NAME

putpwent – Writes password file entry

SYNOPSIS

```
#include <pwd.h>
int putpwent (struct passwd *p, FILE *f);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

AT&T extension

DESCRIPTION

The putpwent function is the inverse of getpwent. Given a pointer to a passwd structure created by getpwent (or getpwuid or getpwnam), putpwent writes a line on the stream *f*, which must match the format of /etc/passwd (see passwd(5)).

NOTES

This function is included only for compatibility with previous systems. Writing something in /etc/passwd does not make an entry in the user information database and so is ineffective. The passwd file is automatically maintained by udbgen(8).

WARNINGS

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

RETURN VALUES

If an error is detected during its operation, putpwent returns nonzero; otherwise, it returns 0.

SEE ALSO

getpwent(3C)

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

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

NAME

puts, fputs, fputws – Puts a string on a stream

SYNOPSIS

```
#include <stdio.h>
int puts (const char *s);
int fputs (const char *s, FILE *stream);
#include <wchar.h>
int fputws(const wchar_t *ws, FILE *stream);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (puts and fputs only)
XPG4 (fputws only)

DESCRIPTION

The puts function writes the null-terminated string to which *s* points, followed by a newline character, to the standard output stream stdout. The fputs function writes the null-terminated string to which *s* points to the specified output *stream*. Neither function writes the terminating null character.

The fputws function writes a character string that corresponds to the (null-terminated) wide-character string to which *ws* points to the stream to which *stream* points. No character that corresponds to the terminating null, wide-character code is written. The st_ctime and st_mtime fields of the file are marked for update between execution of fputws and completion of the next call to fflush(2) or fclose(2) on the same stream or a call to exit or abort.

NOTES

The puts function appends a newline character; fputs does not.

RETURN VALUES

These functions return a nonnegative value. The puts and fputs functions return an end of file (EOF) on error (if they try to write on a file that has not been opened for writing). The fputws function, on an error, returns – 1, sets an error indicator for the stream, and sets errno to indicate the error.

FORTRAN EXTENSIONS

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

```
INTEGER*8 FPUTS, stream, I
I = FPUTS(s, stream)
```

Argument *s* must be left-justified, word-aligned, and terminated by a null byte.

SEE ALSO

`ferror`(3C), `fopen`(3C), `fread`(3C), `printf`(3C), `putc`(3C)

NAME

qsort – Performs sort

SYNOPSIS

```
#include <stdlib.h>

void qsort (void *base, size_t nmemb, size_t size, int (*compar)(const void *,
const void *));
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The qsort function sorts an array of *nmemb* objects, the initial element of which is pointed to by *base*. The *size* argument specifies the size of each object.

The contents of the array are sorted into ascending order according to a comparison function to which *compar* points, which is called with two arguments that point to the objects being compared. The function returns an integer that is less than, equal to, or greater than 0 if the first argument is considered to be respectively less than, equal to, or greater than the second.

If two elements compare as equal, their order in the sorted array is unspecified.

NOTES

The comparison function's arguments should be of type void* and should be cast back to type pointer-to-element within the function. The comparison function need not compare every byte; therefore, arbitrary data may be contained in the elements in addition to the values being compared.

The output order of two items that compare as equal is unpredictable.

RETURN VALUES

The qsort function returns no value.

EXAMPLES

The following example shows how the qsort function executes:

```
#include <stdlib.h>

struct element {        /* array of elements to be sorted */
        int key;        /* key to sort on */
        .
```

```
        .
        .
} q[nel];

struct element *base = &q[0];

int compar(const void *a, const void *b)
                        /* comparsion function for qsort() */
{
        return (((struct element *)a)->key - ((struct element *)b)->key);
}

main() {
         .
         .
         .
         qsort(base, nel, sizeof(*base), compar);
         .
         .
         .
}
```

SEE ALSO

bsearch(3C), lsearch(3C)

NAME

raise – Sends a signal to the executing program

SYNOPSIS

```
#include <signal.h>
int raise (int sig);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The raise function sends the signal *sig* to the executing program.

RETURN VALUES

The raise function returns 0 if successful, nonzero if unsuccessful.

NAME

rand, srand, rand_r – Generates pseudo-random integers

SYNOPSIS

```
#include <stdlib.h>
int rand (void);
void srand (unsigned int seed);
int rand_r (unsigned int *seedptr);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (rand and srand)
PThreads (rand_r)

DESCRIPTION

The rand function computes a sequence of pseudo-random integers in the range 0 to RAND_MAX, which is defined in the header file stdlib.h.

The srand function uses the *seed* argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to rand. If srand is then called with the same seed value, the sequence of pseudo-random numbers is repeated. If rand is called before any calls to srand have been made, the same sequence is generated as when srand is first called with a seed value of 1.

The rand_r function provides functionality equivalent to the rand function but with an interface that is safe for multitasked applications. It takes a pointer to the seed value (*seedptr*) as an argument. This function allows for easy maintenance of separate random number generators and safe management of random number sequences for multitasked applications.

RETURN VALUES

The rand and rand_r functions return a pseudo-random integer.

The srand function returns no value.

SEE ALSO

drand48(3C)

NAME

rcmd, rresvport, ruserok – Returns a stream to a remote command

SYNOPSIS

```
#include <unistd.h>

int rcmd (char **ahost, unsigned short inport, char *locuser, char *remuser, char
*cmd, int fd2p);

int rresvport (int *port);

int ruserok (char *rhost, int superuser, char *ruser, char *luser);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The rcmd function allows the super user to execute a remote command *cmd* on a remote machine, using an authentication scheme based on reserved port numbers. The rresvport function returns a descriptor to a socket with an address in the privileged port space. Servers on the local host use the ruserok function to authenticate users on a remote host who request service by means of the rcmd function. All three functions are present in the same file and are used by the rshd(8) server (among others).

The rcmd function uses the gethostbyname function (see gethost(3C)), to look up the host *ahost*, returning –1 if the host does not exist. Otherwise, *ahost* is set to the official name of the remote host, and a connection is established to a server residing at the Internet port *inport*.

If the rcmd function succeeds, a socket of type SOCK_STREAM is returned to the caller and given to the remote command *cmd* as the file descriptors stdin (for reading from the socket) and stdout (for writing to the socket). If *fd2p* is nonzero, an auxiliary channel to a control process is set up, and a descriptor for it is placed in *fd2p*. The control process returns diagnostic output from the command (stderr) on this channel and also accepts bytes on this channel as being UNICOS signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, stderr is made the same as stdout. In this case, no provision is made for sending arbitrary signals to the remote process, although you may be able to establish contact by using out-of-band data.

The service request protocol and user validation are described in detail in rshd(8).

The rresvport function obtains a socket with a privileged address bound to it. This socket is suitable for use by rcmd and several other functions. Privileged addresses consist of a port in the range 0 through 1023. Only the super user is allowed to bind an address of this sort to a socket.

The ruserok function takes a remote host's name (*rhost*), as returned by the gethostent function (see gethost(3C)), the name of the remote user (*ruser*), the name of the local user (*luser*) whose account will be accessed by the remote user, and a flag (either 0 or 1) indicating if the local user's name is that of the super user. It then checks the local host's /etc/hosts.equiv file and the .rhosts file, if it exists, in the current directory (normally the local user's home directory) for authorization for the person requesting service.

NOTES

There is no way to specify options to the socket(2) call that rcmd makes.

RETURN VALUES

A 0 is returned if the name of the remote host is listed in the /etc/hosts.equiv file, or if the remote host name and remote user name are listed in the .rhosts file. The system configuration can require the /etc/hosts.equiv and .rhosts files each to contain a match for the remote host, and also require the remote user and local user names to match; otherwise, ruserok returns a –1. If the super user flag is 1, indicating that the specified local user is the super user, the checking of /etc/host.equiv is bypassed. (See hosts.equiv(5) and rhosts(5).) Also, if the rhosts file is writable by group or other, ruserok returns a –1.

FILES

/etc/hosts.equiv

$HOME/.rhosts

SEE ALSO

gethost(3C), rexec(3C)

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

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

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

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

TCP/IP Network User's Guide, Cray Research publication SG–2009

NAME

rcmdexec – Returns a stream to a remote command

SYNOPSIS

```
int rcmdexec (char **ahost, unsigned short rshellp, unsigned short rexecp, char
*locuser, char *remuser, char *passwd, char *cmd, int fd2p);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The rcmdexec function is a combination of the rcmd function and the rexec function. (See rcmd(3C) and rexec(3C) for additional argument information.)

Argument *rshellp* is the *inport* value for function rcmd, and *rexecp* is the *inport* value for function rexec.

The rcmdexec function first attempts a rcmd function call. If the internal connect(2) call fails with error ECONNREFUSED, rcmdexec immediately tries an rexec function call. If the internal connect(2) call again fails with error ECONNREFUSED, rcmdexec sleeps for a period of time, and goes back and tries both calls again. The total time out period is about 30 seconds.

SEE ALSO

rcmd(3C), rexec(3C)

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

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

NAME

re_comp, re_exec – Matches regular expressions

SYNOPSIS

```
#include <unistd.h>
char *re_comp (char *s);
int re_exec (char *s);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

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

If string *s* was compiled successfully, function re_comp returns 0; otherwise, a string that contains an error message is returned. If re_comp is passed 0 or a null string, it returns without changing the currently compiled regular expression.

If string *s* matches the last compiled regular expression, function re_exec returns 1; if string *s* failed to match the last compiled regular expression, it returns 0; if the compiled regular expression was not valid (indicating an internal error), it returns –1.

The strings passed to both re_comp and re_exec can have trailing or embedded newline characters; they are terminated by a null character. Taking account of these differences, the regular expressions recognized are described in the ed(1) man page.

RETURN VALUES

Function re_exec returns –1 for an internal error.

If an error occurs, function re_comp returns one of the following strings:

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

SEE ALSO

ed(1), egrep(1), ex(1), fgrep(1), grep(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

regcmp, regex – Compiles and executes a regular expression

SYNOPSIS

```
#include <stdlib.h>
char *regcmp (char *string, ...);
char *regex (char *re, char *subject, ...);
char *__loc1;
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

AT&T extension

DESCRIPTION

The regcmp function compiles a regular expression and returns a pointer to the compiled form. The malloc(3C) function creates space for the compiled regular expression; to free the allocated space, you must call the free (see malloc(3C)) function; regcmp returns NULL if if an incorrect argument is passed. regcmp returns NULL. The regcmp(1) command has been written to generally preclude the need for this function at execution time.

The regex function executes a compiled pattern against the subject string. Additional arguments are passed to receive values back. On failure, regex returns NULL; on success, it returns a pointer to the next unmatched character.

Functions regcmp and regex take a variable number of char * arguments following *string* and *subject*, respectively. The last argument to regcmp must be (char *)0. A global character pointer, __loc1, points to where the match began. Both regcmp and regex were mostly borrowed from the editor, ed(1); however, the syntax and semantics have been changed slightly. The following are the valid symbols and their associated meanings:

Symbol	Description
[ ] * . ^	These symbols retain their current meaning (as in ed(1)).
$	Matches the end of the string; \n matches a newline character.
-	Within brackets, the minus means *through* (for example, [a-z] is equivalent to [abcd. . .xyz]). The - can appear as itself only if used as the first or last character (for example, the character class expression []- matches the characters] and -).
+	A regular expression followed by + means *one or more times*; for example, [0-9]+ is equivalent to [0-9][0-9]*.

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

(. . .)$*n* The value of the enclosed regular expression is to be returned. The value is stored in the (*n*+1)th argument following the subject argument. At most, 10 enclosed regular expressions are allowed. The regex function makes its assignments unconditionally.

(. . .) Parentheses are used for grouping. An operator, such as "*", "+", "-", or "/", can work on a single character or a regular expression enclosed in parentheses; for example, (a*(cb+)*)$0.

By necessity, all of the preceding symbols are special; to be used as themselves, you must escape them by using a .

CAUTIONS

The user program may run out of memory if regcmp is called iteratively without freeing the compiled regular expressions no longer required.

EXAMPLES

Example 1: The following example matches a leading newline character in the subject string at which cursor points.

```
#include <stdlib.h>

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

Example 2: The following example matches through the string "Testing3" and returns the address of the character after the last matched character (cursor+11). The string "Testing3" is copied to the character array ret0.

```
#include <stdlib.h>

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

Example 3: The following example applies a precompiled regular expression `name` in `file.i` (see `regcmp`(1)) against `string`.

```
#include <stdlib.h>
#include "file.i"

char *string, *newcursor;
      .
      .
      .
newcursor = regex(name, string);
```

SEE ALSO

`malloc`(3C)

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

NAME

regcomp, regexec, regerror, regfree – Regular-expression library

SYNOPSIS

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

int regcomp(regex_t *preg, const char *pattern, int cflags);

int regexec(const regex_t *preg, const char *string, size_t nmatch,
   regmatch_t pmatch[], int eflags);

size_t regerror(int errcode, const regex_t *preg, char *errbuf,
   size_t errbuf_size);

void regfree(regex_t *preg);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

These functions implement regular expressions (REs); see regex(3C). Functions are used as follows:

Function	Description
regcomp	Compiles an RE written as a string into an internal form
regexec	Matches the regcomp value against a string and reports results
regerror	Transforms error codes from either regcomp or regexec into human-readable messages
regfree	Frees any dynamically allocated storage used by the internal form of an RE

The header file regex.h declares two structure types, regex_t and regmatch_t, the former for compiled internal forms and the latter for match reporting. It also declares the four functions, a type regoff_t, and a number of constants with names that start with REG_.

The regcomp function compiles the regular expression contained in the *pattern* string, subject to the flags in *cflags*, and places the results in the regex_t structure to which *preg* points. The cflags variable is the bitwise OR of 0 or more of the following flags:

Function	Description
REG_EXTENDED	Compile modern (extended) REs, rather than the obsolete (basic) REs that are the default.
REG_ICASE	Compile for matching that ignores upper/lower case distinctions. See regex(7).
REG_NOSUB	Compile for matching that must report only success or failure, not what was matched.

REG_NEWLINE	Compile for newline-sensitive matching. By default, newline is a completely ordinary character with no special meaning in either REs or strings. With this flag, [^ bracket expressions and . never match newline, a ^ anchor matches the null string after any newline in the string in addition to its normal function, and the $ anchor matches the null string before any newline in the string in addition to its normal function.
REG_WORDS	Compile for matching that treats < as the beginning of a word and > as the end of a word.

The regcomp function returns 0 and fills in the structure to which *preg* points. One member of that structure is publicized: re_nsub, of type size_t, contains the number of parenthesized subexpressions within the RE (except this member's value is undefined if the REG_NOSUB flag was used). If regcomp fails, it returns a nonzero error code; see ERROR CODES.

The regexec function matches the compiled RE to which *preg* points against the *string*, subject to the flags in *eflags*, and reports results by using *nmatch*, *pmatch*, and the returned value. The RE must have been compiled by a previous invocation of regcomp. The compiled form is unchanged during execution of regexec; therefore, one compiled RE can be used simultaneously by multiple threads.

By default, the null-terminated string to which *string* points is considered to be the text of an entire line, minus any terminating newline. The *eflags* argument is the bitwise OR of 0 or more of the following flags:

Flag	**Description**
REG_NOTBOL	The first character of the string is not the beginning of a line, so the ^ anchor should not match before it. This condition does not affect the behavior of newlines under REG_NEWLINE.
REG_NOTEOL	The null terminating the string does not end a line, so the $ anchor should not match before it. This condition does not affect the behavior of newlines under REG_NEWLINE.
REG_STARTEND	The string is considered to start at *string* + *pmatch*[0].*rm_so* and to have a terminating NUL located at *string* + *pmatch*[0].*rm_eo* (NUL is not required at that location), regardless of the value of *nmatch*. The *pmatch* and *nmatch* variables are defined below. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be ported to other systems.

For a discussion of what is matched in cases in which an RE or a portion thereof could match any of several substrings of *string*, see regex(3C).

Usually, regexec returns 0 for success and the nonzero code REG_NOMATCH for failure. Other nonzero error codes may be returned in exceptional situations; see ERROR CODES.

If REG_NOSUB was specified in the compilation of the RE, or if *nmatch* is 0, regexec ignores the *pmatch* argument. Otherwise, *pmatch* points to an array of *nmatch* structures of type *regmatch_t*. Such a structure has at least the members *rm_so* and *rm_eo*, both of type *regoff_t* (a signed arithmetic type at least as large as an *off_t* and a *ssize_t*), containing respectively the offset of the first character of a substring and the offset of the first character after the end of the substring. Offsets are measured from the beginning of the *string* argument given to regexec. An empty substring is denoted by equal offsets, both indicating the character following the empty substring.

The 0th member of the *pmatch* array is filled in to indicate the substring of *string* that was matched by the entire RE. Remaining members report the substring that was matched by parenthesized subexpressions within the RE; member *i* reports subexpression *i*, with subexpressions counted (starting at 1) by the order of their opening parentheses in the RE, left to right. The *rm_so* and *rm_eo* arguments are set to –1 for unused entries in the array (entries that correspond either to subexpressions not used in the match, or to subexpressions that are not in the RE (that is, *i* > *preg* –> *re_nsub*). If a subexpression is used in the match several times, the reported substring is the last one it matched. When the RE `(b*)+` matches `bbb`, the parenthesized subexpression matches each of the three `b`s and then an infinite number of empty strings following the last `b`, so the reported substring is one of the empty strings.)

The `regerror` function maps a nonzero *errcode* from either `regcomp` or `regexec` to a printable message. If *preg* is nonnull, the error code should have arisen from use of the `regex_t` to which *preg* points, and if the error code came from `regcomp`, it should have been the result from the most recent `regcomp` using that `regex_t`. The `regerror` function may be able to supply a more detailed message by using information from the `regex_t`. The `regerror` function places the null-terminated message into the buffer to which *errbuf* points, limiting the length (including the null) to at most *errbuf_size* bytes. If the whole message does not fit, as much of it as will fit before the terminating null is supplied. In any case, the returned value is the size of buffer needed to hold the whole message (including terminating null). If *errbuf_size* is 0, *errbuf* is ignored, but the return value is still correct.

The `regfree` function frees any dynamically allocated storage associated with the compiled RE to which *preg* points. The remaining `regex_t` is no longer a valid compiled RE, and the effect of supplying it to `regexec` or *regerror* is undefined.

Implementation Choices

Many details are optional under POSIX, either explicitly undefined or forbidden by the RE grammar. The Cray Research implementation treats them as follows. For a discussion of the definition of case-independent matching, see `regex`(3C).

- No particular limit exists on the length of REs, except insofar as memory is limited. Memory usage is approximately linear in RE size, and largely insensitive to RE complexity, except for bounded repetitions. See BUGS for one short RE using them that will run almost any system out of memory.
- Any backslashed character other than the ones specifically legitimized by POSIX produces a `REG_EESCAPE` error.
- Any unmatched `[` is a `REG_EBRACK` error.
- Equivalence classes cannot begin or end bracket-expression ranges. The endpoint of one range cannot begin another.
- `RE_DUP_MAX`, the limit on repetition counts in bounded repetitions, is 255.
- A repetition operator (`?`, `*`, `+`, or bounds) cannot follow another repetition operator. A repetition operator cannot begin an expression, or subexpression, or follow `^` or `|`.
- A `|` cannot appear first or last in a (sub)expression or after another `|` (for example, an operand of `|` cannot be an empty subexpression). An empty parenthesized subexpression, `()`, is legal and matches an empty (sub)string. An empty string is not a legal RE.

- A { followed by a digit is considered the beginning of bounds for a bounded repetition, which must then follow the syntax for bounds. A { **not** followed by a digit is considered an ordinary character.
- A ^ and $ beginning and ending subexpressions in obsolete (basic) REs are anchors, not ordinary characters.

NOTES

One known functionality bug exists. The implementation of internationalization is incomplete; the locale is always assumed to be the POSIX default, and only the collating elements and other such elements of that locale are available.

ERROR CODES

Nonzero error codes from `regcomp` and `regexec` include the following:

Error Code	Description
REG_BADBR	Repetition count(s) in { } not valid
REG_BADPAT	Regular expression not valid
REG_BADRPT	?, *, or + operand not valid
REG_EBRACE	Braces { } not balanced
REG_EBRACK	Brackets [ ] not balanced
REG_ECOLLATE	Collating element not valid
REG_ECTYPE	Character class not valid
REG_EESCAPE	A \ applied to unescapable character
REG_EFATAL	Internal error
REG_ENEWLINE	A \n found before end of pattern
REG_ENOSYS	Function not supported
REG_ENSUB	More than nine ( ) pairs
REG_EPAREN	Parentheses ( ) not balanced
REG_ERANGE	Character range in [ ] not valid
REG_ESPACE	Ran out of memory
REG_STACK	Backtrack stack overflow
REG_ESUBREG	Backreference number not valid
REG_NOMATCH	regexec() failed to match

SEE ALSO

regex(3C)

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

NAME

regexp.h – Library header for regular expression compile and match functions

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

MACROS

None

TYPES

None

FUNCTION DECLARATIONS

The following functions are declared in regexp.h:

```
char *compile (char*instring, char*expbuf, const char *endbuf, int eof);
int advance (const char *string, const char *expbuf);
int step (const char *string, char *expbuf);
```

and the following are declared as external variables:

```
extern char *loc1, *loc2, *locs;
```

DESCRIPTION

These general-purpose regular expression matching functions in the form of ed(1), are defined in the header file regexp.h. Programs such as ed(1), sed(1), grep(1), expr(1), and so on, which perform regular expression matching, use this source file. In this way, only this file need be changed to maintain regular expression compatibility.

The interface to the header file regexp.h is excessively complex. Programs that include this file must have the following five macros declared before the #include <regexp.h> statement. These macros are used by the compile function.

Macro	Description
GETC( )	Returns the value of the next character in the regular expression pattern. Successive calls to GETC( ) should return successive characters of the regular expression.
PEEKC( )	Returns the next character in the regular expression. Successive calls to PEEKC( ) should return the same character (which should also be the next character returned by GETC( )).

`UNGETC(`*c*`)` Causes the argument *c* to be returned by the next call to `GETC( )` (and `PEEKC( )`). No more than one character of pushback is ever needed and this character is guaranteed to be the last character read by `GETC( )`. The value of the macro `UNGETC(`*c*`)` is always ignored.

`RETURN(`*pointer*`)` Used on normal exit of the `compile` function. The value of the argument *pointer* is a pointer to the character after the last character of the compiled regular expression. This macro is useful to programs that have to manage memory allocation.

`ERROR(`*val*`)` Used on abnormal return from the `compile` function. The argument *val* is an error number (see the following table for meanings). This call should never return.

Error	**Meaning**
11	Range endpoint too large
16	Bad number
25	''\ digit'' out-of-range
36	Illegal or missing delimiter
41	No remembered search string
42	`\( \)` imbalance
43	Too many `\(`
44	More than two numbers given in `\{ \}`
45	`}` expected after `\`
46	First number exceeds second in `\{ \}`
49	`[ ]` imbalance
50	Regular expression overflow

Arguments to the `compile` function are as follows:

Argument	**Description**
instring	Never used explicitly by the `compile` function but is useful for programs that pass down different pointers to input characters. It is sometimes used in the `INIT` declaration (see following explanation). Programs that call functions to input characters or have characters in an external array can pass down a value of `((char *) 0)` for this parameter.
expbuf	Character pointer to the place where the compiled regular expression will be placed.
endbuf	One more than the highest address where the compiled regular expression may be placed. If the compiled expression cannot fit in (*endbuf-expbuf*) bytes, a call to `ERROR(50)` is made.
eof	Character that marks the end of the regular expression; for example, in `ed`(1), this character is usually a `/`.

Each program that includes this file must have a `#define` statement for `INIT`. This definition is placed right after the declaration for the function `compile` and the opening brace ({). It is used for dependent declarations and initializations. Most often it is used to set a register variable to point to the beginning of the regular expression so that this register variable can be used in the declarations for `GETC( )`, `PEEKC( )`, and `UNGETC( )`. Otherwise, it can be used to declare external variables that might be used by `GETC( )`, `PEEKC( )`, and `UNGETC( )`. See the following example of the declarations taken from `grep`(1).

There are other functions in this file that perform actual regular expression matching, one of which is the function `step`.

Arguments to the `step` function are as follows:

Argument	Description
string	Pointer to a string of characters to be checked for a match. This string should be null terminated.
expbuf	Compiled regular expression that was obtained by a call of the function `compile`.

The `step` function returns 1, if the given string matches the regular expression, and 0 if the expressions do not match. If there is a match, two external character pointers are set as a side effect to the call to `step`. The variable set in `step` is *loc1*. This is a pointer to the first character that matched the regular expression.

The variable *loc2*, which is set by the function `advance`, points to the character after the last character that matches the regular expression. Thus, if the regular expression matches the entire line, *loc1* points to the first character of *string* and *loc2* points to the null at the end of *string*.

The `step` function uses the external variable *circf*, which is set by `compile` if the regular expression begins with `^`. If this is set, `step` only tries to match the regular expression to the beginning of the string. If more than one regular expression is to be compiled before the first is executed, the value of *circf* should be saved for each compiled expression and *circf* should be set to that saved value before each call to `step`.

The `advance` function is called from `step` with the same arguments as `step`. The purpose of `step` is to step through the *string* argument and call `advance` until `advance` returns 1 indicating a match or until the end of *string* is reached. If you want to constrain *string* to the beginning of the line in all cases, `step` need not be called, simply call `advance`.

When `advance` encounters an `*` or `\{ \}` sequence in the regular expression, it advances its pointer to the string to be matched as far as possible and recursively calls itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, `advance` backs up along the string until it finds a match or reaches the point in the string that initially matched the `*` or `\{ \}`. It is sometimes desirable to stop this backing up before the initial point in the string is reached. If the external character pointer *locs* is equal to the point in the string at sometime during the backing up process, `advance` breaks out of the loop that backs up and returns 0. This is used by `ed`(1) and `sed`(1) for substitutions done globally (not just the first occurrence, but the whole line); so, for example, expressions such as `s/y*//g` do not loop forever.

EXAMPLES

The following example shows how the regular expression macros and calls look from `grep`(1):

```
#define INIT        register char *sp = instring;
#define GETC( )     (*sp++)
#define PEEKC( )    (*sp)
#define UNGETC(c)   (--sp)
#define RETURN(c)   return;
#define ERROR(c)    regerr( )

#include <regexp.h>
 ...
          compile(*argv, expbuf, &expbuf[ESIZE], (int) '\0');
 ...
          if (step(linebuf, expbuf))
                   succeed( );
```

NOTES

Because `regexp.h` is a header file, no actual `libc` modules exist. This source code is provided for users wanting to use functions for regular expression work.

SEE ALSO

`ed`(1), `expr`(1), `grep`(1), `sed`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

remainder, remainderf, remainderl – Divides its arguments and returns the remainder

SYNOPSIS

```
#include <fp.h>

double remainder (double x, double y);
float remainderf (float x, float y);
long double remainderl (long double x, long double y);
```

IMPLEMENTATION

CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

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

DESCRIPTION

The remainder, remainderf, and remainderl functions compute the remainder *r* = *x* REM *y*. If *y* is not equal to 0, according to the IEEE floating-point standard, the remainder "is defined regardless of the rounding mode by the mathematical relation *r* = *x* - *y* * *n*, where *n* is the integer nearest the exact value of *x*/*y*; whenever |*n* - *x*/*y* | = *1/2*, then *n* is even. Thus, the remainder is always exact. If *r* = 0, its sign shall be that of *x*."

RETURN VALUES

These functions return the remainder of the first argument divided by the second argument.

SEE ALSO

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

NAME

remove – Removes files

SYNOPSIS

```
#include <stdio.h>
int remove (const char *file);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The remove function causes the file whose name is the string to which *file* points to become inaccessible by that name. A subsequent attempt to open the file by using that name fails, unless you create the file from scratch. If the file is open, the behavior of remove is implementation-defined. On Cray Research systems, the file remains accessible to the file descriptor (or stream).

If *file* does not specify a directory, remove(*file*) is equivalent to unlink(*file*). If *file* specifies a directory, remove(*file*) is equivalent to rmdir(*file*).

RETURN VALUES

If the operation succeeds, the remove function returns 0; if it fails, it returns a nonzero value.

SEE ALSO

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

NAME

`rename` – Renames a file

SYNOPSIS

```
#include <stdio.h>
int rename(const char *old, const char *new);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The `rename` function changes the name of a file. The *old* argument points to the path name of the file to be renamed. The *new* argument points to the new path name of the file.

If the *old* argument and the *new* argument both refer to links to the same existing file, the `rename` function returns successfully and performs no other action.

If the *old* argument points to the path name of a file that is not a directory, the *new* argument does not point to the path name of a directory. If the link specified by the *new* argument exists, it is removed and *old* is renamed *new*. In this case, a link named *new* exists throughout the renaming operation and refers either to the file referred to by *new* or *old* before the operation began. Write access permission is required for both the directory that contains *old* and the directory that contains *new*.

If the *old* argument points to the path name of a directory, the *new* argument points to the path name of a file that is a directory. If the directory specified by the *new* argument exists, it is removed and *old* renamed *new*. In this case, a link named *new* exists throughout the renaming operation and refers either to the file referred to by *new* or *old* before the operation began. If *new* specifies an existing directory, it must be an empty directory.

The *new* path name does not contain a path prefix that names *old*. Write access permission is required for the directory that contains *old* and the directory that contains *new*. If the *old* argument points to the path name of a directory, write access permission is required for the directory named by *old*, and, if it exists, the directory named by *new*.

If the link specified by the *new* argument exists and the file's link count becomes 0 when it is removed and no process has the file open, the space occupied by the file is freed and the file is no longer accessible. If one or more processes have the file open when the last link is removed, the link is removed before `rename` returns, but the removal of the file contents is postponed until all references to the file are closed.

NOTES

Under UNICOS, rename(2) is implemented as a system call, but the rename function also is defined to be a part of the ANSI Standard C library. For this reason, this documentation appears both here and in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012.

RETURN VALUES

The rename function returns 0 if the operation succeeds and marks for update the st_ctime and st_mtime fields of the parent directory of each file. rename returns a nonzero if it fails; in which case, if the file existed previously, it is still known by its original name.

SEE ALSO

remove(3C)

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

NAME

dn_comp, dn_expand, dn_skipname, hostalias, res_init, res_mkquery, res_query, res_querydomain, res_search, res_send – Provides domain name service resolver functions

SYNOPSIS

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

int res_init (void);

int res_mkquery (int op, char *name, int class, int type, char *data, int datalen,
u_char *newrr, u_char *buf, int buflen);

int res_query (char *name, int class, int type, u_char *answer, int anslen);

int res_search (char *name, int class, int type, u_char *answer, int anslen);

int res_querydomain (char *name, char *domain, int class, int type, u_char
*answer, int anslen);

char *hostalias (char *name);

int res_send (u_char *buf, int buflen, u_char *answer, int anslen);

int dn_expand (u_char *msg, u_char *eomorig, u_char *comp_dn, char *exp_dn, int
length);

int dn_comp (char *exp_dn, char *comp_dn, int length, u_char **dnptrs, u_char
**lastdnptr);

int dn_skipname (unsigned char *comp_dn, unsigned char *eom);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The res_init function initializes the resolver functions, consulting /etc/resolv.conf (if it exists) for configuration information. The res_send(), res_query(), and res_search() routines automatically call res_init() if it has not yet been called.

The `res_query` function creates a standard query for the fully qualified domain name pointed to by *name*, of class *class* and type *type*, sends the query to the configured name server, and awaits an answer. The answer is placed in the buffer, of length *anslen*, pointed to by *answer*.

The `res_search` function calls `res_query` and `res_querydomain` to perform a search for the domain name *name* among the domains specified in the resolver configuration.

The `res_mkquery` function creates a query in the buffer, of length *buflen*, pointed to by *buf*. The query is formulated with the following characteristics: *op* is the opcode of the query; *name* points to the domain name to be queried; *class* is the class of the query; *type* is the type of query; *data* points to *datalen* bytes of associated data for the query; *newrr* points to an existing resource record associated with the query. (Legal values for *op*, *class*, and *type* can be found in the include file `arpa/nameser.h`.) This routine is typically called only by `res_query()`.

The `res_querydomain` function calls `res_query` to perform a lookup of the concatenation of the domain names pointed to by *name* and *domain*, and returns the value returned by `res_query`. This routine is typically called only by `res_search()`.

The `hostalias` function consults the environment variable `HOSTALIASES` for the name of a file that contains a list of domain names and aliases, and returns a pointer to the first alias found in such a file for the domain name pointed to by *name*.

The `res_send` function sends the query pointed to by *buf*, of length *buflen*, to the configured server or servers. It retrieves the answer in the buffer, of length *anslen*, pointed to by *answer*.

The `dn_expand` function expands the compressed domain name pointed to by *comp_dn* into the buffer of length *length*, pointed to by *exp_dn*.

The `dn_comp` function compresses the domain name pointed to by *exp_dn* into the buffer of length *length* pointed to by *comp_dn*. *dnptrs* points to a null-terminated list of pointers to previous compressed names; *lastdnptr* points to the actual end of the array pointed to by *dnptrs*.

The `dn_skipname` function skips over a compressed domain name.

RETURN VALUES

The `res_init` function returns 0 on successful initialization, or –1 on error.

The `res_query` function returns the length of answer received, or –1 on error, in which case it sets the external variable `h_errno` to reflect the type of error.

The `res_mkquery` function returns the size of the resulting query, or –1 on error.

The `hostalias` function returns `NULL` if the environment variable `HOSTALIASES` does not exist, the file referred to by `HOSTALIASES` cannot be opened, or no alias is found for *name*.

The `res_send` function returns the length of the answer received, or –1 on error.

The dn_expand function returns the length of the compressed name, or –1 on error.

The dn_comp function returns the length of the compressed name, or –1 on error.

The dn_skipname functions returns the size of the compressed name skipped, or –1 on error.

FILES

arpa/nameser.h

/etc/resolv.conf

SEE ALSO

gethost(3C), herror(3C)

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

NAME

rexec – Returns a stream to a remote command

SYNOPSIS

```
#include <unistd.h>

int rexec (char **ahost, unsigned short inport, char *user, char *passwd, char
*cmd, int *fd2p);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The rexec function uses gethostbyname (see gethost(3C)) to look up the remote host *ahost*; rexec returns –1 if the remote host does not exist. Otherwise, *ahost* is set to the official name of the host. If a user name and password are both specified, these are used to determine whether authorization exists for the remote host; otherwise, the .netrc file in the user's home directory is searched for the appropriate information. If all searches fail, the user is prompted for a login name and password.

The port *inport* specifies which TCP port to use for the connection; it is normally the value returned from the function getservbyport (see getserv(3C)).

The protocol for the connection is described in detail in rexecd(8).

If the rexec function succeeds, a socket of type SOCK_STREAM is returned to the caller and given to the remote command *cmd* as the file descriptors stdin (for reading to the socket) and stdout (for writing to the socket). If *fd2p* is nonzero, an auxiliary channel to a control process is set up, and a descriptor for it is placed in *fd2p*. The control process returns diagnostic output from the command (stderr) on this channel and also accepts bytes on this channel as being UNICOS signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, stderr is made the same as stdout. In this case, no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

NOTES

There is no way to specify options to the socket(2) call that rexec makes.

FILES

`$HOME/.netrc`

SEE ALSO

`gethost`(3C), `getserv`(3C), `rcmd`(3C), `stdio.h`(3C)

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

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

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

NAME

`rint`, `rintf`, `rintl` – Rounds arguments to an integral value in floating-point format

SYNOPSIS

```
#include <fp.h>

double rint (double x);
float rintf (float x);
long double rintl (long double x);
```

IMPLEMENTATION

CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

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

DESCRIPTION

The `rint`, `rintf`, and `rintl` functions round their arguments to an integral value in floating-point format, using the current rounding direction.

RETURN VALUES

These functions return the rounded integral value of their arguments.

SEE ALSO

`rinttol`(3C)

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

NAME

`rinttol` – Rounds a floating-point number to a long integer value

SYNOPSIS

```
#include <fp.h>
long int rinttol (long double x);
```

IMPLEMENTATION

CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

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

DESCRIPTION

The `rinttol` function rounds its `long double` argument to `long int`, using the current rounding direction. If the rounded value is outside the range of `long int`, the numeric result is unspecified.

RETURN VALUES

The `rinttol` function returns the rounded `long int` value, using the current rounding direction.

SEE ALSO

`rint`(3C)

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

NAME

rpc – Makes a remote procedure call

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The remote procedure call (RPC) functions allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a data packet to the server. On receipt of the packet, the server calls a dispatch function to perform the requested service and sends back a reply. Finally, the procedure call returns to the client.

The RPC library functions follow:

Function	Description
authdes_create	Returns the RPC authentication handle with DES authentication
auth_destroy	Destroys the authentication information handle
authnone_create	Returns an RPC null authentication handle
authunix_create	Returns an RPC UNIX authentication handle
authunix_create_default	Returns the default UNIX authentication handle
callrpc	Calls a remote procedure, given [*prognum*,*versnum*,*procnum*]
clnt_broadcast	Broadcasts the remote procedure call
clnt_call	Calls the remote procedure associated with the client handle
clnt_create	Creates an RPC client when passed a remote host and transport type
clnt_destroy	Destroys the client's RPC handle
clnt_freeres	Frees data allocated by the RPC/XDR system when decoding results
clnt_geterr	Copies error information from the client handle to an error structure
clnt_pcreateerror	Prints a message to stderr that indicates why client handle creation failed
clnt_perrno	Prints a message that corresponds to the condition given to stderr
clnt_perror	Prints a message to stderr that indicates why the RPC call failed
clntraw_create	Creates a simple RPC client for simulation
clnttcp_create	Creates an RPC client by using TCP transport
clntudp_create	Creates an RPC client by using UDP transport
get_myaddress	Gets the machine's IP address
pmap_getmaps	Returns a list of RPC program-to-port mappings
pmap_getport	Returns the port number on which the supporting service waits
pmap_rmtcall	Instructs the portmapper to make an RPC call
pmap_set	Establishes mapping between [*prognum*,*versnum*,*procnum*] and *port*

`pmap_unset`	Destroys mapping between [*prognum*,*versnum*,*procnum*] and *port*
`registerrpc`	Registers a procedure with RPC as the service package
`svc_destroy`	Destroys the RPC service transport handle
`svc_freeargs`	Frees data allocated by the RPC/XDR system when decoding arguments
`svc_getargs`	Decodes the arguments of an RPC request
`svc_getcaller`	Gets the network address of the caller of a procedure
`svc_getreq`	Returns when all associated sockets have been serviced
`svc_getreqset`	Returns when all associated sockets have been serviced
`svc_register`	Associates *prognum* and *versnum* with a service dispatch procedure
`svc_run`	Waits for RPC requests to arrive and calls the appropriate service
`svc_sendreply`	Sends back results of a remote procedure call
`svc_unregister`	Removes mapping of [*prognum*,*versnum*] to dispatch functions
`svcerr_auth`	Refuses service because of an authentication error
`svcerr_decode`	Indicates that a service cannot decode its parameters
`svcerr_noproc`	Indicates that a service has not implemented the desired procedure
`svcerr_noprog`	Shows that a program is not registered with the RPC package
`svcerr_progvers`	Shows that a version is not registered with the RPC package
`svcerr_systemerr`	Indicates that a service detects a system error
`svcerr_weakauth`	Refuses service because of insufficient authentication
`svcraw_create`	Creates a simple RPC service transport for testing
`svctcp_create`	Creates an RPC service based on TCP transport
`svcudp_create`	Creates an RPC service based on UDP transport
`xdr_accepted_reply`	Generates RPC-style replies without using the RPC package
`xdr_authdes_cred`	Sends or receives DES credentials without using the RPC package
`xdr_authdes_verf`	Sends or receives DES verifier without using the RPC package
`xdr_authunix_parms`	Generates UNIX credentials without using the RPC package
`xdr_callhdr`	Generates RPC-style headers without using the RPC package
`xdr_callmsg`	Generates RPC-style messages without using the RPC package
`xdr_opaque_auth`	Describes RPC authenticators, externally
`xdr_pmap`	Describes parameters for portmap procedures, externally
`xdr_pmaplist`	Describes a list of port mappings, externally
`xdr_rejected_reply`	Generates RPC-style rejections without using the RPC package
`xdr_replymsg`	Generates RPC-style replies without using the RPC package
`xprt_register`	Registers RPC service transport with the RPC package
`xprt_unregister`	Unregisters RPC service transport from the RPC package

NOTES

Users access these library functions from `libc`.

FILES

`/lib/libc.a`	File that contains the RPC functions

SEE ALSO

`xdr`(3C)

Remote Procedure Call (RPC) Reference Manual, Cray Research publication SR–2089

"Remote Procedure Call Programming Guide," "Remote Procedure Call Protocol Specification," and "External Data Representation Specification" in Networking on the Sun Workstation, part #800–1324–03, Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94043.

RPC: Remote Procedure Call Version 2 Standard, RFC 1057

NAME

`rtclock` – Gets current real-time clock (RTC) reading

SYNOPSIS

```
#include <time.h>
long rtclock (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The `rtclock` function returns the current reading of the RTC.

SEE ALSO

`cpused`(3C)

NAME

SAMEQU – Specifies equivalent character in Sort/Merge session

SYNOPSIS

```
INTEGER init_array(2**N)
CALL SAMEQU(colseq, chr, init_array)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SAMEQU is used to make the characters specified in an array equivalent to one specific character in a specified collating sequence. The following is a list of valid arguments for this routine.

colseq Hollerith constant of 8 characters or less containing the name of the collating sequence.

chr A right-justified Hollerith character specifying the character that is equivalent to the characters in *init_array*.

init_array An integer array containing a set of characters that are considered equivalent to *chr* in the collating sequence *colseq* after the execution of SAMEQU. Each element of the array holds 1 Hollerith character that is right-justified and padded with zeros on the left. The maximum number of characters in *init_array* is 2^n, where *n* is the size in bits of a character. The default character size is 8. The list of characters in the array is terminated by an array element that contains the value –1.

After execution, the character values listed in *init_array* all compare equally to the specified character *chr* in the collating sequence *colseq*.

EXAMPLES

In the following example, the ASCII collating sequence is modified to make digits 1 through 9 equivalent to digit 0. After the execution of the call, digits 1 through 9 are treated in the same manner as digit 0 in the ASCII collating sequence by the Sort/Merge session.

```
INTEGER SAMPLE(10)
DATA SAMPLE/'1'R,'2'R,'3'R,'4'R,'5'R,'6'R,'7'R,'8'R,'9'R,-1/
CALL SAMEQU('ASCII'H,'0'R,SAMPLE)
```

SEE ALSO

SAMSIZE(3F)

NAME

SAMFILE – Defines subroutines for Sort/Merge operations

SYNOPSIS

CALL SAMFILE(*ftype*[[,*option*,*subname*] [,*option*,*subname*]])

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SAMFILE is used to specify subroutines to be used by the Sort/Merge routine. SAMFILE is used when the following options are specified:

ftype A Hollerith constant that is left-justified and padded with blanks, or an integer variable containing a Hollerith constant that specifies the file access type. Allowable values are:

- `'IN´H` Specifies an input subroutine; use the option ´READ=´ followed by a subroutine name.
- `'IN/M´H` Specifies an input subroutine that generates records that are already sorted; use the option `'READ='` followed by a subroutine name.
- `'OUT´H` Specifies an output subroutine; use the option ´WRITE=´ followed by a subroutine name.

option A Hollerith constant that is left-justified and filled with blanks, or an integer variable containing a Hollerith constant that specifies an action to be performed by a user-supplied subroutine for the Sort/Merge sessions. Depending on *ftype*, *option* can be one of the following:

ftype=`'IN'H` or `'IN/M'H`	***ftype***=`'OUT'H`
`'AFTER='H`	`'AFTER='H`
`'EOF='H`	`'=FIRST='H`
`'READ='H`	`'=EQUALS='H`
`'OPEN='H`	`'=LAST='H`
`'CLOSE='H`	`'EOF='H`
`'ERROR='H`	`'WRITE='H`
	`'OPEN='H`
	`'CLOSE='H`
	`'ERROR='H`

subname The name of the user-supplied subroutine to be used by the Sort/Merge session for the action specified by *option*.

User-supplied Subroutines

The user-supplied subroutines all have a common structure:

```
SUBROUTINE MYROUTINE (INBUF,INSZ,RETURN_CODE,OUTBUF,OUTSZ)
INTEGER INSZ,OUTSZ,RETURN_CODE
INTEGER INBUF(INSZ),OUTBUF(OUTSZ)
```

For an OPEN routine, parameters INBUF, INSZ, OUTBUF, and OUTSZ are dummies and must not be referenced by the subroutine. The RETURN_CODE is undefined on entry; it must be set before return to either ´RETURN´ or ´ABORT´. ´RETURN´ indicates successful initialization and ´ABORT´ indicates the opposite and terminates the sorting process.

For a READ routine, the user's program may fill INBUF(1) through INBUF(INSZ) with data for the input record. It should set INSZ to the number of words actually filled. RETURN_CODE may be set to ´ABORT´ in case of error; this terminates the sort process. If there is no more information, RETURN_CODE should be set to ´EOF´. If the routine successfully fills INBUF with information, RETURN_CODE should be set to ´INCLUDE´ (it is initialized by the Sort/Merge routine to this).

The CLOSE routine parameters INBUF, INSZ, OUTBUF, and OUTSZ are all dummies and must not be referenced by the user subroutine. The RETURN_CODE is undefined on entry; it must be set before return to either ´RETURN´ or ´ABORT´. ´RETURN´ indicates successful initialization and ´ABORT´ indicates the opposite and terminates the sorting process.

The WRITE routine words INBUF(1) through INBUF(INSZ) contain the current sorted output record. The write routine can do whatever it chooses with the contents of this record. The return code defaults to ´RETURN´, which allows the process to proceed. You may also specify ´ABORT´ in the event of error, which terminates the sort. If you specify ´EOF´, the Sort/Merge routine closes the current output sink and advances to the next sink, if any.

An error routine is called in a different manner:

```
SUBROUTINE MYERROR(NUMBER,MESSAGE)
INTEGER NUMBER
INTEGER MESSAGE(8)
```

You can supply values for the error number and the message; if a different error handler is needed for each file, you can embed knowledge about the source (file or internally generated) in the error handler. When the error handler returns, the sort terminates. The return codes are Hollerith constants that are left-justified and blank-filled.

EXAMPLES

In the following example, MYOPEN is called before the first read and MYCLOSE is called after the read routine declares end-of-file. MYERROR is called in the event of an error.

```
 EXTERNAL MYREAD, MYOPEN, MYCLOSE, MYERR
 CALL SAMFILE('IN'H,'READ='H,MYREAD,'OPEN='H,MYOPEN,
$             'CLOSE='H,MYCLOSE,'ERROR='H,MYERR)
```

In the following example, whenever Sort/Merge would write an output record it calls MYWRITE instead.

```
 EXTERNAL MYWRITE,MY_W_OPEN,MY_W_CLOSE,MY_W_ERR
 CALL SAMFILE('OUT'H,'WRITE='H,MY_WRITE,'OPEN='H,MY_W_OPEN,
$             'CLOSE='H,MY_W_CLOSE,'ERROR='H,MY_W_ERR)
```

SEE ALSO

SAMPATH(3F), SAMSORT(3F)

NAME

SAMGO – Initiate a Sort/Merge session

SYNOPSIS

```
CALL SAMGO
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The SAMGO routine starts a Sort/Merge session. This call must be last chronologically in a series of calls that start with either SAMSORT or SAMMERGE. There must be one or more intervening calls to SAMKEY, one or more calls to SAMFILE or SAMPATH specifying input sources, and one or more calls to SAMFILE or SAMPATH specifying output sinks. Optional calls can be made to SAMEQU, SAMOPT, SAMSEQ, or SAMSIZE.

SEE ALSO

SAMEQU(3F), SAMFILE(3F), SAMKEY(3F), SAMOPT(3F), SAMPATH(3F), SAMSEQ(3F), SAMSIZE(3F), SAMSORT(3F)

NAME

SAMKEY – Defines sort keys for a Sort/Merge session

SYNOPSIS

CALL SAMKEY(*type*[,*order*],arg_1, ... arg_n [,*colseq*])

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SAMKEY is one of a sequence of calls that specify the sort or merge operation to be performed. Unless you provide your own comparison routine (using the ´COMPARE=´ argument to SAMSORT or SAMMERGE) there must be a least one call to SAMKEY between a call to SAMSORT or SAMMERGE and the call to SAMGO, which initiates the Sort/Merge operation.

A call to SAMKEY must be preceded by a call to either SAMSORT or SAMMERGE.

There is no limit on the number of calls to SAMKEY.

The order in which you make the calls to SAMKEY determines the significance of the key in the sort. The first key specified has the most weight in comparisons.

The following is a list of valid arguments to this routine:

type A Hollerith constant or an integer variable containing a Hollerith constant specifying a Sort/Merge operation. *type* is a Hollerith constant that is left-justified, blank-filled and can contain the following values:

- ´BIN´H Sorts a field of arbitrary length as an unsigned integer. The field can be as small as 1 bit; there is no limit on length.
- ´INT´H Sorts a signed (2's complement) integer field. This is usually applied to a 64-bit Cray word. The maximum length is 64 bits; the field can be shorter. Such a field will usually start on a word boundary.
- ´FLT´H Sorts a floating-point number in Cray internal format. The maximum field length is 64 bits. The field can be shorter, but results will not be meaningful with a length less than 18 bits. The numbers should be normalized.
- ´CHR´H Sorts fields containing characters. The length of the string should be a multiple of the character size, which defaults to 8 bits (but which can be set with the SAMSIZE subroutine call). The default collating sequence is ASCII, but the collating options can be used or you can provide your own. There is no practical limit on the size of the field.

´DEC´H Compares a character field that contains decimal numbers. Results are meaningful only if the collating sequence is equivalent to ASCII or ASCIIUP sequences (an error results in other cases). The field must contain only decimal digits and at most one decimal point and at most one leading sign (positive or negative). Exponential notation (for example, 1.0E+2) cannot be used. Leading and trailing blanks are ignored. Embedded blanks are treated as zeros.

The *type* specified determines the file format used by the sorting routines. A *type* of 'BIN'H, 'INT'H, or 'FLT'H indicates UNFORMATTED input/output files. A *type* of 'CHR'H or 'DEC'H indicates FORMATTED input/output files.

order A Hollerith constant or an integer variable containing a Hollerith constant specifying the order in which records should be sorted. Allowable values are ´DESCEND´H or ´ASCEND´H (the default will be sorted in ascending field value). The value must be left-justified and filled with blanks.

arg Argument(s) specifying the starting location of the field. If there is no default length or if the default length is inappropriate, *arg* also specifies the length or ending position of the field. All the arguments associated with the starting location must precede any argument associated with the ending location or length. Arguments are provided in pairs (one of the key words followed by a decimal value). The key words are:

```
'START'H
'LENGTH'H
'END'H
'WORD='H
'CHR='H
'BIT='H
```

colseq An integer variable or Hollerith constant specifying the collating sequence to be used if the key type is ´CHR´. This may be a user-specified collating sequence, or it can be one of the following built-in collating sequences:

´ASCII´H	Sorts in ASCII sequence
´ASCIIUP´H	Sorts in ASCII sequence except lowercase letters are treated as uppercase letters
´EBCDIC´H	Sorts in ASCII sequence according to the EBCDIC sequence
´EBCDICUP´H	Sorts in ASCII sequence according to the EBCDIC sequence except that lowercase letters are treated as uppercase letters

Values for *arg*

Hollerith constants are used to specify information about location. The starting location of a field is tracked as a bit offset within the record. The first (high-order) bit is bit number 1. You can also specify a word offset or a character offset. The word offset is transformed into a bit offset at the rate of 64 bits per word (or the value specified in a SAMSIZE subroutine call). A character offset is transformed into a bit offset at a rate of 8 bits per character (or the size specified in a SAMSIZE call).

You can specify one, two, or all three of the parameters for specifying a bit offset. If you are doing a ´BIN´ comparison, you might want to start a field at the 5th bit of the 6th character of the 7th word. Any of the following sets of arguments would work:

```
,'WORD='H,7,'CHR='H, 6,'BIT='H,5,
,'WORD='H,6,'CHR='H,14,'BIT='H,5,
,'WORD='H,5,'CHR='H,22,'BIT='H,5,
,'WORD='H,4,'CHR='H,30,'BIT='H,5,
,'WORD='H,3,'CHR='H,38,'BIT='H,5,
,'WORD='H,2,'CHR='H,46,'BIT='H,5,
,'WORD='H,1,'CHR='H,54,'BIT='H,5,
            'CHR='H,53,'BIT='H,13,
                       'BIT='H,429,
```

If unspecified, the word, character, or bit parameter value is assumed to be 1. The word, character, and bit parameters values are transformed into a bit offset in the obvious way.

You must first specify the starting position. The *arg* parameter should be the value ´START´. It should be followed by 2, 4, or 6 parameters giving the actual starting position. Following this, you must specify either an ending bit position or a length. In either case, you combine the keywords in exactly the same way as for the starting bit position. Use the 'END'H or ´LENGTH'H keywords.

EXAMPLES

In the following example, the key is a character string that starts with the fourth character of the record and consists of 10 characters. The example does not specify a ranking order or collating sequence; therefore, the defaults apply.

```
CALL SAMKEY('CHR'H,'START'H,'CHR='H,4,'LENGTH'H,'CHR='H,10)
```

In the following example, the key is an integer that starts with the seventh bit position and is 12 bits. The keys are ranked in ascending order.

```
CALL SAMKEY('INT'H,'START'H,'BIT='H,7,'LENGTH'H,'BIT='H,12)
```

SEE ALSO

SAMGO(3F), SAMKEY(3F), SAMPATH(3F), SAMSORT(3F)

NAME

SAMOPT – Specifies sort options used in a Sort/Merge session

SYNOPSIS

CALL SAMOPT(*arg*[,*arg*])

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SAMOPT is used to specify the sort options available during a Sort/Merge session.

arg A Hollerith constant, or an integer variable containing a Hollerith constant, specifying sort options. *arg* is a Hollerith constant that is left-justified and padded with blanks. Specify one or both of the following values for *arg*:

- ´NOVERIFY´H Disables Sort/Merge verification of file order. The Sort/Merge routine will verify that files exist in the order in which they have been declared. Specify the order by starting the sort with the SAMMERGE subroutine which affects all input files, or by calling SAMFILE or SAMPATH with an *ftype* of ´IN/M´H for a specific file.
- ´RETAIN´H Maintains the original input order of a sequence of records with equivalent keys. If this option is not specified, the Sort/Merge routine does not maintain the original order.

SEE ALSO

SAMFILE(3F), SAMPATH(3F), SAMSORT(3F)

NAME

SAMPATH – Defines input and output files and characteristics for a Sort/Merge session

SYNOPSIS

CALL SAMPATH(*ftype*,*pathname*,[,*option*, *subname*]...)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SAMPATH specifies Sort/Merge input and output files to be used during a Sort/Merge session. User-supplied routines can be added at various stages of processing during the session.

The SAMFILE routine should be used when an output file is not required during a Sort/Merge session.

The following is a list of valid arguments to this routine:

ftype A Hollerith constant, or an integer variable containing a Hollerith constant that specifies a file access type. *ftype* is a Hollerith constant that must be left-justified and padded with blanks. Allowable values are:

- ´IN´H Specifies an input file
- ´IN/M´H Specifies an input file that generates records that are sorted
- ´OUT´H Specifies an output file

pathname An integer or Hollerith constant specifying the absolute or relative path name of the input or ouptut file to be used during a Sort/Merge session.

option An integer or Hollerith constant specifying when to use a subroutine supplied through *subname*. *option* is a Hollerith constant that must be left-justified and padded with blanks. The allowable values are:

- ´AFTER´H Specifies that a subroutine will be called after a record has been selected for either input or output
- ´=FIRST=´H, ´=EQUALS=´H, ´=LAST=´H Specifies that a subroutine be called to process the first member of a class of records with equal keys or to process the last member of the class, or to process the middle records

subname The name of a subroutine to be called as specified in *option*.

User-supplied Routines

The user may supply subroutines to be used by the Sort/Merge routine during a Sort/Merge operation. See the description of user-supplied subroutines in SAMFILE. The third parameter in a call to one of these subroutines may be a STATUS or an ACTION variable. The following Hollerith constants can be specified as STATUS and ACTION variables:

STATUS Used to process records in routines. STATUS can have the following values:

- ´INCLUDE´ Includes the new record during input processing. You must copy the contents of INREC(1:INWDS) to OUTREC and set OUTWDS to INWDS. You can modify the record as you copy it, and revise the length of the record.
- ´REPLACE´ Creates a new record during input processing. The new record replaces the input record. A replacement counter is incremented (this counter appears in the statistical summary if requested).
- ´INSERT´ Inserts a new record during input processing or end-of-file (EOF) processing. Store the new record in OUTREC and set OUTWDS appropriately. An insertion counter is incremented (this counter appears in the statistical summary if requested).
- ´OMIT´ Omits the record during input processing. The Sort/Merge package ignores the current record and retrieves the next record. An omission counter is incremented (this counter appears in the statisical summary if requested).
- ´EOF´ States that the current input file is finished. The Sort/Merge package proceeds as if an EOF had been detected. If one has been specified, the next input file is used.
- ´ABORT´ States that an error occurred during input processing or EOF processing. If you provide an error-processing routine, it is called and the Sort/Merge session terminates.
- ´RETURN´ Used only in EOF processing. The EOF condition is propagated back to the Sort/Merge package.

ACTION Used in output processing. The following values are available for the ACTION variable:

- ´INCLUDE´ Includes the record presented as INREC as the output record.
- ´OMIT´ Discards the current record. An omission counter is incremented.
- ´REPLACE´ Writes the record stored to OUTREC (the length of which is stored in OUTWDS). A replacement record counter is incremented. The record offered in INREC is discarded.
- ´INSERT´ Writes the record stored to OUTREC (the length of which is stored in OUTWDS). A replacement record counter is incremented. Sort/Merge immediately calls the routine presenting the same record.

´EOF´	Calls a user-supplied EOF routine (if available). If only one output file was specified, the sort terminates normally. If more than one output file was specified, the current file is closed.
´ABORT´	Terminates the sort/merge session.
´RETURN´	Used only in EOF processing. The EOF condition is propagated back to the sort/merge package.

Equivalence Class Processing

Equivalence class processing is similar to other output processing. The possible return codes are the same and operate in the same way. Any records that are inserted do not affect the determination of equivalence classes. If a user-supplied routine inserts a record, it will be recalled with the original record, even though it is not first. The Sort/Merge session will call the user-supplied subroutine with the original record until that record has been processed.

It is not necessary to write routines for all three equivalence class possibilities. Any user-supplied routine to handle the EOF condition will be called if the sort runs out of records or if another user-supplied routine has returned the 'EOF' for STATUS.

The only valid ACTION values are ´INSERT´ or ´RETURN´. Store the record to be inserted in OUTREC and set its length in OUTWDS. Sort/Merge writes the record and calls the EOF processing routine again. The inserted record is not considered in the equivalence processing. If a record is supplied in OUTREC, its length must be less than the maximum record length, which is 20 words (unless changed with the SAMTUNE call).

EXAMPLES

The following example gains control after each record is read:

```
      EXTERNAL MYAFTER
C     other parts of the program
      CALL SAMPATH('IN'H, 'the_data_file'H, 'AFTER='H, MYAFTER)
```

After the records are read from the_data_file, the Sort/Merge package calls the MYAFTER subroutine, which should have the following elements:

```
      SUBROUTINE MYAFTER(INREC,INWDS,STATUS,OUTREC,OUTWDS)
      INTEGER INWDS, STATUS, OUTWDS
      INTEGER INREC(INWDS), OUTRED(OUTWDS)
```

The size of the input and output records are in Cray 64-bit words. When reading character records, the record size is rounded to an integral number of Cray words. The output record is also an integral number of Cray words.

The following example gains control after an EOF condition has been detected on an input file:

```
      EXTERNAL MYEOF
C     other parts of the program
      CALL SAMPATH('IN'H,'the_data_file'H,'EOF='H, MYEOF)
```

The following example includes both an ´AFTER´ routine and an ´EOF´ routine:

```
      EXTERNAL MYAFTER, MYEOF
    C  other parts of the program
      CALL SAMPATH('IN'H,'the_data_file'H,'AFTER='H,MYAFTER,'EOF='H,MYEOF
```

When the EOF condition occurs in the file, the subroutine is called. It should have the following elements:

```
      SUBROUTINE MYEOF (INREC,INWDS,STATUS,OUTREC,OUTWDS)
      INTEGER INWDS, STATUS, OUTWDS
      INTEGER INREC(INWDS), OUTREC(OUTWDS)
```

The following example captures records before they are written during output:

```
      EXTERNAL OUTAFTER
     C  other program parts
      CALL SAMPATH('OUT'H,path_name,'AFTER='H,OUTAFTER)
```

Before the output record is written, the Sort/Merge package calls the subroutine, which should have the following elements:

```
      SUBROUTINE OUTAFTER(INREC,INWDS,ACTION,OUTREC,OUTWDS)
      INTEGER INWDS,ACTION,OUTWDS
      INTEGER INREC(INWDS), OUTREC(OUTWDS)
```

SEE ALSO

SAMSORT(3F), SAMFILE(3F)

NAME

SAMSEQ – Specifies and defines a collating sequence

SYNOPSIS

```
INTEGER init_array (2**N)
CALL SAMSEQ(colseq, init_array)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SAMSEQ is used to define a collating sequence used in Sort/Merge sessions.

The following is a valid list of arguments to this routine:

colseq A Hollerith constant or an integer variable containing a Hollerith constant specifying a collating sequence. *colseq* is a Hollerith constant with a maximum of 8 characters and is left-justified and filled with blanks. This can be specified in subsequent SAMKEY calls.

init_array An integer array containing the collating sequence. It can have a maximum size of 2**N, where N is the character size (by default 8 bits; this can be set using the SAMSIZE routine). Each element in the array holds 1 Hollerith character that is right-justified and padded with zeros on the left. The character used in the low-order bits of the first element of the array is used as the low value in the collating sequence. The value used for the second element becomes the next lowest value in the collating sequence, and so on. The value –1 terminates the array entries.

EXAMPLES

In the following example, a user-specified collating sequence contains the default character set in reverse order. A file is sorted in reverse order by an ASCII key; the ´DESCEND´H keyword is not used in the SAMKEY call. The following program fragment defines a collating sequence in which X´FF´ compares low with respect to other elements. X´42´ (the ´B´ character) compares low with respect to X´41´ (the ´A´ character). You can then use the ´IICSA´H collating sequence in a SAMKEY call, achieving a descending sort.

```
      INTEGER REVERSED(256)
      DO 100 I = 1,256
      REVERSED(I) = 256 - I
  100 CONTINUE
      CALL SAMSEQ('IICSA'H,REVERSED)
```

In the following example, the SAMPLE collating sequence will consider 999 to be smaller than 000. In addition, any unspecified characters compare larger than any specified character and are equal to each other. Therefore 99Z is larger than 999, but is equal to 99A. The nonstandard notation ´*´R indicates that the character value is to be right-justified in the word and that unused character positions are to be set to 0.

```
INTEGER SAMPLE(11)
DATA SAMPLE/'9'R, '8'R, '7'R, '6'R, '5'R, '4'R, '3'R, '2'R, '1'R, -1/
CALL SAMSEQ('SAMPLE'H,SAMPLE)
```

SEE ALSO

SAMKEY(3F), SAMSIZE(3F)

NAME

SAMSIZE – Specifies word and character sizes for Sort/Merge session

SYNOPSIS

```
CALL SAMSIZE(keyword, value)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SAMSIZE specifies a character size or word size for character comparisons.

keyword A Hollerith constant, or an integer variable containing a Hollerith constant, specifying either character or word size. *keyword* is a Hollerith constant, containing either ´CHR=´H or ´WORD=´H.

value An integer variable, expression, or constant specifying the number of keyword items.

To specify a character size for use in character comparisons, specify ´CHR=´H for *keyword*, followed by an integer between 1 and 12 for *value*.

To specify a word size for character comparison, specify ´WORD=´H for *keyword* and an integer for *value*. The default *value* is 64; there is no maximum. This value is used to interpret the ´WORD=´H keyword in the SAMKEY subroutines.

Both options can be specified in the same call.

SEE ALSO

SAMKEY(3F)

NAME

SAMSORT, SAMMERGE – Begins a Sort/Merge specification

SYNOPSIS

```
CALL SAMSORT(errflag [,´STAT´H] [,´STATF=´H,dsname] [,´ERRORF=´H,dsname]
[,´ERROR=´H,errsub] [,´COMPARE=´H,compsub])
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SAMSORT starts the specification of a Sort/Merge session. After this call, you must call SAMKEY one or more times, call either SAMFILE or SAMPATH one or more times, and call other subroutines as needed.

SAMSORT must not be called a second time until the sort you are specifying has been executed by a call to SAMGO. If no error is encountered by SAMSORT, *errflag* is set to 0. If an error is detected by SAMSORT, *errflag* is set to 1. If an error is detected by a user-supplied routine, *errflag* is set to 2.

The following is a list of valid arguments for this routine:

errflag	An integer variable that is a member of a common block.
´STAT´H	Used to generate statistics on the Sort/Merge operation. Use either this option or ´STATF´H.
´STATF´H	Used to generate statistics; use either this option or ´STAT´H.
dsname	An integer variable, expression, or constant containing the path name of the file (a total of 8 characters or less) to which statistics generated by ´STATF´H should be written or to which error messages should be saved. If more than 8 characters are needed for a path name, use the SAMPATH subroutine. The Hollerith constant is left-justified and blank-filled.
´ERRORF´H	Used to specify a file to receive error messages. If more than 8 characters are needed for a path name, use the SAMPATH subroutine instead.
´ERROR´H	Used to specify an error routine. This routine cannot assist in error recovery; it can, however, do final cleanup such as closing files, writing error messages, and so on.
errsub	The name of the error subroutine.
´COMPARE´H	Used to specify an optional user-supplied comparison routine. See the CAUTIONS section for information about including a user-supplied comparison routine.
compsub	The name of the comparison subroutine.

CAUTIONS

It is recommended that you do not supply your own comparison routine to be used during the sorting or merging process. This prevents the Sort/Merge routine from storing key information compactly and from using the ORDERS(3F) routine from the scientific libraries.

SEE ALSO

SAMEQU(3F), SAMFILE(3F), SAMGO(3F), SAMKEY(3F), SAMPATH(3F), SAMOPT(3F), SAMSEQ(3F), SAMSIZE(3F)

NAME

SAMTUNE – Modifies selected parameters used in a Sort/Merge session

SYNOPSIS

CALL SAMTUNE(*keyword, value, ...*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SAMTUNE is used to alter the parameters used in a Sort/Merge session.

Performance in a Sort/Merge session is determined by the amount of available central memory. Sort/Merge uses half of a megaword of memory by default; the majority of the memory is used for buffers for intermediate files used during the merge phase. During the first phase all of the memory is used to form runs of greatest length.

keyword A Hollerith constant or integer variable containing a Hollerith constant specifying tuning keywords. *keyword* is a Hollerith constant that is left-justified and blank-filled.

value An integer constant or variable, depending on the *keyword*.

Initial runs are formed using the ORDERS(3F) routine in libsci unless a user-supplied comparison routine is specified. The ORDERS routine is vectorized; the later merge stage sorts the records using a tournament sort method. The merge phase sorts are scalar.

The following are valid *keyword*s. Any keywords must be used as is, with equal signs and single quotation marks.

´AVRL=´H Specifies the average record length. The default is the maximum record length. This value is used to allocate internal tables. It is recommended that you use an average record length that is too small rather than one that is too large.

´MXRL=´H Specifies the maximum record length. The default is 20 Cray words (160 bytes). If the maximum record length is too short, the sort aborts during the input phase.

´NRECEST=´H An estimate of the number of records in the input files. The default is 1,000,000 records. This value is no longer used.

´DISK=´H No longer used in this implementation. Instead, you may specify a colon-separated list of directories in the CSORTDIR environment variable.

´DSLO=´H No longer used in this implementation.

´NAMEBM=´H No longer used in this implementation.

´NAMESSD=´H No longer used in this implementation.

´NDS=´H	Specifies the number of temporary datasets to be used during the merge phase. The default is 10; minimum is 4. Change this parameter only if you must run a Sort/Merge routine in minimum memory. A large value is recommended, but many temporary datasets are assigned smaller buffers and buffers should be large to maximize I/O efficiency. This makes it difficult to assign an efficient number for this keyword.
´NDSSD=´H	No longer used in this implementation.
´NBSSD=´H	No longer used in this implementation.
´NDBM=´H	No longer used in this implementation.
´NBBM=´H	No longer used in this implementation.
´NBDSK=´H	Specifies the number of sort buffers to be allocated to each temporary dataset. The size of the buffer is specified by the ´MNBL=´H and ´MXBL=´H parameters. The default value is 2.
´MNBL=´H	Specifies the number of word blocks in each sort buffer. The default is 42 (the track size of a DD-49). If you supply both a minimum and a maximum, the minimum must be the smaller of the two numbers.
´MXBL=´H	Specifies the number of word blocks in each sort buffer. The default is 42. If you supply both a minimum and a maximum, the minimum must be the smaller of the two numbers.

SEE ALSO

SAMSORT(3F)