NAME

scalb, scalbf, scalbl – Computes *x* * FLT_RADIX^n efficiently

SYNOPSIS

CRAY T90 systems with IEEE hardware:

```
#include <fp.h>

double scalb (double x, long int n);
float scalbf (float x, long int n);
long double scalbl (long double x, long int n);
```

Cray MPP systems:

```
#include <fp.h>

double scalb (double x, long int n);
```

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 scalb functions and macro compute *x* * FLT_RADIX^n efficiently, rather than by computing FLT_RADIX^n explicitly.

RETURN VALUES

All return *x* * FLT_RADIX^n .

The second parameter has type long int, unlike the corresponding int parameter for ldexp(3C), because the factor required to scale from the smallest positive floating-point value to the largest finite one, on many implementations, is too large to represent in the minimum-width int format allowed by Standard C.

SEE ALSO

float.h(3C) for information on the FLT_RADIX macro
frexp(3C) for information on ldexp

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

NAME

scandir, alphasort – Scans a directory

SYNOPSIS

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

int scandir (const char *dirname, struct dirent ***namelist,
      int (*select)(struct dirent *),
      int (*compar)(const void *, const void *));

int alphasort (const void *d1, const void *d2);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

Function scandir reads the directory *dirname* and builds an array of pointers to directory entries using malloc(3C). The *namelist* argument is a pointer to an array of structure pointers. The *select* argument is a pointer to a function that is called with a pointer to a directory entry and should return a nonzero value if the directory entry should be included in the array. If this pointer is null, all the directory entries will be included. The *compar* argument is a pointer to a function that is passed to qsort(3C) to sort the completed array. If this pointer is null, the array is not sorted.

Function alphasort sorts the array of pointers to directory entries alphabetically. It returns an integer that is greater than, equal to, or less than zero according to whether the string pointed to by the d_name field in the dirent structure pointed to by *d1* is greater than, equal to, or less than the string pointed to by the d_name field in the dirent structure pointed to by *d2*.

Function scandir returns the number of entries in the array and a pointer to the array through *namelist*.

NOTES

In some other operating systems, *namelist* is of type struct direct, not struct dirent. Therefore, when code is ported from other systems, all instances of struct direct must be changed to struct dirent.

RETURN VALUES

These functions return –1 if the directory cannot be opened for reading or if `malloc`(3C) cannot allocate enough memory to hold all the data structures.

SEE ALSO

`directory`(3C), `malloc`(3C), `qsort`(3C)

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

NAME

scanf, fscanf, sscanf – Converts formatted input

SYNOPSIS

```
#include <stdio.h>
int scanf (const char *format, ... );
int fscanf (FILE *stream, const char *format, ... );
int sscanf (const char *s, const char *format, ... );
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The scanf function reads from the standard input stream stdin, fscanf reads from the specified input *stream*, and sscanf reads from the character string *s*. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string *format* and a set of arguments that indicates where the converted input should be stored. If insufficient arguments for the format exist, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

You can apply conversions to the *nth* argument after the *format* in the argument list, rather than the next unused argument. In this case, the conversion character % is replaced with the sequence %*n*$; *n* is a decimal integer in the range 1 to NL_ARGMAX (defined in limits.h). This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages. In format strings that contain the %*n*$ from of conversion specifications, it is unspecified whether numbered arguments in the argument list can be referenced from the format string more than once.

The *format* can contain either form of a conversion specification, that is % or %*n*$, but usually the two forms cannot be mixed within a single *format* string. The only exception to this is that you can mix %% or %* with the %*n*$ form.

Argument *format* is a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: one or more white-space characters, an ordinary multibyte character (neither % nor a white-space character), or a conversion specification. Each conversion specification is introduced by the % symbol, or the character sequence %*n*$, after which the following appear in sequence:

1. An optional assignment-suppressing character ∗.
2. An optional decimal integer that specifies the maximum field width.

3. An optional `h`, `l` (ell), `ll` (ell ell), or `L` indicating the size of the receiving object. The conversion specifiers `d`, `i`, and `n` are preceded by `h` if the corresponding argument is a pointer to `short int` rather than a pointer to `int`, by `l` if it is a pointer to `long int`, or by `ll` if it is a pointer to `long long int`. Similarly, the conversion specifiers `o`, `u`, and `x` are preceded by `h` if the corresponding argument is a pointer to `unsigned short int` rather than a pointer to `unsigned int`, or by `l` if it is a pointer to `unsigned long int`. Finally, the conversion specifiers `e`, `f`, and `g` are preceded by `l` if the corresponding argument is a pointer to `double` rather than a pointer to `float`, or by `L` if it is a pointer to `long double`. If an `h`, `l`, or `L` appears with any other conversion specifier, the behavior is undefined.
4. A character that specifies the type of conversion to be applied. The valid conversion specifiers are as specified in this entry.

The `fscanf` function executes each directive of the format in turn. If a directive fails, as detailed in the following paragraphs, the `fscanf` function returns. Failures are described as input failures (due to the unavailability of input characters) or matching failures (due to inappropriate input).

A directive composed of white-space characters is executed by reading input up to the first nonwhite-space character (which remains unread), or until no more characters can be read.

A directive that is an ordinary multibyte character is executed by reading the next characters of the stream. If one of the characters differs from one comprising the directive, the directive fails, and the differing and subsequent characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described for each specifier. A conversion specification is executed in the following steps:

1. Input white-space characters (as specified by function `isspace`) are skipped, unless the specification includes a `[`, `c`, `C`, or `n` specifier. These white-space characters are not counted against a specified field width.
2. An *input item* is read from the stream, unless the specification includes an `n` specifier. An input item is defined as the longest matching sequence of input characters, unless that exceeds a specified field width, in which case, it is the initial subsequence of that length in the sequence. The first character, if any, after the input item remains unread. If the length of the input item is 0, the execution of the directive fails; this condition is a matching failure, unless an error prevented input from the stream, in which case, it is an input failure.
3. Except in the case of a `%` specifier, the input item (or, in the case of a `%n` directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails; this condition is a matching failure. Unless assignment suppression was indicated by a `*`, the result of the conversion is placed in the object to which the first argument points following the *format* argument that has not already received a conversion result if the conversion specification is introduced by %, or in the *n*th argument if introduced by the character sequence %*n*$. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The following conversion specifiers are valid:

Code	Description
d	Matches an optionally `signed` decimal integer, whose format is the same as expected for the subject sequence of function `strtol`, with the value 10 for the `base` argument. The corresponding argument is a pointer to integer.
i	Matches an optionally `signed` integer, whose format is the same as expected for the subject sequence of function `strtol` with the value 0 for the `base` argument. The corresponding argument is a pointer to integer.
o	Matches an optionally `signed` octal integer, whose format is the same as expected for the subject sequence of function `strtoul` with the value 8 for the `base` argument. The corresponding argument is a pointer to `unsigned` integer.
u	Matches an optionally `signed` decimal integer, whose format is the same as expected for the subject sequence of function `strtoul` with the value 10 for the `base` argument. The corresponding argument is a pointer to `unsigned` integer.
x	Matches an optionally `signed` hexadecimal integer, whose format is the same as expected for the subject sequence of function `strtoul` with the value 16 for the `base` argument. The corresponding argument is a pointer to `unsigned` integer.
e, f, g	Matches an optionally `signed` floating-point number, whose format is the same as expected for the subject string of the `strtod` function. The corresponding argument is a pointer to `float`.
s	Matches a sequence of nonwhite-space characters. The corresponding argument is a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which is added automatically.
[	Matches a nonempty sequence of characters from a set of expected characters (the scanset). The corresponding argument is a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which is added automatically. The conversion specifier includes all subsequent characters in the *format* string, up to and including the matching right bracket (]). The characters between the brackets (the scanlist) comprise the scanset, unless the character after the left bracket is a circumflex (^), in which case, the scanset contains all characters that do not appear in the scanlist, between the circumflex and the right bracket. If the conversion specifier begins with [ ] or [^], the right bracket character is in the scanlist and the next right bracket character is the matching right bracket that ends the specification; otherwise the first right bracket character is the one that ends the specification. If a - symbol is in the scanlist and is neither the first character, nor the second character in which the first character is a ^, nor the last character, the behavior is implementation-defined. On Cray Research systems, this character indicates a range of characters, starting with the character just before -, and ending with the character just after the -.

c Matches a sequence of characters of the number specified by the field width (1 if no field width is present in the directive). The corresponding argument is a pointer to the initial character of an array large enough to accept the sequence. No null character is added.

p Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the %p conversion of the fprintf(3C) function. The corresponding argument is a pointer to a pointer to void. The interpretation of the input item is implementation-defined. If the input item is a value converted earlier during the same program execution, the pointer that results must compare equal to that value; otherwise, the behavior of the %p conversion is undefined. On Cray Research systems, the conversion is the same as the o conversion.

n No input is consumed. The corresponding argument is a pointer to integer into which will be written the number of characters read from the input stream so far by this call to scanf. Execution of a %n directive does not increment the assignment count returned at the completion of scanf execution.

B Matches an optionally signed binary integer, whose format is the same as expected for the subject sequence of function strtoul with the value 2 for the base argument. The corresponding argument is a pointer to unsigned integer.

C Matches a sequence of characters of the number specified by the field width (1 if no field width is present in the directive). The sequence is converted to a sequence of wide-character codes in the same manner as the mbstowcs(3C) function. The corresponding argument must be a pointer to an array of type wchar_t large enough to accept the sequence that is the result of the conversion. No null wide-character code is added. In this case, the normal skip over white-space characters is suppressed.

S Matches a sequence of characters that are not white space. The sequence is converted to a sequence of wide-character codes in the same manner as the mbstowcs(3C) function. The corresponding argument must be a pointer to an array of type wchar_t large enough to accept the sequence and a terminating null wide-character code, which will be added automatically. If the field width is specified, it denoted the maximum number of characters to accept.

% Matches a single %; no conversion or assignment occurs. The complete conversion specification is %%.

If a conversion specification is not valid, the behavior is undefined.

The conversion specifiers E, G, and X are valid also and behave the same as, respectively, e, g, and x.

If end-of-file (EOF) is encountered during input, conversion is terminated. If EOF occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream. Trailing white space (including newline characters) is left unread, unless matched by a directive. The success of literal matches and suppressed assignments cannot be directly determined, other than by using the `%n` directive.

The `scanf` function is equivalent to `fscanf` with the argument `stdin` interposed before the arguments to `scanf`.

The `sscanf` function is equivalent to `fscanf`, except that the argument *s* specifies a string (rather than a stream) from which the input will be obtained. Reaching the end of the string is equivalent to encountering EOF for the `fscanf` function. If copying occurs between objects that overlap, the behavior is undefined.

NOTES

The success of literal matches and suppressed assignments cannot be determined directly.

RETURN VALUES

The `fscanf`, `scanf`, and `sscanf` functions return the value of the `EOF` macro if an input failure occurs before any conversion. Otherwise, the functions return the number of input items assigned, which can be fewer than provided for, or even 0, if an early matching failure occurs.

SEE ALSO

`getc`(3C), `mbstring`(3C), `printf`(3C), `strtol`(3C)

NAME

sdsalloc, sdsfree, sdsrealloc, sdsinfo, SDSALLOC, SDSREALC, SDSFREE, SDSINFO – Secondary data segment (SDS) management routines

SYNOPSIS

Calls from C:

```
#include <sdsalloc.h>
int sdsalloc (int size [,int *ierr]);
int sdsfree (int oblk [,int *ierr]);
int sdsrealloc (int oblk, int size [,int *ierr]);
int sdsinfo (struct sdsinfo *info);
```

Calls from Fortran:

```
INTEGER SDSALLOC, SDSREALC, SDSFREE
INTEGER NBLK, SIZE, OBLK, ISTAT, IERR, INFO(10), LEN
nblk = SDSALLOC (size [,ierr])
nblk = SDSREALC (oblk, size [,ierr])
istat = SDSFREE (oblk [,ierr])
CALL SDSINFO (istat, info, len)
```

IMPLEMENTATION

All Cray Research systems except CRAY T3E systems

DESCRIPTION

These routines manage the SDS space that can be associated with a process. The routines allocate and deallocate space from the system, using the ssbreak(2) system call, and assign that space to callers as appropriate. The total SDS space associated with the process containing allocated and unallocated regions is called the *arena*.

The following is a list of valid arguments for these routines:

Argument	Description
nblk	Type integer (output) Returns zero based block number of allocated SDS space.
size	Type integer (input) Size of SDS allocation request in 4096-byte blocks.
oblk	Type integer (input) Block number of currently allocated SDS space.

istat	Type integer (output) Return status (0 equals success, and –1 equals error)
ierr	Error code (optional parameter, output)
info	Array into which SDSINFO places the output data.
len	Number of words in *info* array.

SDSREALC and sdsrealloc reallocate the number of contiguous 4096-byte blocks requested by *size*. The current allocation starting at *oblk* is expanded if possible. Data is preserved from the old allocation to the new allocation if it must be moved.

The return value of sdsalloc, sdsrealloc, SDSALLOC, and SDSREALC is a zero-based block number of allocated SDS space.

SDSFREE and sdsfree return the allocated SDS space identified by *oblk* to the arena.

The smallest unit of allocation is one 4096-byte block. Fragmentation is reduced by using a lowest fit algorithm. When allocating or reallocating, the arena is searched in address order. This order ensures that the lowest SDS address that fits is always used.

SDSINFO and sdsinfo provide detailed information about the state of the SDS arena and allows some tuning of applications by using SDS. The sdsinfo structure is found in the header file sdsinfo.h, and contains members, as follows:

```
struct sdsinfo {
      int remain;     /* INFO(1) */
                  /* amount of SDS space available from the system */
                  /* that has not yet been requested and added to */
                  /* the arena */

      int curalloc;     /* INFO(2) */
                  /* Current arena size, in 4096-byte blocks. */

      int maxblk;       /* INFO(3) */
                  /* maximum size allocation request */

      int maxfree;    /* INFO(4) */
                  /* largest free block already in the arena */

      int maxalloc;     /* INFO(5) */
                  /* size of largest allocation in the arena */

      int totfree;      /* INFO(6) */
                  /* total free space in arena */

      int totalloc;   /* INFO(7) */
                  /* total allocated space */
```

```
        int numfree;      /* INFO(8) */
                          /* number of separate unallocated areas in the arena */

        int numalloc;     /* INFO(9) */
                          /* number of allocations in the arena */

        int pad[3];
                          /* not used. space reserved for future expansion */
 };
```

On the first SDS allocation request, these routines try to determine the maximum amount of SDS space allowed for the process. The SDS arena is then enlarged (using `ssbreak`(2)) to that size, and it is not shrunk until all space in the arena is freed. At that point, the entire SDS arena is released to the system (using `ssbreak`(2)). Under the Network Queuing System (NQS), the user-declared process and job SDS limits are used to determine this maximum arena size.

For users who want to modify this behavior, the following three environment variables are provided:

Variable	Description
`SDSLIMIT`	Maximum size of the arena. This variable does not override the system-imposed limits if those system-imposed limits are smaller.
`SDSINCR`	The preferred size `ssbreak`(2) request that should be tried to increase the size of the arena.
`SDSMAXFR`	The maximum amount of free space that is allowed to remain in the high addresses of the arena before the space is returned to the system by using `ssbreak`(2).

To implement the default behavior, set `SDSLIMIT`, `SDSINCR`, and `SDSMAXFR` to the system-imposed SDS process limit.

CAUTIONS

If these routines detect that the user has executed an `ssbreak`(2) directly, the space that is allocated is considered an allocation. This procedure only partially allows the mixing of `ssbreak`(2) and `SDSALLOC` requests, and it is not recommended. Use of the `ssbreak`(2) system call with a negative argument in conjunction with these routines produces unpredictable results. Because these routines are used in the system libraries to perform operations on auxiliary arrays, and in flexible file I/O (FFIO) processing, you should not assume that these routines are not being used.

On CRAY T3D systems, `sdsalloc` does not handle allocation of SDS space from more than one processing element (PE). You should not use these routines from more than one PE.

EXAMPLES

The following example illustrates the use of all of the SDS management routines.

```
      program example
      integer BLKS
      parameter (BLKS=10)
      integer dat(512*BLKS)
      integer  sdsalloc, sdsrealc, sdsfree, ssread, sswrite
      external sdsalloc, sdsrealc, sdsfree, ssread, sswrite, sdsinfo
      integer ssize, base, sdsaddr, iter, iret, info(10)
c
      base = sdsalloc(1) ! allocate dummy area
c
      do 100 ssize = 1,20
c
        base = sdsrealc(base, ssize * BLKS)
        if (base .lt.0) then
          print *,'realloc failed on iteration ',ssize
          stop 'realc'
        endif
c
        do 10 iter = 0,SSIZE-1
          print *,'write ',iter
          do 15 j = 1,512 * BLKS
            dat(j) = iter*512*BLKS + j
   15     continue
          sdsaddr = base + iter*BLKS
          iret = SSWRITE(dat, sdsaddr, 512*BLKS) ! words, not blocks, to write
          if (iret .ne. 0) print *,'Oops, return is ', iret,
     +                             ' on iteration ',iter
   10   continue
c
c       Use SDSINFO to inquire about the state of the SDS arena
c
      length = 9 ! get 9 words
      call SDSINFO(istat, info, length)
      if (istat.ne.0) stop 'SDSINFO'
      print *,'Current size of arena is ',info(2)
      print *,'Number of allocations is ',info(9)
c
c Read the data back
c
        do 20 iter = 0,SSIZE-1
          print *,'read ',iter
          sdsaddr = base + iter*BLKS
          iret = SSREAD(dat, sdsaddr, 512*BLKS) ! words, not blocks, to read
          if (iret .ne. 0) print *,'Oops, return is ', iret,
```

```
     +                              ' on iteration ',iter
          do 25 j = 1,512*BLKS
            if (dat(j) .ne. iter*512*BLKS + j) stop 'BAD'
 25       continue
 20     continue
100  continue
     iret =  sdsfree(base)
     if (iret .ne. 0) print *,'sdsfree failed!'
     end
```

A sample of the output from this program follows:

```
write 0
Current size of arena is 2048
Number of allocations is 1
read 0
write 0
write 1
Current size of arena is 2048
Number of allocations is 1
read 0
read 1
write 0
write 1
write 2
Current size of arena is 2048
Number of allocations is 1
read 0
read 1
read 2
write 0
write 1
write 2
write 3
Current size of arena is 2048
Number of allocations is 1
read 0
read 1
.
.
.
write 17
write 18
write 19
Current size of arena is 2048
```

```
Number of allocations is 1
.
.
.
```

SEE ALSO

`malloc`(3C)

`assign`(1) in the *Application Programmer's I/O Guide*, Cray Research publication SG–2168

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

Application Programmer's I/O Guide, Cray Research publication SG–2168, for information on FFIO

NAME

secnames, secbits, secnums, secwords – Converts security classification bit patterns or numbers to strings and vice versa

SYNOPSIS

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/secparm.h>
#include <sys/sectab.h>

int secnames(long bits, char *names[ ], int flag);

long secbits(char *namelist, int flag);

int secnums(char *name, int flag);

int secwords(int num, char *name[ ], int flag);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The secnames and secbits functions convert security compartment, category, security flag, and permission names from bit patterns to ASCII strings, and vice versa.

The secnums and secwords functions convert security level and integrity class names from numbers to ASCII strings, and vice versa. The use of integrity classes is not supported.

For secnames, the caller supplies a permission, category, flag, or compartment bit mask in *bits*. The caller must also supply a string array, *names*, into which the pointers to the ASCII names of the security compartments, categories, flag, or permissions are stored. The array is terminated with a null pointer.

For secbits, the caller supplies a pointer to a string of security compartment, category, flag, or permission names, separated by commas and null terminated. The function returns a bit mask representing all valid security compartment, category, flag, or permission bit positions represented by the names in the ASCII string.

Argument *flag* causes secnames to convert the bit mask and secbits to convert the comma-separated list of compartment names, respectively, according to the following values:

Flag value	Description
0 or 1	Returns compartments
2	Returns permissions

3 or 4	Returns categories
5	Returns flags

For `secnums`, the caller supplies a pointer to a security level or integrity class name string. The function returns a numeric value that corresponds to the supplied name. The use of integrity classes is not supported.

For `secwords`, the caller supplies a numeric value. The caller must also supply a string array, *name*, into which a pointer to the ASCII name of that value is stored.

Argument *flag* causes `secnums` and `secwords` to return a numeric value or name, respectively, for a given security level or class name, according to the following values:

Flag value	**Description**
0, 1, or 2	Returns security level
3, 4, or 5	Returns integrity class. The use of integrity classes is not supported.

RETURN VALUES

Upon successful completion, `secbits` and `secnums` return values as previously described. If unsuccessful, `secbits` and `secnums` return –1.

For `secnames` and `secwords`, a 0 is returned if the function completes successfully; otherwise, a –1 is returned.

SEE ALSO

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

General UNICOS System Administration, Cray Research publication SG–2301

NAME

`security` – Introduction to security functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

These functions operate the security features.

ASSOCIATED FUNCTIONS

Function	Description
`ia_failure`	Processes identification and authentication (I&A) failures
`ia_mlsuser`	Determines the user's mandatory access control (MAC) attributes
`ia_success`	Processes identification and authentication (I&A) successes
`ia_user`	Performs user identification and authentication (I&A)
`mldlist`	Obtains the list of mandatory access control (MAC) labels currently represented in a multilevel directory
`mldname`	Expands a multilevel symbolic link reference at an arbitrary mandatory access control (MAC) label
`mldwalk`	Walks the labeled subdirectories of a multilevel directory (MLD)
`mls_create`	Creates an opaque security label structure
`mls_dominate`	Performs a security label domination test
`mls_equal`	Performs a security label equality test
`mls_export`	Converts internal security label to text representation
`mls_extract`	Extracts label from an opaque security label structure
`mls_free`	Frees security label storage space
`mls_glb`	Computes the greatest lower bound
`mls_import`	Converts text security label to internal representation
`mls_lub`	Computes the least upper bound
`priv_clear_file`	Clears all privilege sets in a file privilege state
`priv_clear_proc`	Clears all privilege sets in a process privilege state
`priv_dup_file`	Creates a copy of a file privilege state
`priv_dup_proc`	Creates a copy of a process privilege state
`priv_free_file`	Deallocates file privilege state space
`priv_free_proc`	Deallocates process privilege state space
`priv_get_fd`	Gets the privilege state of a file
`priv_get_file`	Gets the privilege state of a file
`priv_get_file_flag`	Indicates the existence of a privilege in a file privilege set
`priv_get_proc`	Gets the privilege state of the calling process
`priv_get_proc_flag`	Indicates the existence of a privilege in a process privilege state
`slgtrust`, `slgtrustobj`	Writes trusted process security log record

SEE ALSO

General UNICOS System Administration, Cray Research publication SG–2301

NAME

setbuf, setvbuf – Assigns buffering to a stream

SYNOPSIS

```
#include <stdio.h>
void setbuf (FILE *stream, char *buf);
int setvbuf (FILE *stream, char *buf, int mode, size_t size);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

You can use the setvbuf function after a stream has been opened but before any other operation is performed on the stream. The *mode* argument determines how *stream* is buffered. Legal values for *mode* (defined in the header file stdio.h) are as follows:

Type	Description
_IOFBF	Input/output is fully buffered.
_IOLBF	Output is line-buffered; the buffer is flushed when a newline character is written, the buffer is full, or input is requested.
_IONBF	Input/output is completely unbuffered.

If the stream is unbuffered, *buf* and *size* are ignored.

If *buf* is not a null pointer, the array to which it points is used for buffering. Argument *size* specifies the size (in bytes) of the array to be used. The standard I/O functions do not use all of the size bytes as an I/O buffer; some bytes are currently required for use as a pad, beyond the buffer's end and before its beginning. If *buf* is a null pointer, a buffer of length *size* plus the bytes required for padding is allocated automatically by setbuf and later deallocated by close processing. Therefore, for optimal I/O performance, let *buf* be a null pointer and choose size =*n* * *sector_size* for some integer *n* (*sector_size* is the number of bytes in a disk sector).

The contents of the buffer at any time are indeterminate.

Except that it returns no value, the setbuf function called with a nonnull *buf* argument is equivalent to the setvbuf function invoked with the arguments _IOFBF for mode, a nonnull pointer *buf*, and BUFSIZ for size. The setbuf function called with a null *buf* argument is equivalent to the setvbuf function invoked with the argument _IONBF for *mode*.

NOTES

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

If *buf* is a null pointer, computation of an appropriate buffer size is easier. In that case, the library automatically allocates a buffer with the necessary pad space added to the end.

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

RETURN VALUES

If you provide an illegal value for *mode* or *size*, or if the request cannot be honored, `setvbuf` returns a nonzero value; otherwise, it returns 0 on success. The `setbuf` function returns no value.

SEE ALSO

`fopen`(3C), `getc`(3C), `malloc`(3C), `putc`(3C)

`dsk`(4) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014, for your system

NAME

setenv, unsetenv – Sets or removes the value of an environment variable

SYNOPSIS

```
#include <stdlib.h>
int setenv (char *name, char *value, int rewrite);
void unsetenv (char *name);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The setenv function sets the value of the environment variable *name* to *value*. If argument *rewrite* is 0, the new value is put into the environment list only if *name* is not currently in the environment list. If argument *rewrite* is nonzero, and if *name* already exists in the environment list, its value is replaced by *value*.

The unsetenv function removes all references to *name* from the environment list.

WARNINGS

On Cray MPP systems, each processing element (PE) gets a separate copy of the environment; therefore, alterations to the environment using setenv or unsetenv on one PE are not reflected on other PEs.

RETURN VALUES

If the value is placed into the environment, or if *rewrite* is 0 and *name* is already in the environment, the setenv function returns 0; otherwise, –1 is returned, and the new value is not placed into the environment list.

SEE ALSO

getenv(3C), putenv(3C)

NAME

set_jmp – Introduction to nonlocal jump functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The nonlocal jump functions provide a mechanism for bypassing the normal function call and return when an unusual condition is encountered in a program. These functions provide a capability similar to the signal-handling functions and may be used instead of or in conjunction with them.

ASSOCIATED HEADERS

<setjmp.h>

ASSOCIATED FUNCTIONS

Function	Description
setjmp(3C)	Saves stack environment before nonlocal goto
longjmp	Restores stack environment after nonlocal goto (see setjmp)
sigsetjmp	Saves stack environment before nonlocal goto and saves signal mask (see setjmp)
siglongjmp	Restores stack environment after nonlocal goto and restores signal mask (see setjmp)

SEE ALSO

sig_han(3C)

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

NAME

setjmp, longjmp, sigsetjmp, siglongjmp – Executes nonlocal goto

SYNOPSIS

```
#include <setjmp.h>
int setjmp (jmp_buf env);
void longjmp (jmp_buf env, int val);
int sigsetjmp (sigjmp_buf env, int savemask);
void siglongjmp (sigjmp_buf env, int val);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (functions setjmp and longjmp)
POSIX (functions sigsetjmp and siglongjmp)

DESCRIPTION

The setjmp macro and the longjmp function are useful for dealing with errors and interrupts encountered in a low-level function of a program.

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

An invocation of the setjmp macro can appear only in one of the following contexts:

- The entire controlling expression of a selection or iteration statement (if, for, while, do, switch)
- One operand of a relational or equality operator with the other operand an integral constant expression, and with the resulting expression being the entire controlling expression of a selection or iteration statement
- The operand of a unary ! operator with the resulting expression being the entire controlling expression of a selection or iteration statement
- The entire expression of an expression statement (possibly cast to void)
- The sole expression as the right operand of the '=' operator, with the left operand being a simple variable (available only in extended mode)

The longjmp function restores the environment saved by the most recent invocation of the setjmp macro, with the corresponding jmp_buf argument. If there has been no such invocation, or if the function containing the invocation of the setjmp macro has terminated execution in the interim, the behavior is undefined.

All accessible objects have values as of the time longjmp was called, except that the values of objects of automatic storage duration that are local to the function containing the invocation of the corresponding setjmp macro that do not have volatile-qualified type, and have been changed between the setjmp invocation and longjmp call, are indeterminate.

As it bypasses the usual function call and return mechanisms, the longjmp function executes correctly in contexts of interrupts, signals, and any of their associated functions. However, if the longjmp function is invoked from a nested signal handler (that is, from a function invoked as a result of a signal raised during the handling of another signal), the behavior is undefined.

Macro sigsetjmp works in the same manner as setjmp, but saves the signal mask. If the value of the *savemask* argument is not 0, sigsetjmp saves the process's current signal mask as a part of the calling environment.

Function siglongjmp works in the same manner as longjmp, but restores the signal mask if and only if the *env* argument was initialized by a call to sigsetjmp with a nonzero *savemask* argument.

NOTES

setjmp and sigsetjmp are macros. If the macro definition is suppressed in order to access an actual function, or if a program defines an external identifier with the name setjmp or sigsetjmp, the behavior is undefined.

Use of these functions and macros may invalidate the results of using Flowtrace, Perftrace, or Watchword.

The space for variable length arrays declared at block scope (that is, not parameters), and the space obtained using calls to the malloc function can be lost if their storage is active across a longjmp call.

RETURN VALUES

The setjmp and siglongjmp macros return the value 0 if the return is from a direct invocation; they return a nonzero value if the return is from a call to the longjmp or siglongjmp function.

After longjmp or siglongjmp is completed, program execution continues as if the corresponding invocation of the setjmp or sigsetjmp macro had just returned the value specified by *val*. The longjmp or siglongjmp functions cannot cause the setjmp or sigsetjmp macros to return the value 0; if *val* is 0, it is changed to the value 1.

SEE ALSO

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

flowtrace(7), perftrace(7)

NAME

`setjmp.h` – Library header for nonlocal jump functions

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

TYPES

The type defined in `setjmp.h` is as follows:

Type	Standards	Description
`jmp_buf`	ISO/ANSI	An array type suitable for holding the information needed to restore a calling environment.
`sigjmp_buf`	POSIX	An array type suitable for holding the information needed to restore a calling environment and signal mask.

FUNCTION DECLARATIONS

Functions declared in header `setjmp.h` are as follows:

`longjmp` `setjmp` `sigsetjmp†` `siglongjmp†`

† Available only in extended mode

NAME

`setlocale` – Selects program's locale

SYNOPSIS

```
#include <locale.h>
char *setlocale (int category, const char *locale);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI
POSIX

DESCRIPTION

The `setlocale` function can be used to change or query the program's entire current locale or portions thereof. The `setlocale` function selects the appropriate portion of the program's locale as specified by the arguments *category* and *locale*.

The following values can be used for *category*:

Value	Description
`LC_ALL`	Names the program's entire locale.
`LC_COLLATE`	Affects the behavior of the `strcoll`(3C) and `strxfrm`(3C) functions.
`LC_CTYPE`	Affects the behavior of the character-handling functions and the multibyte functions.
`LC_MONETARY`	Affects the behavior of the `strfmon`(3C) function.
`LC_NUMERIC`	Affects the decimal-point character for the formatted input/output functions and the string conversion functions, as well as the nonmonetary formatting information returned by the `localeconv`(3C) function.
`LC_TIME`	Affects the behavior of the `strftime`(3C) and `strptime`(3C) functions.
`LC_MESSAGES`	This currently does not affect the behavior of any functions but is intended for use in program messages.

Some environment variables correspond to the preceding *category* values and have the same spellings.

The *locale* argument is a pointer to a character string that can be an explicit string, a null pointer, or a null string.

If *locale* is an explicit string, a value of "`C`" for *locale* specifies the minimal environment for C translation; the value "`POSIX`" is a synonym for "`C`".

If *locale* is a null pointer, the locale of the process is queried according to the value of *category* and a string is returned that describes the current locale, which can be used on a subsequent call to `setlocale`.

If *locale* is a null string (""), the `setlocale` function takes the name of the new locale for the specified category from the environment as determined by the first condition met below:

1. If `LC_ALL` is defined in the environment and is not null, its value is used.
2. If there is a variable defined in the environment with the same name as the *category* value and it is not null, its value is used.
3. If the `LANG` environment variable is defined and it is not null, its value is used.

If the resulting value is a supported locale, `setlocale` sets the specified category of the locale of the process to that value and returns the value as specified in the RETURN VALUES section. If the value does not name a supported locale, `setlocale` returns a null pointer, and the locale of the process is unchanged. If no nonnull environment variable is present to supply a value, `setlocale` sets the specified *category* of the locale to the systemwide default value of "`C`".

At program startup, the equivalent of the following is executed:

```
setlocale(LC_ALL, "C");
```

RETURN VALUES

If a pointer to a string is given for *locale*, and the selection can be honored, the `setlocale` function returns a pointer to the string associated with the specified *category* for the new locale. If the selection cannot be honored, the `setlocale` function returns a null pointer, and the program's locale is not changed.

The pointer to string returned by the `setlocale` function is such that a subsequent call with that string value and its associated category restores that part of the program's locale. The string pointed to cannot be modified by the program, but may be overwritten by a subsequent call to `setlocale`.

SEE ALSO

`locale`(3C), `locale.h`(3C), `localeconv`(3C), `strftime`(3C), `string`(3C)

NAME

shmalloc, shfree, shrealloc, shmalloc_nb, shfree_nb, shrealloc_nb, shmalloc_check, shmalloc_stats – Shared heap memory management functions

SYNOPSIS

```
#include <malloc.h>
void *shmalloc(size_t size);
void shfree(void *ptr);
void *shrealloc(void *ptr, size_t size);
void *shmalloc_nb(size_t size);
void shfree_nb(void *ptr);
void *shrealloc_nb(void *ptr, size_t size);
int shmalloc_check(int level);
void shmalloc_stats(int level);
extern long malloc_error;
```

IMPLEMENTATION

Cray MPP systems

STANDARDS

CRI extension

DESCRIPTION

The shmalloc function returns a pointer to a block of at least *size* bytes suitably aligned for any use. This space is allocated from the shared heap (in contrast to malloc(3C), which allocates from the private heap), and the same address is returned on all programming elements (PE). The space returned is left uninitialized.

The shfree function causes the block to which *ptr* points to be deallocated, that is, made available for further allocation. If *ptr* is a null pointer, no action occurs; otherwise, if the argument does not match a pointer earlier returned by a shared heap function, or if the space has already been deallocated, malloc_error is set to indicate the error, and shfree returns.

The shrealloc function changes the size of the block to which *ptr* points to the size (in bytes) specified by *size*. The contents of the block are unchanged up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion of the block is indeterminate. If *ptr* is a null pointer, the shrealloc function behaves like the shmalloc function for the specified size. If *size* is 0 and *ptr* is not a null pointer, the block to which it points is freed. Otherwise, if *ptr* does not match a pointer earlier returned by a shared heap function, or if the space has already been deallocated, the malloc_error variable is set to indicate the error, and shrealloc returns a null pointer. If the space cannot be allocated, the block to which *ptr* points is unchanged.

The shmalloc, shfree, and shrealloc functions are provided so that multiple PEs in an application can allocate memory blocks with the same address on all PEs; these memory blocks can then be used with the shared memory (shmem) library. Each of these functions call the barrier(3C) function before returning; this ensures that all PEs participate in the memory allocation, and that the memory on other PEs can be used as soon as the local PE returns. The user is responsible for calling these functions with identical argument(s) on all PEs; if differing *size* arguments are used, subsequent calls may not return the same shared heap address on all PEs.

The shmalloc_nb, shfree_nb, shrealloc_nb functions work the same as shmalloc, shfree, and shrealloc, respectively, except that they contain no call to the barrier(3C) function. These functions can be used in applications where not all PEs participate in shared memory allocation; their use is discouraged for most programs. Note that calls to shmalloc, shfree, and shrealloc should not be attempted after calling shmalloc_nb, shfree_nb, or shrealloc_nb, as the shared heap may become inconsistent between the PEs participating in the shared memory allocation, and those not participating.

The shmalloc_check function checks the consistency of shmalloc's memory structure. If *level* is less than 0, shmalloc_check silently performs validation of the shared heap, and returns 0 if the heap is consistent, or nonzero if the heap has been corrupted. If *level* equals 0, shmalloc_check prints a message to stderr that describes the first inconsistency found. If *level* is greater than 0, shmalloc_check prints a line to stderr that describes each shared heap block in addition to checking the shared heap.

The shmalloc_stats function prints out memory manager statistics and heap block information to stdout. If *level* equals 0, shmalloc_stats reports the number of calls to each shared heap function, as well as summary statistics on the number and total size of the busy blocks, free blocks, and "spec" blocks (that is, blocks that are created by user calls to shsbreak) in the shared heap. If *level* equals 1, shmalloc_stats prints a line with a * for each busy block, a . for each free block, and a @ for each "spec" block, in addition to the level 0 statistics. If *level* equals 2, shmalloc_stats prints a line that describes each shared heap block, in addition to the level 0 statistics. The number of calls for each function are available only by linking with the libmalloc library; all of the other information is available in the default memory manager.

CAUTIONS

The shmalloc, shfree, and shrealloc functions differ from the private heap allocation functions in that all PEs in a partition must call them (a barrier is used to ensure this). The shmalloc_nb, shfree_nb, and shrealloc_nb functions do not use a barrier, and can be used by a subset of all PEs. None of these functions should be called within a master region; if this is done, they print out an error message and abort. The shmalloc_check and shmalloc_stats functions do not have any of these limitations.

RETURN VALUES

The shmalloc and shmalloc_nb functions return a pointer to the allocated space (which should be identical on all PEs); otherwise, they return a null pointer (with malloc_error set).

The shfree, shfree_nb, and shmalloc_stats functions return no value.

The shrealloc and shrealloc_nb functions return a pointer to the allocated space (which may have moved); otherwise, they return a null pointer (with malloc_error set).

If the shared heap has been corrupted, the shmalloc_check function returns nonzero; otherwise, it returns 0.

SEE ALSO

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

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

NAME

SHMALLOC, SHFREE, SHLOC – Shared pointer intrinsics

SYNOPSIS

```
INTRINSIC SHMALLOC, SHFREE, SHLOC
CALL SHMALLOC(pointer, alloc_stat [,length])
CALL SHFREE(pointer)
CALL SHLOC(pointer, array)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The SHMALLOC, SHFREE, and SHLOC intrinsics are the only allowable operations on shared pointers, whose pointee arrays are declared in a SHARED directive (see EXAMPLES section). A shared pointer can, during execution, point to different arrays, but all of the arrays must have identical distributions and extents.

The SHMALLOC intrinsic allocates shared heap memory. SHMALLOC gets the size of the space it allocates from the size of the pointee array associated with *pointer*, unless you include a *length* argument. When the pointee and the allocated array are the same size, passing two arguments (*pointer* and *alloc_stat*) to SHMALLOC is sufficient. If the allocated size differs from the pointee array's size, you must use the *length* (in words) argument. The *alloc_stat* argument indicates whether shared memory was successfully allocated; 0 indicates success, and nonzero indicates an error. Error conditions are identical to those of HPALLOC(3F).

The SHFREE intrinsic frees a shared memory block previously allocated by using SHMALLOC.

To maintain shared heap consistency, all processing elements (PEs) in a program must call SHMALLOC or SHFREE; otherwise, the program hangs.

The SHLOC intrinsic assigns to *pointer* the address of the shared *array*. You must use either SHLOC or SHMALLOC to associate a pointer with a shared array before using the pointer. Its functionality is like that of LOC on other systems, but SHLOC operates on shared data.

EXAMPLES

Example 1: The following example shows using SHLOC to associate a shared pointer with a shared array:

```
      POINTER (Q, X)
      REAL X(128, 128)
      REAL Y(128, 128)
      INTRINSIC SHLOC
CDIR$ SHARED X(:BLOCK, :BLOCK)
CDIR$ SHARED Y(:BLOCK, :)

      CALL SHLOC(Q, X)       ! associate pointer Q with array X
      CALL SHLOC(Q, Y)       ! error: distribution mismatch
```

Example 2: The following example shows using SHMALLOC to allocate shared memory of the same size as the pointee array:

```
      POINTER (P, PA)
      REAL PA(1024)
      INTRINSIC SHMALLOC
CDIR$ SHARED PA(:BLOCK(2))

      CALL SHMALLOC(P, ISTAT) ! allocate shared memory
```

Example 3: The following example shows using SHMALLOC to allocate shared memory of a different size from the pointee array, and using SHFREE to free the memory:

```
      POINTER (R, W)
      REAL W(128, 10000000)
      INTRINSIC SHMALLOC, SHFREE
CDIR$ SHARED W(:BLOCK, :)

      CALL SHMALLOC(R, ISTAT, 128*100)  ! only part of array allocated
      IF (ISTAT .EQ. 0) THEN
        CALL FRED(R)
        CALL SHFREE(R)                  ! free memory
      ENDIF
```

SEE ALSO

hpalloc(3F), shpalloc(3F), shpdeallc(3F), shpclmove(3F)

NAME

SHPALLOC – Allocates a block of memory from the shared heap

SYNOPSIS

```
CALL SHPALLOC(addr, length, errcode, abort)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

SHPALLOC allocates a block of memory from the program's shared heap that is greater than or equal to the size requested. If the request cannot be satisfied from the free blocks currently in the heap, it will try to allocate more memory from the system. To maintain shared heap consistency, all PEs in an program must call SHPALLOC with the same value of *length*; if any processing elements (PEs) are missing, the program will hang.

The SHPALLOC function accepts the following arguments:

Argument	Description
addr	First word address of the allocated block (output).
length	Number of words of memory requested (input).
errcode	Error code is 0 if no error was detected; otherwise, it is a negative integer code for the type of error (output).
abort	Abort code; nonzero requests abort on error; 0 requests an error code (input).

By using the Fortran POINTER mechanism in the following manner, you can use array A to refer to the block allocated by SHPALLOC:

```
POINTER (addr, A(1))
```

RETURN VALUES

Error conditions are as follows:

Error Code	Condition
-1	Length is not an integer greater than 0.
-2	No more memory is available from the system (checked if the request cannot be satisfied from the available blocks on the shared heap).

SEE ALSO

hpalloc(3F), shmalloc(3F), shpclmove(3F), shpdeallc(3F)

NAME

SHPCLMOVE – Extends a shared heap block or copies the contents of the block into a larger block

SYNOPSIS

CALL SHPCLMOVE (*addr*, *length*, *status*, *abort*)

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The SHPCLMOVE function either extends a shared heap block if the block is followed by a large enough free block or copies the contents of the existing block to a larger block and returns a status code indicating that the block was moved. This function also can reduce the size of a block if the new length is less than the old length. All processing elements (PEs) in a program must call SHPCLMOVE with the same value of *addr* to maintain shared heap consistency; if any PEs are missing, the program hangs.

The SHPCLMOVE function accepts the following arguments:

Argument	Description
addr	On entry, first word address of the block to change; on exit, the new address of the block if it was moved.
length	Requested new total length (input).
status	Status is 0 if the block was extended in place, 1 if it was moved, and a negative integer for the type of error detected (output).
abort	Abort code. Nonzero requests abort on error; 0 requests an error code (input).

RETURN VALUES

Error conditions are as follows:

Code	Condition
-1	Length is not an integer greater than 0.
-2	No more memory is available from the system (checked if the block cannot be extended and the free space list does not include a large enough block).
-3	Address is outside the bounds of the shared heap.
-4	Block is already free.
-5	Address is not at the beginning of a block.

SEE ALSO

hpalloc(3F), shmalloc(3F), shpalloc(3F), shpdeallc(3F)

NAME

SHPDEALLC – Returns a shared memory block of memory to the shared heap

SYNOPSIS

```
CALL SHPDEALLC(addr, errcode, abort)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

SHPDEALLC returns a block of memory (allocated using SHPALLOC) to the list of available space in the shared heap. To maintain shared heap consistency, all processing elements (PEs) in a program must call SHPDEALLC with the same value of *addr*; if any PEs are missing, the program hangs.

The SHPDEALLC function accepts the following arguments:

Argument	Description
addr	First word address of the block to deallocate (input).
errcode	Error code is 0 if no error was detected; otherwise, it is a negative integer code for the type of error (output).
abort	Abort code. Nonzero requests abort on error; 0 requests an error code (input).

Error conditions are as follows:

Code	Condition
-3	Address is outside the bounds of the shared heap.
-4	Block is already free.
-5	Address is not at the beginning of the block.

SEE ALSO

hpalloc(3F), shmalloc(3F), shpalloc(3F), shpclmove(3F)

NAME

shutdsav – Sets up calling program to be checkpointed on system shutdown

SYNOPSIS

```
#include <stdlib.h>
int shutdsav (char *path, int flags);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

CRI extension

DESCRIPTION

The shutdsav function establishes a signal handler that catches the SIGSHUTDN signal (defined in signal.h(3C)), indicating that the system will shut down soon. When the signal handler receives the SIGSHUTDN signal, it checkpoints the program by using the chkpnt(2) system call, creating a restart file with the specified name.

The shutdsav function accepts the following arguments:

Argument	Description
path	Path name of the file to be created as the restart file. The *path* argument must not refer to a file that already exists, because the chkpnt(2) system call, which performs the actual checkpoint work, will not overwrite an existing file. If *path* does not designate an absolute path name, the restart file is created relative to the current working directory of the calling program at the time the SIGSHUTDN signal is received. Finally, for the checkpoint to be successful, the caller must be able to write to the directory in which the restart file will be created.
flags	(Control flags) If the least significant bit of *flags* is 0, the calling program does not checkpoint itself on a system shutdown if it is running as part of an NQS batch job. This is important because, by default, NQS checkpoints all of the jobs under its control when a system shutdown occurs, making any additional checkpoint work by the program unnecessary. Alternatively, if the least significant bit of *flags* is set, the calling program tries to checkpoint itself, even when running as part of an NQS batch job. In all other cases, the calling program tries to checkpoint itself when it receives a SIGSHUTDN signal. Finally, all bits other than the least significant bit of *flags* are reserved for future use, and should be set to 0.

NOTES

For a process to be checkpointed successfully, certain conditions must be satisfied. For a discussion of the restrictions placed upon a process that is to be checkpointed, see chkpnt(2).

SEE ALSO

signal.h(3C)

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

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

SHUTDSAV(3F) in the , for details of the Fortran interface.

NAME

sig_han – Introduction to signal-handling functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The signal handling functions provide various means for handling signals, which are events that may occur during program execution and be reported to the executing program. The signals may occur because of hardware error detection, because of external events, or because of events generated by the program. When the signal occurs, the action taken depends on what action, if any, the program has specified for that signal.

Because computer systems and system environments vary greatly, the set of signals that may occur is system dependent. The set that applies to a CRI system environment is described in the header <signal.h>.

ASSOCIATED HEADERS

<signal.h>

ASSOCIATED FUNCTIONS

gsignal – Raises a software signal (see ssignal)
killpg – Sends signal to process group
raise – Sends a signal to an executing program
shutdsav – Sets up calling program to be checkpointed on system shutdown
sigaddset – Adds a signal to a signal set (see sigsetops)
sigdelset – Deletes a signal from a signal set (see sigsetops)
sigemptyset – Initializes a signal set to exclude all POSIX signals (see sigsetops)
sigfillset – Initializes a signal set that includes all POSIX signals (see sigsetops)
sigismember – Tests a signal to see if it is a member of a specified set (see sigsetops)
signal – Handles signals
sigoff – Allows signal catching to be postponed without a system call
sigon – Reenables signal catching after a sigoff call (see sigoff)
ssignal – Associates a function with a software signal

SEE ALSO

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

NAME

`signal` – Handles signals

SYNOPSIS

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The `signal` function chooses one of three ways to handle the receipt of signal number *sig*.

If the value of *func* is `SIG_DFL`, default handling for that signal occurs. If the value of *func* is `SIG_IGN`, the signal is ignored; otherwise, *func* points to a function to be called when that signal occurs. Such a function is called a *signal handler*.

When a signal occurs, if *func* points to a function, first the equivalent of the following is executed (except if *sig* is `SIGILL`, `SIGTRAP`, or `SIGPWR`):

```
signal(sig, SIG_DFL);
```

Next, the equivalent of the following is executed:

```
(*func)(sig);
```

The *func* function may terminate by executing a `return` statement or by calling the `abort`, `exit`, or `longjmp` function. The program resumes execution at the point at which it was interrupted.

If the signal occurs other than as the result of calling the `abort` or `raise` function, the behavior is undefined if the signal handler calls any function in the standard library other than the `signal` function itself (with a first argument of the signal number corresponding to the signal that caused the invocation of the handler) or refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type `volatile sig_atomic_t`. Furthermore, if such a call to `signal` results in a `SIG_ERR` return, the value of `errno` is indeterminate.

NOTES

Under UNICOS, `signal`(2) is implemented as a system call, but the `signal` function is defined also to be a part of the 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

If the request can be honored, the `signal` function returns the value of *func* for the most recent call to `signal` for the specified signal *sig*. Otherwise, a value of `SIG_ERR` is returned, and a positive value is stored in `errno`.

SEE ALSO

`abort`(3C), `exit`(3C), `setjmp`(3C), `signal.h`(3C), `sigon`(3C)

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

`kill`(2), `pause`(2), `ptrace`(2), `signal`(2), `wait`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

`signal.h` – Library header for signal-handling functions

IMPLEMENTATION

All Cray Research systems

TYPES

The following type is defined in the standard header `signal.h`:

Type	Standards	Description
`sig_atomic_t`	ISO/ANSI	The integral type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts.

MACROS

The following macros are defined as *signals* in the standard header `signal.h`:

Signal	Standards	Description
`SIGABRT`, `SIGIOT`, `SIGHWE`	ISO/ANSI	Expands to a positive integral constant expression that is the signal number corresponding to abnormal termination, such as is initiated by the `abort` function. (`SIGHWE` is used only on CRAY Y-MP systems.)
`SIGALRM`	POSIX	Expands to a positive integral constant expression that is the signal number corresponding to an alarm clock.
`SIGBUFIO`	CRI	Reserved for CRI-library usage on Cray MPP systems.
`SIGCLD`, `SIGCHLD`	POSIX	Expands to a positive integral constant expression that is the signal number corresponding to the death of a child process.
`SIGCONT`	POSIX	Expands to a positive integral constant expression that is the signal number corresponding to continuing a stopped process.
`SIGCPULIM`	CRI	Expands to a positive integral constant expression that is the signal number corresponding to CPU limit exceeded.
`SIGDLK`	CRI	Expands to a positive integral constant expression that is the signal number corresponding to a true deadlock detected.
`SIGERR`, `SIGEMT`	CRI	Expands to a positive integral constant expression that is the signal number corresponding to an error exit.

Signal	Standards	Description
SIGFPE	ISO/ANSI	Expands to a positive integral constant expression that is the signal number corresponding to an erroneous arithmetic operation, such as zero divide, an operation resulting in overflow, or a floating-point exception.
SIGHUP	POSIX	Expands to a positive integral constant expression that is the signal number corresponding to a hangup.
SIGILL	ISO/ANSI	Expands to a positive integral constant expression that is the signal number corresponding to detection of an invalid function image, such as an illegal instruction (not reset when caught).
SIGINFO	CRI	Expands to a positive integral constant expression that is the signal number corresponding to a quota warning or limit reached.
SIGINT	ISO/ANSI	Expands to a positive integral constant expression that is the signal number corresponding to receipt of an interactive attention signal; for example, an interrupt.
SIGIO	BSD	Expands to a positive integral constant expression that is the signal number corresponding to an input/output possible signal.
SIGKILL	POSIX	Expands to a positive integral constant expression that is the signal number corresponding to a kill (cannot be caught or ignored).
SIGMTKILL	CRI	Reserved for use by the Cray multitasking library.
SIGMT	CRI	Reserved for use by the Cray multitasking library.
SIGORE, SIGSEGV	CRI	Expands to a positive integral constant expression that is the signal number corresponding to an operand range error.
SIGPIPE	POSIX	Expands to a positive integral constant expression that is the signal number corresponding to a write on a pipe and no one to read it.
SIGWRBKPT	CRI	Expands to a positive integral constant expression that is the signal number corresponding to a breakpoint on the CRAY C90 series.
SIGPRE, SIGBUS	CRI	Expands to a positive integral constant expression that is the signal number corresponding to a program range error.
SIGPWR	AT&T	Expands to a positive integral constant expression that is the signal number corresponding to a power failure.
SIGQUIT	POSIX	Expands to a positive integral constant expression that is the signal number corresponding to quit (ASCII FS).

Signal	Standards	Description
SIGRECOVERY	CRI	Expands to a positive integral constant expression that is the signal number corresponding to a recovery signal.
SIGRPE	CRI	Expands to a positive integral constant expression that is the signal number corresponding to a register parity on CRAY Y-MP systems.
SIGSEGV	ISO/ANSI	Expands to a positive integral constant expression that is the signal number corresponding to invalid access to storage. (Also known as SIGORE.)
SIGSHUTDN	CRI	Expands to a positive integral constant expression that is the signal number corresponding to system shutdown imminent (advisory).
SIGSTOP	POSIX	Expands to a positive integral constant expression that is the signal number corresponding to sendable stop signal, not from tty.
SIGSYS	AT&T	Expands to a positive integral constant expression that is the signal number corresponding to a bad argument to system call.
SIGTERM	ISO/ANSI	Expands to a positive integral constant expression that is the signal number corresponding to a software termination request (from kill) sent to the program.
SIGTRAP	AT&T	Expands to a positive integral constant expression that is the signal number corresponding to a trace trap (not reset when caught).
SIGTSTP	POSIX	Expands to a positive integral constant expression that is the signal number corresponding to stop signal from tty.
SIGTTIN	POSIX	Expands to a positive integral constant expression that is the signal number corresponding to a reader's process group upon background tty read.
SIGTTOU	POSIX	Expands to a positive integral constant expression that is the signal number corresponding to a writer's process group upon background tty write.
SIGURG	BSD	Expands to a positive integral constant expression that is the signal number corresponding to an urgent condition on an I/O channel.
SIGUME	CRI	Expands to a positive integral constant expression that is the signal number corresponding to an uncorrectable memory error.
SIGUSR1	POSIX	Expands to a positive integral constant expression that is the signal number corresponding to user-defined signal 1.
SIGUSR2	POSIX	Expands to a positive integral constant expression that is the signal number corresponding to user-defined signal 2.

Signal	Standards	Description
SIGWINCH	AT&T	Expands to a positive integral constant expression that is the signal number corresponding to window size changes.

The following macros are defined as *signal actions* in the standard header `signal.h`:

Macro	Standards	Description
SIG_DFL	ISO/ANSI	Expands to a constant expression with a distinct value that is type compatible with the second argument to and the return value of the `signal` function, and whose value compares unequal to the address of any declarable function.
SIG_ERR	ISO/ANSI	Expands to a constant expression with a distinct value that is type compatible with the second argument to and the return value of the `signal` function, and whose value compares unequal to the address of any declarable function.
SIG_HOLD	AT&T	Expands to a constant expression with a distinct value that is type compatible with the second argument to and the return value of the `sigset` function, and whose value compares unequal to the address of any declarable function.
SIG_IGN	ISO/ANSI	Expands to a constant expression with a distinct value that is type compatible with the second argument to and the return value of the `signal` function, and whose value compares unequal to the address of any declarable function.

FUNCTION DECLARATIONS

`_lwp_kill`(2)†	`raise`(3C)	`sighold`(2)†	`sigprocmask`(2)†
`_lwp_killm`(2)†	`sigaction`(2)†	`sigignore`(2)†	`sigrelse`(2)†
`bsdsignal`(2)†	`sigaddset`(3C)†	`sigismember`(3C)†	`sigset`(2)†
`bsdsigpause`(2)†	`sigblock`(2)†	`signal`(3C)	`sigsetmask`(2)†
`gsignal`(3C)†	`sigctl`(2)†	`sigoff`†	`sigvec`(2)†
`kill`(2)†	`sigdelset`(3C)†	`sigon`(3C)†	`ssignal`†
`killm`(2)†	`sigemptyset`(3C)†	`sigpause`(2)†	
`killpg`(3C)†	`sigfillset`(3C)†	`sigpending`(2)†	

† Available only in extended mode.

NOTES

When compiling in extended mode, the header <sys/types.h> is included in <signal.h>. Thus, all of the types, macros, etc. defined in <sys/types.h> are available in addition to those mentioned above. Refer to sys/types.h> for information.

NAME

`signbit` – Determines if the sign of its argument is negative

SYNOPSIS

```
#include <fp.h>
int signbit (floating-type x);
```

IMPLEMENTATION

Cray MPP systems
CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

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

DESCRIPTION

The `signbit` macro determines if the argument value (including infinity, zero, or NaN) is negative. On Cray MPP systems, *floating-type* is a parameter of type `double`. On CRAY T90 systems with IEEE hardware, *floating-type x* is a parameter of any floating type. If the argument is not a floating type, the behavior is undefined.

If the `signbit` macro definition is suppressed in order to access an actual function, or if a program defines an external identifier with the name of this macro, the behavior is undefined.

RETURN VALUES

The macro returns a nonzero value if the sign of its argument value is negative.

SEE ALSO

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

NAME

sigoff, sigon – Controls signal-catching status

SYNOPSIS

```
#include <signal.h>
int sigoff(void);
int sigon(void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The sigoff function allows signal catching to be postponed without issuing a system call. However, signals that are not currently registered to be caught are not affected; that is, they would still kill the process or be ignored. Signals are not queued, so only one instance of each signal type is remembered while sigoff is in effect.

The sigon function reenables signal catching. Any signals that were postponed are delivered immediately.

Both sigoff and sigon return the previous status; a nonzero return value indicates that signal catching had been postponed.

When executing a signal handler, the initial signal-catching status differs depending upon which function was used to register for the signal (see sigctl(2), signal(2), or sigaction(2) for further information). Both sigoff and sigon can be issued inside a signal handler to change status.

SEE ALSO

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

NAME

sigemptyset, sigfillset, sigaddset, sigdelset, sigismember – Manipulates signal sets

SYNOPSIS

```
#include <signal.h>
int sigemptyset (sigset_t *set);
int sigfillset (sigset_t *set);
int sigaddset (sigset_t *set, int signo);
int sigdelset (sigset_t *set, int signo);
int sigismember (const sigset_t *set, int signo);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX

DESCRIPTION

These functions manipulate sets of signals. They operate on data objects addressable by the application, not on any set of signals known to the system, such as the set blocked from delivery by a process or the set pending for a process.

Function `sigemptyset` initializes the signal set pointed to by argument *set*, such that all signals defined in the Posix standard are excluded.

Function `sigfillset` initializes the signal set pointed to by argument *set*, such that all signals defined in the Posix standard are included.

Applications must call either `sigemptyset` or `sigfillset` at least once for each object of type `sigset_t` prior to any other use of that object. If such an object is not initialized in this way, but is supplied as an argument to functions `sigaddset`, `sigdelset`, `sigismember`, `sigprocmask`, `sigpending`, or `sigsuspend`, the results are undefined.

Functions `sigaddset` and `sigdelset` respectively add and delete the individual signal specified by the argument *signo* from the signal set pointed to by the argument *set*.

Function `sigismember` tests whether the signal specified by the value of the argument *signo* is a member of the signal set pointed to by the argument *set*.

RETURN VALUES

Upon successful completion, function `sigismember` returns a value of 1 if the specified signal is a member of the specified set; otherwise, a value of zero is returned. Upon successful completion, the other functions return a value of zero. For all of these functions, if an error is detected, a value of –1 is returned and `errno` is set to indicate the error. Values for `errno` can be the following:

`EFAULT` The set argument points outside the allocated address space.

`EINVAL` Invalid *signo*.

SEE ALSO

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

NAME

`sigwait` – Synchronous signal handling

SYNOPSIS

```
#include <signal.h>
int sigwait (const sigset_t *set, int *sig);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

PThreads

DESCRIPTION

The `sigwait` function waits for the reception of any of the signals in the specified *set* and returns that signal number to the caller.

The signals defined by *set* should be blocked at the time of the call to `sigwait`; otherwise `sigwait` may not behave in the manner described here. During the execution of `sigwait`, the registration for the signals in *set* is changed. The original registration is restored when `sigwait` returns, but the execution of a signal handler for a signal not in *set* may be affected by this if it or any function called by that handler is sensitive to the registration of the signals in *set*.

RETURN VALUES

Upon successful completion, `sigwait` stores the signal number of the received signal at the location referenced by *sig* and returns 0. Otherwise, an error number is returned to indicate the error.

MESSAGES

The `sigwait` function fails if one of the following error conditions occurs:

Error Code	Description
`EINVAL`	The *set* argument contains an invalid or unsupported signal number.

NAME

sin, sinf, sinl, csin, cos, cosf, cosl, ccos, tan, tanf, tanl – Determines the sine, cosine, or tangent of a value

SYNOPSIS

```
#include <math.h>
#include <complex.h> (for functions csin and ccos only)
double sin (double x);
float sinf (float x);
long double sinl (long double x);
double complex csin (double complex x);
double cos (double x);
float cosf (float x);
long double cosl (long double x);
double complex ccos (double complex x);
double tan (double x);
float tanf (float x);
long double tanl (long double x);
```

IMPLEMENTATION

All Cray Research systems (sin, csin, cos, ccos, tan only)
Cray MPP systems (sinf, cosf, tanf only)
Cray PVP systems (sinl, cosl, tanl only)

STANDARDS

ISO/ANSI (sin, cos, tan only)
CRI extension (all others)

DESCRIPTION

Functions sin, sinf, sinl, and csin return, respectively, the sine of a double, float, long double, or double complex value *x* in radians.

Functions cos, cosf, cosl, and ccos return, respectively, the cosine of a double, float, long double, or double complex value *x* in radians.

Functions `tan`, `tanf`, and `tanl` return, respectively, the tangent of a `double`, `float`, or `long double` value *x* in radians.

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

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:

- `sin(`*NaN*`)` returns NaN, and `errno` is set to `EDOM`.
- `sinl(`*NaN*`)` returns NaN, and `errno` is set to `EDOM`.
- `sin(`+/-*Inf*`)` returns NaN, and `errno` is set to `EDOM`.
- `sinl(`+/-*Inf*`)` returns NaN, and `errno` is set to `EDOM`.
- `cos(`*NaN*`)` returns NaN, and `errno` is set to `EDOM`.
- `cosl(`*NaN*`)` returns NaN, and `errno` is set to `EDOM`.
- `cos(`+/-*Inf*`)` returns NaN, and `errno` is set to `EDOM`.
- `cosl(`+/-*Inf*`)` returns NaN, and `errno` is set to `EDOM`.
- `tan(`*NaN*`)` returns NaN, and `errno` is set to `EDOM`.
- `tanl(`*NaN*`)` returns NaN, and `errno` is set to `EDOM`.
- `csin(`*x*+*NaN***1.0i*`)` returns NaN+Nan*1.0*i*, and `errno` is set to `EDOM`.
- `csinl(`*x*+*y*`*1.0`*i*`)`, where *x* is NaN or +/- infinity, returns NaN+Nan*1.0*i*, and `errno` is set to `EDOM`.
- `ccos(`*x*+*NaN***1.0i*`)` returns NaN+Nan*1.0*i*, and `errno` is set to `EDOM`.
- `ccos(`*x*+*y*`*1.0`*i*`)`, where *x* is NaN or +/- infinity, returns NaN+Nan*1.0*i*, and `errno` is set to `EDOM`.

SEE ALSO

`COS`(3M), `SIN`(3M), `TAN`(3M) in the *Intrinsic Procedures Reference Manual*, Cray Research publication SR–2138

NAME

sinh, sinhf, sinhl, cosh, coshf, coshl, tanh, tanhf, tanhl – Determines hyperbolic sine, cosine, or tangent of value

SYNOPSIS

```
#include <math.h>
double sinh (double x);
float sinhf (float x);
long double sinhl (long double x);
double cosh (double x);
float coshf (float x);
long double coshl (long double x);
double tanh (double x);
float tanhf (float x);
long double tanhl (long double x);
```

IMPLEMENTATION

All Cray Research systems (sinh, cosh, tanh only)
Cray MPP systems (sinhf, coshf, tanhf only)
Cray PVP systems (sinhl, coshl, tanhl only)

STANDARDS

ISO/ANSI (sinh, cosh, tanh only)
CRI extension (all others)

DESCRIPTION

Functions sinh, sinhf, and sinhl return, respectively, the hyperbolic sine of a double, float, or long double value, *x*. A range error occurs if they are called with an argument that would cause overflow.

Functions cosh, coshf, and coshl return, respectively, the hyperbolic cosine of a double, float, or long double value, *x*. A range error occurs if they are called with an argument that would cause overflow.

Functions tanh, tanhf, and tanhl return, respectively, the hyperbolic tangent of a double, float, or long double value, *x*.

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, giving a traceback and a core file. Specifying the `cc`(1) command-line option `-h stdc` (signifying strict conformance mode) or `-h matherr=errno` causes all of these functions to perform domain and range checking, set `errno` on error, and return to the caller on error. 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, `SINH`(3M)).

Also, 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

`cosh(`*NaN*`)`, `coshl(`*NaN*`)`, `sinh(`*NaN*`)`, `sinhl(`*NaN*`)`, `tanh(`*NaN*`)`, and `tanhl(`*NaN*`)` return NaN and `errno` is set to `EDOM` on Cray MPP systems and CRAY T90 systems with IEEE arithmetic when the program is compiled with `-hstdc` or `-hmatherror=errno`.

SEE ALSO

`errno.h`(3C)

`SINH`(3M), `COSH`(3M), `TANH`(3M) in the *Intrinsic Procedures Reference Manual*, Cray Research publication SR–2138

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

NAME

`sleep` – Suspends execution for a specified interval

SYNOPSIS

```
#include <unistd.h>
unsigned int sleep (unsigned int seconds);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX

DESCRIPTION

The `sleep` function suspends the current process from execution for at least the number of seconds specified by the argument *seconds*. The suspension time may be longer than requested by an arbitrary amount due to the scheduling of other activity in the system.

FORTRAN EXTENSION

Function `sleep` can also be called from Fortran programs, as follows:

```
INTEGER*8 SLEEP, seconds, I
I = SLEEP(seconds)
```

SEE ALSO

NAME

slgtrust, slgtrustobj – Writes trusted process security log record

SYNOPSIS

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/secparm.h>
#include <sys/priv.h>
#include <sys/utsname.h>
#include <sys/slrec.h>
#include <errno.h>

int slgtrust(char *tpname, char *tpaction);

int slgtrustobj(char *tpname, char *tpaction, int olvl, long ocomp);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The slgtrust routine formats trusted process security log records and requests the kernel to write the records to the security log. The slgtrustobj routine formats trusted process security log records, which contain an object label.

RETURN VALUES

If successful, a 0 is returned. If unsuccessful, a –1 is returned and errno is set to the appropriate value.

SEE ALSO

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

NAME

sort – Introduction to sort/merge routines

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The sort/merge routines let you sort records; the records can be acquired from input files, or they can be generated from within the program. User-supplied routines can be added at various stages of the processing to provide for error messages, end-of-file processing, or other specialized record processing functions.

To use the sort/merge routines, you must include the libsort library through the -lsort option on the segldr(1) command.

The following routines are used to define or set the parameters in a sort/merge session:

SAMSORT, SAMMERGE
: Begins a sort/merge specification; after this call you must call SAMKEY at least once, call either SAMFILE or SAMPATH at least once, and call any other subroutines as needed.

SAMPATH
: Defines the input and output files, as well as other user-supplied routines.

SAMFILE
: Defines input sources and output sinks.

SAMKEY
: Defines the sort keys used; there must be at least one call to SAMKEY between a call to SAMSORT or SAMMERGE and the call to SAMGO which initiates the operation.

SAMOPT
: Specifies sort options, such as verification during the sort/merge session. This routine can also be used to retain the original order of records with equal keys.

SAMSIZE
: Specifies the word and character sizes used for character comparisons.

SAMEQU
: Specifies equivalent characters in sort/merge sessions.

SAMSEQ
: Specifies and defines a collating sequence (ascending or descending order).

SAMTUNE
: Used to modify selected parameters such as average record length, maximum record length, and the number of sort buffers to be allocated to each temporary dataset.

SAMGO
: Initiates 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 at least one intervening call to SAMKEY, at least one call to SAMFILE or SAMPATH to specify input sources, as well as at least one call to SAMFILE or SAMPATH to specify output sinks.

NAME

sqrt, sqrtf, sqrtl, csqrt, hypot – Determines the square root or hypotenuse of a value

SYNOPSIS

```
#include <math.h>
#include <complex.h> (function csqrt only)
double sqrt (double x);
float sqrtf (float x);
long double sqrtl (long double x);
double complex csqrt (double complex x);
double hypot (double x, double y);
```

IMPLEMENTATION

All Cray Research systems (sqrt, csqrt, hypot only)
Cray MPP systems (sqrtf only)
Cray PVP systems (sqrtl only)

STANDARDS

ISO/ANSI (sqrt only)
XPG4 (hypot only)
CRI extension (all others)

DESCRIPTION

The sqrt, sqrtf, sqrtl, and csqrt functions compute the nonnegative square root of *x* for double, float, long double, and double complex numbers, respectively. For these functions, a domain error occurs if the argument is negative.

The hypot function returns the hypotenuse (Euclidean distance) *sqrt(x*x + y*y)*, taking precautions against unwarranted overflows.

When code containing calls to these functions is compiled by Cray Standard C 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 on error. For more information, see the corresponding libm man page (for example, SQRT(3M)).

Specifying the cc(1) command-line option -h stdc (signifying strict conformance mode) or -h matherr=errno causes the sqrt functions to perform domain and range checking, set errno on error, and return to the caller on error. Domain and range checking is always performed by hypot, regardless of the compilation mode.

Also, in strict conformance mode, vectorization is inhibited for loops containing calls to these functions. Vectorization is not inhibited in extended mode for loops containing calls to functions `sqrt`, `sqrtf`, `sqrtl`, and `csqrt`. Vectorization is always inhibited for loops containing calls to the `hypot` routine.

RETURN VALUES

The `sqrt`, `sqrtf`, `sqrtl`, and `csqrt` functions return the `double`, `float`, `long double`, and `double complex` value of the square root, respectively.

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:

- `sqrt(`*NaN*`)` returns NaN, and `errno` is set to `EDOM`; no exception is raised.
- `sqrtl(`*NaN*`)` returns NaN, and `errno` is set to `EDOM`; no exception is raised.
- `hypot(`*NaN*`,`*y*`)` returns NaN, and `errno` is set to `EDOM`, regardless of the compilation mode.
- `hypot(`*x*`,`*NaN*`)` returns NaN, and `errno` is set to `EDOM`, regardless of the compilation mode.

On Cray MPP systems and CRAY T90 systems with IEEE arithmetic, the value returned by these functions when a domain error occurs can be selected by the environment variable `CRI_IEEE_LIBM`. The second column in the following table describes what is returned when `CRI_IEEE_LIBM` is not set, or is set to a value other than 1. The third column describes what is returned when `CRI_IEEE_LIB` is set to 1. For both columns, `errno` is set to `EDOM`.

Error	`CRI_IEEE_LIB`=0	`CRI_IEEE_LIB`=1
`sqrt(`*x*`)`, where *x* is less than zero	0.0	NaN
`sqrtl(`*x*`)`, where *x* is less than zero	0.0	NaN

SEE ALSO

`errno.h`(3C)

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

`cc`(1) (online only)

NAME

ssignal, gsignal – Generates software signals

SYNOPSIS

```
#include <signal.h>
int (*ssignal (int sig, int (*action)(int)));
int gsignal (int sig);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

AT&T extension

DESCRIPTION

The ssignal and gsignal functions implement a software facility similar to signal(2). The C library uses this facility to enable you to indicate the disposition of error conditions, and it is also made available to you for your own purposes.

Software signals made available to users are associated with integers in the inclusive range 1 through 15. A call to ssignal associates a procedure, *action*, with the software signal *sig*; the software signal, *sig*, is raised by a call to gsignal. Raising a software signal causes the action established for the specified signal to be taken.

The first argument to ssignal is a number identifying the type of signal for which an action is to be established. The second argument defines the action; it is either the user-defined name of an *action function* or one of the manifest constants SIG_DFL (default) or SIG_IGN (ignore). The ssignal function returns the action previously established for that signal type; if no action has been established or the signal number is illegal, ssignal returns SIG_DFL.

The gsignal function raises the signal identified by its argument, *sig*:

- If an action function has been established for *sig*, that action is reset to SIG_DFL, and the action function is entered with argument *sig*. The gsignal function returns the value returned to it by the action function.
- If the action for *sig* is SIG_IGN, gsignal returns the value 1 and takes no other action.
- If the action for *sig* is SIG_DFL, gsignal returns the value 0 and takes no other action.
- If *sig* has an illegal value or no action was ever specified for *sig*, gsignal returns the value 0 and takes no other action.

SEE ALSO

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

NAME

start, sitelocal_start – Common start-up routine for programs, user exit for start-up

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

Two modules comprise the start-up code for all programs: $START and $START$. The first is an assembly language module to which the kernel passes information in registers. On non-MPP systems, the start function does executable expansion, presets nonzero BSS space in the program (if requested by using the -f option to segldr(1)), determines the size of the initial heap and stack segments, and jumps to $START$ (this call creates an initial stack segment and heap area for the program). On MPP systems, $START simply stores the registers that the kernel passes to it, and jumps to $START$.

The $START$ routine is written in C, and it does the rest of the initialization necessary for the program, including calling target(2) to get machine-specific information, calling the _siginit(2) system call to register an initial signal save area with the kernel, and calling various initialization routines if they happen to be loaded into the program. On MPP systems, $START$ also initializes the private and shared heap segments, and presets nonzero BSS space if requested.

As the last step before calling the main routine of the program, $START$ checks for the existence of the sitelocal_start routine; if this routine is linked into the program, it will be called. This step allows site-specific "user exit" code to be run before the main routine is called. If the main routine of the program returns, $START$ calls exit(2) with the return value from the main routine as its argument.

MESSAGES

ERROR: fixed heap space too small to copy arguments...
: An expandable blank common block is being used, and the initial heap space specified on the segldr command line is too small. Set the initial heap size to a larger value.

ERROR: program executing at word zero
: Some routine has done a jump to address 0, because of a bug in the program or library code.

$START$: target() syscall failed, program exiting
: The target(2) system call in $START$ returned –1; contact your system administrator or site analyst.

$START$: _siginit() call failed, program exiting
: The _siginit(2) system call in $START$ returned –1; contact your system administrator or site analyst.

$START$: heap initialization failed, program exiting
: The MPP private heap initialization routine failed, either from bad heap values specified to segldr, or from too small a memory limit; contact your system administrator or site analyst.

`$START$: shared heap initialization failed, program exiting`
The MPP shared heap initialization routine failed, either from bad heap values specified to `segldr`, or from too small a memory limit; contact your system administrator or site analyst.

EXAMPLES

The following example shows how to define a routine called `sitelocal_start` that will be run before the main program:

```
$ cat main.c
main()
{
      printf("hello world\n");
}
$ cat sl.c
void
sitelocal_start(void)
{
      printf("in sitelocal_start\n");
}
$ cc -c main.c sl.c
main.c:
sl.c:
$ # using cc to link the program
$ cc -o main main.o sl.o
$ # OR using segldr to link the program
$ segldr -o main main.o sl.o -Dhardref=sitelocal_start
$ ./main
in sitelocal_start
hello world
$
```

SEE ALSO

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

NAME

stdarg.h – Library header for variable arguments

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

TYPES

The type declared in stdarg.h, which conforms to the ISO/ANSI standard, is as follows:

va_list A type suitable for holding information needed by the macros va_start, va_arg, and va_end. If access to the varying arguments is desired, the called function declares an object (referred to as ap in this description) having type va_list. The object ap can be passed as an argument to another function; if that function invokes the va_arg macro with parameter *ap*, the value of ap in the calling function is indeterminate; it is passed to the va_end macro prior to any further reference to ap.

MACROS

The macros declared in stdarg.h, which all conform to the ISO/ANSI standard, are as follows:

va_start Invoke va_start before any access to the unnamed arguments, as follows:

va_start (*ap*, *parmN*);

The va_start macro initializes *ap* for subsequent use by va_arg and va_end .

The parameter *parmN* is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the , . . .). If the parameter *parmN* is declared with the register storage class, with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions, the behavior is undefined.

va_start is a macro, not a function. If the macro definition is suppressed in order to access an actual function, the behavior is undefined.

The va_start macro returns no value.

`va_arg` The `va_arg` macro expands to an expression that has the type and value of the next argument in the call. It is invoked as follows:

```
va_arg (ap, type);
```

The parameter *ap* is the same as the `va_list` *ap* initialized by `va_start`. Each invocation of `va_arg` modifies *ap* so that the values of successive arguments are returned in turn. The parameter *type* is a type name specified such that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a * to *type*. If there is no actual next argument, or if *type* is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior is undefined.

`va_arg` is a macro, not a function. If the macro definition is suppressed in order to access an actual function, the behavior is undefined.

The first invocation of the `va_arg` macro after invocation of the `va_start` macro returns the value of the argument after that specified by *parmN*. Successive invocations return the values of the remaining arguments in succession.

`va_end` The `va_end` macro facilitates a normal return from the function whose variable argument list was referred to by the expansion of macro `va_start` that initialized the `va_list` *ap*. It is invoked as follows:

```
void va_end (va_list ap);
```

The `va_end` macro can modify *ap* so that it is no longer usable (without an intervening invocation of `va_start`). If there is no corresponding invocation of the `va_start` macro, or if the `va_end` macro is not invoked before the return, the behavior is undefined.

`va_end` is a macro, not a function. If the macro definition is suppressed in order to access an actual function, the behavior is undefined.

The `va_end` macro returns no value.

FUNCTION DECLARATIONS

None

NOTES

The preceding describes the Cray Standard C (and ISO/ANSI) approach to variable-length argument list processing. The SVID approach is defined in `varargs.h`. (See `varargs`(3).) The two approaches are not compatible; if both headers are included in the same compilation unit, the compiler issues redefinition error messages.

A typical function definition for a function with variable argument list that uses `stdarg.h` is as follows:

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

f(FILE *iop, char *format, ...)
{
     va_list ap;
     va_start(ap, format);
     . . .
}
```

To use the Cray Standard C compiler without function prototype and elipses notation, change the code as follows:

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

f(iop, format, ap)
     FILE *iop;
     char *format;
     va_list ap;
{
     va_start(ap, format);
     . . .
}
```

SEE ALSO

`varargs.h`(3C)

NAME

`stddef.h` – Library header for common definitions

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

TYPES

The following types are defined in the standard header `stddef.h`. Unless noted as a CRI extension, each item conforms to the ISO/ANSI standard.

Type	Description
`ptrdiff_t`	The signed integral type of the result of subtracting two pointers.
`size_t`	The unsigned integral type of the result of the `sizeof` operator.
`wchar_t`	An integral type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales; the null character shall have the code value zero and each member of the basic character set shall have a code value equal to its value when used as the lone character in an integer character constant.

MACROS

The following macros are defined in the standard header `stddef.h`. Each item conforms to the ISO/ANSI standard.

Type	Description
`NULL`	An implementation-defined null pointer constant, equal to zero on CRI systems. Also defined in headers `locale.h`, `stdio.h`, `stdlib.h`, `string.h`, and `time.h`.
`offsetof(`*type*, *member-designator*)	An integral constant expression that has type `size_t`, the value of which is the offset in bytes, to the structure member (designated by *member-designator*), from the beginning of its structure (designated by *type*). The *member-designator* shall be such that given "`static` *type* `t;`", then the expression `&(t.` *member-designator*) evaluates to an address constant. (If the specified member is a bit-field, the behavior is undefined.)

FUNCTION DECLARATIONS

None

NOTES

`wchar_t` is defined in header files `stdlib.h` and `stddef.h` to be of type `int`. In releases before UNICOS 8.0, it was type `char`.

NAME

stdio.h – Library header for input and output functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The stdio.h header file describes input and output functions.

Types

The types defined in stdio.h are as follows. All types conform to the ISO/ANSI standard.

Type	Description
size_t	The unsigned integral type of the result of the sizeof operator.
FILE	An object type that can record all of the information needed to control a stream, including its file position indicator, a pointer to its associated buffer (if any), an *error indicator* that records whether a read/write error has occurred, and an *end-of-file indicator* that records whether the end of the file has been reached.
fpos_t	An object type that can record all of the information needed to specify every position within a file uniquely.

Macros

The macros defined in stdio.h are as follows. Unless noted, all macros conform to the ISO/ANSI standard.

Macro	Description
BUFSIZ	Expands to an integral constant expression the size of the buffer used by the setbuf(3C) function.
EOF	Expands to a negative integral constant expression that is returned by several functions to indicate *end-of-file*, that is, no more input from a stream. On Cray Research systems, EOF is –1.
FILENAME_MAX	Expands to an integral stdio.h constant expression that is the size needed for an array of char large enough to hold the longest filename string that can be opened (1024 on Cray Research systems, which includes the terminating null character).
FOPEN_MAX	Expands to an integral constant expression that is the minimum number of files that are guaranteed to be open simultaneously (at least 100 on Cray Research systems).
_IOFBF, _IOLBF, _IONBF	Expands to integral constant expressions with distinct values, suitable for use as the third argument to the setvbuf(3C) function.
L_ctermid	(XPG4) Expands to an integral constant expression that is the size needed for an array of char large enough to hold a string that contains the path name of the controlling terminal for the current process.

`L_cuserid` (XPG4) Expands to an integral constant expression that is the size needed for an array of `char` large enough to hold a a character-string representation of the login name of the owner of the current process.

`L_tmpnam` Expands to an integral constant expression that is the size needed for an array of `char` large enough to hold a temporary file name string generated by the `tmpnam`(3C) function.

`NULL` A null pointer constant equal to 0.

`SEEK_CUR`, `SEEK_END`, `SEEK_SET`
Expands to integral constant expressions with distinct values, suitable for use as the third argument to the `fseek`(3C) function.

`stderr`, `stdin`, `stdout`
Expressions of type "pointer to `FILE`" that point to the `FILE` objects associated with the standard error, input, and output streams, respectively.

`TMP_MAX` Expands to an integral constant expression that is the minimum number of unique file names that can be generated by the `tmpnam`(3C) function.

`P_tmpdir` (XPG4) Expands to an integral constant expression that is the size needed for an array of `char` large enough to hold a string that contains the path prefix for used by the `tempnam`(3C) function.

Function Declarations

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

`clearerr`	`fgets`	`ftell`	`putc`	`sscanf`
`ctermid`	`fileno`	`fwrite`	`putchar`	`tempnam`
`cuserid`	`fopen`	`getc`	`puts`	`tmpfile`
`fwrite`	`fprintf`	`getchar`	`putw`	`tmpnam`
`fclose`	`fputc`	`gets`	`remove`	`ungetc`
`fdopen`	`fputs`	`getw`	`rewind`	`vfprintf`
`feof`	`fread`	`mktemp`	`scanf`	`vprintf`
`ferror`	`freopen`	`pclose`	`setbuf`	`vsprintf`
`fflush`	`fscanf`	`perror`	`setlinebuf`	
`fgetc`	`fseek`	`popen`	`setvbuf`	
`fgetpos`	`fsetpos`	`printf`	`sprintf`	

NAME

stdipc, ftok – Standard interprocess communication (IPC) package

SYNOPSIS

```
#include <sys/ipc.h>
key_t ftok (const char *path, int id);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

AT&T extension

DESCRIPTION

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

ftok returns a key based on *path* and *id* that is usable in subsequent msgget(2), semget(2), and shmget(2) functions. *path* must be the path name of an existing file that is accessible to the process. *id* is a character that uniquely identifies a project. Note that ftok returns the same key for linked files when called with the same *id* and that it returns different keys when called with the same file name but different *ids*.

NOTES

If the file whose *path* is passed to ftok is removed when keys still refer to the file, future calls to ftok with the same *path* and *id* return an error. If the same file is recreated, then ftok is likely to return a different key from the original call.

RETURN VALUES

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

SEE ALSO

`msgget`(2), `semget`(2), `shmget`(2)

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

`ipc`(7) Online only

NAME

stdlib.h – Library header for general utility functions

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

TYPES

The types defined in the header file stdlib.h are as follows:

Type	Standards	Description
div_t	ISO/ANSI	Structure type that is the type of the value returned by the div function. It consists of members int quot (quotient) and int rem (remainder), in either order.
ldiv_t	ISO/ANSI	Structure type that is the type of the value returned by the ldiv function.
size_t	ISO/ANSI	The unsigned integral type of the result of the sizeof operator.
wchar_t	ISO/ANSI	An integral type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales; the null character shall have the code value zero and each member of the basic character set shall have a code value equal to its value when used as the lone character in an integer character constant.

MACROS

The macros defined in the header file stdlib.h are as follows:

Macros	Standards	Description
EXIT_FAILURE EXIT_SUCCESS	ISO/ANSI	Both of these macros expand to integral expressions that can be used as the argument to the exit function to return unsuccessful or successful termination status, respectively, to the host environment. On Cray Research systems, the values are 1 and 0, respectively.

Macros	Standards	Description
MB_CUR_MAX	ISO/ANSI	Expands to a positive integer expression whose value is the maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category LC_CTYPE), and whose value is never greater than MB_LEN_MAX. The standard guarantees the value of MB_CUR_MAX to be at least 1. On Cray Research systems, MB_CUR_MAX is defined as 1.
NULL	ISO/ANSI	An implementation-defined null pointer constant, equal to zero on Cray Research systems.
RAND_MAX	ISO/ANSI	Expands to an integral constant expression, the value of which is the maximum value returned by the rand function; The ISO/ANSI C standard guarantees the value of RAND_MAX to be at least 32767. On Cray Research systems, RAND_MAX is defined as 32767.

FUNCTION DECLARATIONS

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

abort	div	labs	mrand48†	srand48†
abs	drand48†	lcong48†	nrand48†	strtod
atexit	erand48†	ldiv	putenv†	strtol
atof	exit	lrand48†	qsort	strtold
atoi	free	malloc	rand	strtoul
atol	getenv	mblen	realloc	system
bsearch	getopt†	mbstowcs	seed48†	wcstombs
calloc	jrand48†	mbtowc	srand	wctomb

† Available only in extended mode

SEE ALSO

stddef.h(3C)

NAME

STKSTAT, STACKSZ – Reports stack statistics

SYNOPSIS

```
#include <stkstat.h>
void STKSTAT(struct stkstat *ptr);
void STACKSZ(void);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

CRI extension

DESCRIPTION

The STKSTAT function fills the structure pointed to by *ptr* with statistics from the stack manager. STACKSZ calls STKSTAT, then prints the statistics (on `stdout`) in the format shown below.

EXAMPLES

This is sample output from STACKSZ on Cray parallel vector systems:

```
04357  Total size of all stacks (2287)
04000  Highest stack hi-water mark (2048)
    2  Current number of stacks
    2  Total number of stacks
    2  Most stacks at one time
    1  Number of stacks that grew
    2  Number of times stacks grew
```

FORTRAN EXTENSIONS

These functions can also be called from Fortran, as shown below. The `ISTAT` array is declared as 20 words long to allow for future growth of the `stkstat` structure.

```
DIMENSION ISTAT(20)
CALL STKSTAT(ISTAT)
```

or

```
CALL STACKSZ
```

NAME

strcasecmp, strncasecmp – Performs case-insensitive string comparison

SYNOPSIS

```
#include <string.h>
int strcasecmp (const char *s1, const char *s2);
int strncasecmp (const char *s1, const char *s2, size_t n);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The strcasecmp function performs a case-insensitive comparison on the strings (arrays of characters terminated by null characters) pointed to by its arguments, mapping uppercase alphabetic characters to lowercase for comparison. It returns an integer less than, equal to, or greater than 0, depending on whether *s1* is lexicographically less than, equal to, or greater than *s2*, respectively.

The strncasecmp function performs the same comparison, but compares a maximum of *n* characters.

SEE ALSO

string(3C)

NAME

`strfmon` – Converts a monetary value to a string

SYNOPSIS

```
#include <monetary.h>
ssize_t strfmon (char *s, size_t maxsize, const char *format, ...);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The `strfmon` function places characters into the array pointed to by *s*, as controlled by the string pointed to by *format*. No more than *maxsize* bytes are placed into the array.

The *format* is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in the fetching of zero or more arguments that are converted and formatted. If there are insufficient arguments for the format, the results are undefined. If the format is exhausted while arguments remain, the excess arguments are simply ignored.

A conversion specification consists of the following sequence:

- A % character
- Optional flags
- Optional field width
- Optional left precision
- Optional right precision
- A required conversion character that determines the conversion to be performed

Flags

One or more of the following optional flags can be specified to control the conversion:

=f The *f* is used as the numeric fill character. The fill character must be representable in a single byte in order to work with precision and width counts. The default numeric fill character is the space character. This flag does not affect field width filling, which always uses the space character. This flag is ignored unless left precision (see the Left Precision subsection) is specified.

^ Do not format the currency amount with grouping characters. The default is to insert the grouping characters if defined for the current locale.

`+` or `(` Specify the style of representing positive and negative currency amounts. Only one of `+` or `(` may be specified. If `+` is specified, the locale's equivalent of `+` and `-` are used (for example, in the United States: the empty string if positive and `-` is negative). If the `(` character is specified, negative amounts are enclosed within parentheses. If neither flag is specified, the `+` style is used.

`!` Suppress the currency symbol from the output conversion.

`-` Specify the alignment. If this flag is present, all fields are left-justified (padded to the right) rather than right-justified.

Field Width

w Decimal number specifying a minimum field width in bytes in which the result of the conversion is right-justified (or left-justified if the `-` flag is specified). The default is zero.

Left Precision

#n The *n* is a decimal number specifying the maximum digits to be formatted to the left of the radix character. This option can be used to keep the formatted output from multiple calls to the `strfmon` aligned in the same columns. It can also be used to fill unused positions with a special character, as in `$***123.45`. This option causes an amount to be formatted as if it has the number of digits specified by *n*. If more than *n* digit positions are required, this conversion specification is ignore. Digit positions in excess of those actually required are filled with the numeric fill character (see the *=f* flag description).

If grouping has not been suppressed by using the `^` flag, and it is defined for the current locale, grouping separators are inserted before the fill characters (if any) are added. Grouping separators are not applied to fill characters, even if the fill character is a digit.

To ensure alignment, any characters appearing before or after the number in the formatted output, such as currency or sign symbols, are padded as necessary with space characters to make their positive and negative formats an equal length.

Right Precision

.p The *p* is a decimal number specifying how many digits follow the radix character. If the value of the right precision *p* is 0, no radix character appears. If a right precision value is not included, a default specified by the current locale is used. The amount being formatted is rounded to the specified number of digits prior to formatting.

Conversion Characters

The conversion characters and their meanings are as follows:

`i` The `double` argument is formatted according to the locale's international currency format (for example, in the United States: `USD 1,234.56`).

`n` The `double` argument is formatted according to the locale's national currency format (for example, in the United States: `$1,234.56`).

`%` Convert to a %; no argument is converted. The entire conversion specification must be %%.

Locale Information

The LC_MONETARY category of the program's locale affects the behavior of this function including the monetary radix character (which may be different from the numeric radix character affected by the LC_NUMERIC category), the grouping separator, the currency symbols and formats. The international currency symbol should be conformant with the ISO 4217:1987 standard.

RETURN VALUES

If the total number of resulting bytes including the terminating null byte is not more than *maxsize*, the strfmon function returns the number of bytes placed into the array pointed to by *s*, not including the terminating null byte. Otherwise, –1 is returned, the contents of the array are indeterminate, and errno is set to indicate the error.

ERRORS

The strfmon function fails if:

[ENOSYS] The function is not supported.

[E2BIG] Conversion stopped due to lack of space in the buffer.

EXAMPLES

The following example shows a locale for the United States and the values 123.45, –123.34, and 3456.781:

Conversion specification	Output	Comments
%n	123.45 -$123.45 $3,456.78	Default formatting
%11n	$123.45 -$123.45 $3,456.78	Right align within an 11-character field
$#5n	$ 123.45 -$ 123.45 $ 3,456.78	Aligned columns for values up to 99,999
%=*#5n	$***123.45 -$***123.45 $*3,456.78	Specify a fill character
%=0#5n	$000123.45 -$000123.45 $03,456.78	Fill characters do not use grouping even if the fill character is a digit

Conversion specification	Output	Comments
%^#5n	$ 123.45 -$ 123.45 $ 3456.78	Disable the grouping separator
%^#5.0n	$ 123 -$ 123 $ 3457	Round off to whole units
%(#5.4n	$ 123.4500 -$ 123.4500 $ 3456.7810	Increase the precision
%(#5n	123.45 ($ 123.45) $3,456.78	Use an alternative positive/negative style
%!(#5n	123.45 (123.45) 3,456.78	Disable the currency symbol

SEE ALSO

localeconv(3C)

NAME

strftime, cftime, ascftime, wcsftime – Formats time information in a character string

SYNOPSIS

```
#include <time.h>

size_t strftime (char *s, size_t maxsize, const char *format,
const struct tm *timeptr);

int cftime (char *s, char *format, const time_t *clock);

int ascftime (char *s, const char *format, const struct tm *timeptr);

#include <wchar.h>

size_t wcsftime (wchar_t *wcs, size_t maxsize, const char *format,
const struct tm *timeptr);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (strftime only)

XPG4 (wcsftime only)

AT&T extension (cftime and ascftime only)

DESCRIPTION

The strftime, cftime, and ascftime functions place characters into the array pointed to by *s*, as controlled by the string pointed to by *format*.

The wcsftime function places wide characters into the array pointed to by *wcs*, as controlled by the string pointed to by *format*. This function behaves as if the character string generated by the strftime function is passed to the mbstowcs function as the character string argument, and that function places the result in the wide character string argument of the wcsftime function up to a limit of *maxsize* wide characters.

The format is a multibyte character sequence, beginning and ending in its initial shift state. The *format* string consists of zero or more conversion specifiers and ordinary multibyte characters. A conversion specifier consists of a % character followed by a character that determines the behavior of the conversion specifier. All ordinary multibyte characters (including the terminating null character) are copied unchanged into the array. If copying takes place between objects that overlap, the behavior is undefined. For functions strftime and wcsftime, no more than *maxsize* characters or wide characters are placed into the array.

If *format* is `(char *)0`, the locale's default format is used. For `strftime` and `wcsftime`, the default format is the same as `%c`; for `cftime` and `ascftime`, the default format is the same as `%F`. Functions `cftime` and `ascftime` first try to use the value of the environment variable `CFTIME`, and if that is undefined or empty, the default format is used.

Each conversion specifier is replaced by appropriate characters, as described in the following list. The appropriate characters are determined by the `LC_TIME` category of the current locale and by the values contained in the structure pointed to by *timeptr* for `strftime`, `ascftime`, and `wcsftime`, and by the time represented by *clock* for `cftime`.

Conversion specifiers are as follows:

- `%a` Replaced by the locale's abbreviated weekday name.
- `%A` Replaced by the locale's full weekday name.
- `%b` Replaced by the locale's abbreviated month name.
- `%B` Replaced by the locale's full month name.
- `%c` Replaced by the locale's appropriate date and time representation.
- `%C` † Century (the year divided by 100 and truncated to an integer) as a decimal number (00–99).
- `%d` Replaced by the day of the month as a decimal number (01–31).
- `%D` † Replaced by the date as %*m*/%*d*/%*y*.
- `%e` † Replaced by the day of month (01–31; single digits are preceded by a blank).
- `%F` † Replaced by the date and time as produced by `date`(1) (formerly `%C`).
- `%h` † A synonym for `%b`.
- `%H` Replaced by the hour (24-hour clock) as a decimal number (00–23).
- `%I` Replaced by the hour (12-hour clock) as a decimal number (01–12).
- `%j` Replaced by the day of the year as a decimal number (001–366).
- `%m` Replaced by the month as a decimal number (01–12).
- `%M` Replaced by the minute as a decimal number (00–59).
- `%n` † Replaced by a <newline> character.
- `%p` Replaced by the locale's equivalent of the AM/PM designations associated with a 12-hour clock.
- `%r` † Replaced by the time as %*I*:%*M*:%*S* %*p*.
- `%R` † Replaced by the time as %*H*:%*M*.
- `%S` Replaced by the second as a decimal number (00–61).
- `%t` † Replaced by a <tab> character.
- `%T` † Replaced by the time as %H:%M:%S.
- `%u` † Replaced by the weekday as a decimal number (1(Monday)–7).
- `%U` Replaced by the week number of the year (Sunday as the first day of week 1) as a decimal number (00–53).
- `%w` Replaced by the weekday as a decimal number (0(Sunday)–6).
- `%W` Replaced by the week number of the year (Monday as the first day of week 1) as a decimal number (00–53).
- `%x` Replaced by the locale's appropriate date representation.
- `%X` Replaced by the locale's appropriate time representation.
- `%y` Replaced by the year without century as a decimal number (00–99).

%Y Replaced by the year with century as a decimal number.
%Z Replaced by the time zone name or abbreviation, or by no characters if no time-zone is determinable.
%% Replaced by %.

† Conversion specifier is not part of the ISO/ANSI Standard.

If a conversion specifier is not one of the above, the behavior is undefined.

The difference between %U and %W lies in which day is counted as the first of the week. Week number 01 is the first week in January starting with a Sunday for %U or a Monday for %W. Week number 00 contains those days before the first Sunday or Monday in January for %U and %W, respectively.

Some conversion specifiers can be modified by the E or O modifier characters to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified conversion specifier. If the alternative format or specification does not exist for the current locale, the behavior will be as if the unmodified conversion specifier were used.

Modified conversion specifiers are as follows:

%Ec Replaced by the locale's alternate appropriate date and time representation.
%EC Replaced by the name of the base year (period) in the locale's alternative representation.
%Ex Replaced by the locale's alternate date representation.
%EX Replaced by the locale's alternate time representation.
%Ey Replaced by the offset from %EC (year only) in the locale's alternative representation.
%EY Replaced by the full alternative year representation.
%Od Replaced by the day of the month, using the locale's alternative numeric symbols, filled as needed with leading zeros if there is any alternative symbol for zero, otherwise with leading spaces.
%Oe Replaced by the day of the month, using the locale's alternative numeric symbols, filled as needed with leading spaces.
%OH Replaced by the hour (24-hour clock), using the locale's alternative numeric symbols.
%OI Replaced by the hour (12-hour clock), using the locale's alternative numeric symbols.
%Om Replaced by the month using the locale's alternative numeric symbols.
%OM Replaced by the minutes using the locale's alternative numeric symbols.
%OS Replaced by the seconds using the locale's alternative numeric symbols.
%Ou Replaced by the weekday as a number in the locale's alternative representation (Monday=1).
%OU Replaced by the week number of the year (Sunday as the first day of the week, rules corresponding to %U), using the locale's alternative numeric symbols.
%OV Replaced by the week number of the year (Monday as the first day of the week, rules corresponding to %V), using the locale's alternative numeric symbols.
%Ow Replaced by the number of the weekday (Sunday=0) using the locale's alternative numeric symbols.
%OW Replaced by the week number of the year (Monday as the first day of the week), using the locale's alternative numeric symbols.

`%Oy` Replaced by the year (offset from `%C`) in the locale's alternative representation and using the locale's alternative numeric symbols.

NOTES

By default, the output of `strftime`, `cftime`, `ascftime`, and `wscftime` appears in U.S. English. The user can request that the output of `strftime`, `cftime`, `ascftime`, or `wscftime` be in a specific language by setting the *locale* for *category* `LC_TIME` in `setlocale`.

The time zone is taken from the environment variable `TZ` (see `ctime`(3C) for a description of `TZ`).

RETURN VALUES

If the total number of resulting characters (including the terminating null character) is not more than *maxsize*, `strftime`, `cftime`, and `ascftime` return the number of characters placed into the array pointed to by *s*. The `wcsftime` function returns the number of wide characters rather than characters. The terminating null character is not included in the count. Otherwise, 0 is returned, and the contents of the array are indeterminate.

EXAMPLES

The following example illustrates the use of function `strftime`. It shows what the string in `str` would look like if the structure pointed to by `tmptr` contains the values corresponding to Thursday, August 28, 1986.

```
strftime (str, strsize, "%A %b %d %j", tmptr)
```

With this call, `str` would contain "`Thursday Aug 28 240`".

FILES

`/usr/lib/locale/`*language*`/LC_TIME` Contains locale specific date and time information.

SEE ALSO

`ctime`(3C), `getenv`(3C), `setlocale`(3C)

`environ`(7) (available only online)

NAME

str_han – Introduction to string-handling functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The string-handling functions provide various means for manipulating arrays of characters or for manipulating other objects treated as arrays of characters. Also included in this section are routines that work similarly to the string routines, but operate on arrays of words instead of characters. In the CRI implementation, characters are 8-bit bytes, with 8 bytes per word of memory. The start of a string may or may not be at the start of a word, but all the string-handling functions take this into account.

Various methods are used for determining the lengths of the arrays, but in all cases a `char *` or `void *` argument points to the initial (lowest addressed) character of the array. If an array is accessed beyond the end of an object, the behavior is undefined.

ASSOCIATED HEADERS

<memory.h> Contains prototypes for string-handling functions.

<string.h> Contains prototypes for string-handling functions.

<strings.h> Contains prototypes for string-handling functions.

ASSOCIATED FUNCTIONS

Copying Functions

bcopy – Copies bytes from one byte array to another byte array (see bstring)
memcpy – Copies object in memory (see memory)
memmove – Moves object in memory (see memory)
memwcpy – Copies words from one common memory address to another (see memword)
strcpy – Copies a string into an array (see string)
strncpy – Copies *n* characters of string into an array (see string)
swab – Swaps bytes

Concatenation Functions

strcat – Appends copy of string to another string (see string)
strncat – Appends characters from array to string (see string)

Comparison Functions

bcmp – Compares byte arrays (see bstring)
memcmp – Compares two objects in memory (see memory)
strcasecmp – Performs case-insensitive string comparison
strcmp – Compares strings (see string)
strcoll – Compares strings as interpreted by LC_COLLATE (see string)
strncasecmp – Performs case-insensitive string comparison (see strcasecmp)
strncmp – Compares characters in arrays (see string)

Search Functions:

index – Locates first occurrence of characters in string
memchr – Locates a character in memory (see memory)
rindex – Locates last occurrence of characters in string (see index)
strchr – Locates characters in string (see string)
strcspn – Computes length of substring (see string)
strpbrk – Locates any character of a string in another string (see string)
strrchr – Locates last occurrence of *c* in string (see string)
strspn – Computes length of substring (see string)
strstr – Locates first occurrence of characters in null-terminated string (see string)
strrstr – Locates last occurrence of characters in null-terminated string (see string)
strnstrn – Locates first occurrence of characters in string (see string)
strnstrn – Locates last occurrence of characters in string (see string)

Miscellaneous Functions:

bzero – Places bytes of 0's in a byte array (see bstring)
ffs – Finds the first bit set in the argument, passes it, and returns the index of that bit (see bstring)
memset – Copies a value to memory (see memory)
memwset – Copies a word value to a word array (see memword)
strdup – Returns a copy of a string (see string)
strerror – Maps error number to error message string (see string)
strlen – Computes length of string (see string)
strtok – Breaks string into tokens (see string)
strxfrm – Transforms strings (see string)

SEE ALSO

utilities(3C) for functions that convert strings to numbers or numbers to strings

NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok, strtok_r, strcoll, strerror, strstr, strrstr, strnstrn, strnrstrn, strxfrm, strdup – Performs string operations

SYNOPSIS

```
#include <string.h>
char *strcat (char *s1, const char *s2);
char *strncat (char *s1, const char *s2, size_t n);
int strcmp (const char *s1, const char *s2);
int strncmp (const char *s1, const char *s2, size_t n);
char *strcpy (char *s1, const char *s2);
char *strncpy (char *s1, const char *s2, size_t n);
size_t strlen (const char *s);
char *strchr (const char *s, int c);
char *strrchr (const char *s, int c);
char *strpbrk (const char *s1, const char *s2);
size_t strspn (const char *s1, const char *s2);
size_t strcspn (const char *s1, const char *s2);
char *strtok (char *s1, const char *s2);
char *strtok_r (char *s1, const char *s2, char **lasts);
int strcoll (const char *s1, const char *s2);
char *strerror (int errnum);
char *strstr (const char *s1, const char *s2);
char *strrstr (const char *s1, const char *s2);
char *strnstrn (const char *s1, size_t n1, const char *s2, size_t *s2);
char *strnrstrn (const char *s1, size_t n1, const char *s2,
size_t *s2);
size_t strxfrm (const char *s1, const char *s2, size_t n);
char *strdup (const char *s1);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (except `strdup`, `strrstr`, `strnstrn`, `strnrstrn`, and `strtok_r`)
PThreads (`strtok_r` only)
AT&T extension (`strdup` only)
CRI extension (`strrstr`, `strnstrn`, `strnrstrn` only)

DESCRIPTION

With any of the string functions, if copying takes place between objects that overlap, the behavior is undefined.

The `strcat` function appends a copy of the string pointed to by *s2* (including the terminating null character) to the end of the string pointed to by *s1*. The `strncat` function appends not more than *n* characters (a null character and characters that follow it are not appended) from the array pointed to by *s2* to the end of the string pointed to by *s1*. With both functions, the initial character of *s2* overwrites the null character at the end of *s1*. Function `strncat` always appends a terminating null character to the result.

The `strcmp` function returns an integer that is greater than, equal to, or less than 0, according to whether the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2*. The `strncmp` function compares not more than *n* characters (characters that follow a null character are not compared) from the array pointed to by *s1* to the array pointed to by *s2*.

The `strcpy` function copies the string pointed to by *s2* (including the terminating null character) into the array pointed to by *s1*. The `strncpy` function copies not more than *n* characters (characters that follow a null character are not copied) from the array pointed to by *s2* to the array pointed to by *s1*. If the array pointed to by *s2* is a string that is shorter than *n* characters, null characters are appended to the copy in the array pointed to by *s1*, until *n* characters in all have been written.

The `strlen` function computes the length of the string pointed to by *s*.

The `strchr` function locates the first occurrence of *c* (converted to a `char`) in the string pointed to by *s*. The `strrchr` function locates the last occurrence of *c* (converted to a `char`) in the string pointed to by *s*. With both functions, the terminating null character is considered to be part of the string.

The `strpbrk` function locates the first occurrence in the string pointed to by *s1* of any character from the string pointed to by *s2*.

The `strspn` function computes the length of the maximum initial segment of the string pointed to by *s1* that consists entirely of characters from the string pointed to by *s2*. The `strcspn` function computes the length of the maximum initial segment of the string pointed to by *s1* that consists entirely of characters *not* from the string pointed to by *s2*.

A sequence of calls to the `strtok` function breaks the string pointed to by *s1* into a sequence of tokens, each of which is delimited by a character from the string pointed to by *s2*. The first call in the sequence has *s1* as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by *s2* may be different from call to call. The first call in the sequence searches the string pointed to by *s1* for the first character that is *not* contained in the current separator string pointed to by *s2*. If no such character is found, then there are no tokens in the string pointed to by *s1*, and the `strtok` function returns a null pointer. If such a character is found, it is the start of the first token.

The `strtok` function then searches for a character that *is* contained in the current separator string (*s2*). If no such character is found, the current token extends to the end of the string pointed to by *s1*, and subsequent searches for a token will return a null pointer. If such a character is found, it is overwritten by a null character, which terminates the current token. The `strtok` function saves a pointer to the following character, from which the next search for a token starts. Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as previously described.

The `strtok_r` function provides functionality equivalent to the `strtok` function but with an interface that is safe for multitasked applications. Only an additional argument, *lasts*, is needed to keep track of the search through the string in order to allow for overlapped execution under multitasking.

The `strcoll` function compares the string pointed to by *s1* to the string pointed to by *s2*, both interpreted as appropriate to the `LC_COLLATE` category of the current locale.

The `strerror` function maps the error number in *errnum* to an error message string.

The `strstr` function locates the first occurrence in the string pointed to by *s1* of the sequence of characters in the string pointed to by *s2*. The `strnstrn` function operates identically except that the lengths of the strings are passed as arguments.

The `strrstr` function locates the last occurrence in the string pointed to by *s1* of the sequence of characters in the string pointed to by *s2*. The `strnrstrn` function operates identically except that the lengths of the strings are passed as arguments.

The `strxfrm` function transforms the string pointed to by *s2* and places the resulting string into the array pointed to by *s1*. The transformation is such that if the `strcmp` function is applied to two transformed strings, it returns a value greater than, equal to, or less than 0, corresponding to the result of the `strcoll` function applied to the same two original strings. No more than *n* characters, including the terminating null character, are placed into the resulting array pointed to by *s1*. If *n* is zero, *s1* is permitted to be a null pointer.

The `strdup` function duplicates string *s1* in newly allocated space and returns either a pointer to the new string or null if the required space cannot be allocated. The space is allocated by a call to `malloc`(3C), so it can later be freed by a call to `free`, if desired.

NOTES

The `strdup` function is the only string function that allocates storage. All of the other string functions use space provided by the caller.

The `strcmp` and `strncmp` functions use native character comparison, which is unsigned on Cray Research computer systems and signed on some other machines. Thus, the sign of the value returned when one of the characters has its high-order bit set is implementation dependent.

RETURN VALUES

The `strcat`, `strncat`, `strcpy`, `strncpy` functions return the value of *s1*.

The `strncmp` function returns an integer that is greater than, equal to, or less than 0, according to whether the possibly null-terminated array pointed to by *s1* is greater than, equal to, or less than the possibly null-terminated array pointed to by *s2*.

The `strlen` function returns the number of characters that precede the terminating null character.

The `strchr` and `strrchr` functions return a pointer to the located character, or a null pointer if the character does not occur in the string.

The `strpbrk` function returns a pointer to the character, or a null pointer if no character from *s2* occurs in *s1*.

The `strspn` function returns the length of the maximum initial segment of the string pointed to by *s1* that consists entirely of characters from the string pointed to by *s2*. The `strcspn` function returns the length of the maximum initial segment of the string pointed to by *s1* that consists entirely of characters *not* from the string pointed to by *s2*.

The `strtok` and `strtok_r` functions return a pointer to the first character of a token, or a null pointer if there is no token.

The `strcoll` function returns an integer that is greater than, equal to, or less than 0, according to whether the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2* when both are interpreted as appropriate to the current locale.

The `strerror` function returns a pointer to the string, the contents of which are implementation-defined. The array pointed to cannot be modified by the program, but can be overwritten by a subsequent call to the `strerror` function.

The `strstr` and `strnstrn` functions return a pointer to the located string, or a null pointer if the string is not found. If *s2* points to a string with zero length, string, or a null pointer if the string is not found. If *s2* points to a string with zero length, the function returns *s1*.

The `strrstr` and `strnrstrn` functions return a pointer to the located string, or a null pointer if the string is not found. If *s2* points to a string with zero length, the function returns a pointer to the end of *s1*.

The `strxfrm` function returns the length of the transformed string (not including the terminating null character). If the value returned is *n* or more, the contents of the array pointed to by *s1* are indeterminate.

SEE ALSO

`locale`(3C), `malloc`(3C)

NAME

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

TYPES

The following types are defined in header `string.h`. Unless noted, these are CRI extensions and do not conform to the ISO/ANSI standard.

Type	Description
`size_t`	The unsigned integral type of the result of the `sizeof` operator. ISO/ANSI standard.
`_GPTR`	This `typedef` is provided as a migration aid. It is defined to be a *generic pointer*; when used with the Cray Standard C compiler, `_GPTR` is defined as a pointer to `void`. Also defined in headers `malloc.h`, `stddef.h`, `stdlib.h`, and `stdio.h`.
`_GPTR2CONST`	This `typedef` is provided as a migration aid. It is defined to be a *generic pointer*; when used with the Cray Standard C compiler, `_GPTR2CONST` is equivalent to a pointer to `const void`. Also defined in headers `malloc.h`, `stdlib.h`, `stdio.h`, and `stddef.h`.

MACROS

The macro defined in header `string.h` is as follows:

Type	Description
`NULL`	A null pointer constant equal to zero. ISO/ANSI standard.

FUNCTION DECLARATIONS

Functions declared in header `string.h` are as follows:

```
memccpy     memwcpy     strcpy      strncmp     strrstr
memchr      memwset     strcspn     strncpy     strspn
memcmp      strcat      strdup      strnrstrn   strstr
memcpy      strchr      strerror    strnstrn    strtok
memmove     strcmp      strlen      strpbrk     strxfrm
memset      strcoll     strncat     strrchr
```

NAME

`strings.h` – Library header file for string-handling functions

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

Header `strings.h` is identical to header `string.h`; it is included only for BSD compatibility.

NAME

`strptime` – Date and time conversion

SYNOPSIS

```
#include <time.h>
char *strptime (const char *buf, const char *format, struct tm *tm);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The `strptime` function converts the character string pointed to by *buf* to values that are stored in the `tm` structure pointed to by *tm*, using the format specified by *format*.

The *format* is composed of zero or more directives. Each directive is composed of one of the following: one or more white-space characters (as specified by the `isspace` function), an ordinary character (neither % nor a white-space character), or a conversion specification. Each conversion specification is composed of a % character followed by a conversion character that specifies the replacement required. There must be white-space or other nonalphanumeric characters between any two conversion specifications. The following conversion specifications are supported:

- %a The day of week, using the locale's weekday names; either the abbreviated or full name may be specified.
- %A The same as %a.
- %b The month, using the locale's month names; either the abbreviated or full name may be specified.
- %B The same as %b.
- %c The date and time, using locale's date and time format (for example, as *%x%X*).
- %C The century: a number from 0 through 99; leading zeros are permitted but not required.
- %d The day of the month: a number from 1 through 31; leading zeros are permitted but not required.
- %D The date as *%m/%d/%y*.
- %e The same as %d.
- %h The same as %b.
- %H The hour (24-hour clock): a number from 0 through 23; leading zeros are permitted but not required.
- %I The hour (12-hour clock): a number from 1 through 12; leading zeros are permitted but not required.

%j	The day of the year: a number from 1 through 366; leading zeros are permitted but not required.
%m	The month: a number from 1 through 12; leading zeros are permitted but not required.
%M	The minute: a number from 0 through 59; leading zeros are permitted but not required.
%n	Any white space.
%p	The locale's equivalent of A.M. or P.M.
%r	The time as *%I*:*%M*:*%S%p*.
%R	The time as %H:*%M*.
%S	The seconds: a number from 0 through 61; leading zeros are permitted but not required.
%t	Any white space.
%T	The time as *%H*:*%M*:*%S*.
%U	The week number of the year (Sunday as the first day of the week) as a decimal number from 00 through 53; leading zeros are permitted but not required.
%w	The weekday: a number from 0 through 6, with 0 representing Sunday; leading zeros are permitted but not required.
%W	The week of the year: a number from 00 through 53 (Monday as the first day of the week); leading zeros are permitted but not required.
%x	The date, using the locale's date format.
%X	The time, using the locale's time format.
%y	The year within the century: a number from 0 through 99; leading zeros are permitted but not required. Two digit years 69 through 99, inclusive, are interpreted as 1900 plus the two digit year, whereas two digit years 0 through 68, inclusive, are interprested as 2000 plus the two digit year.
%Y	The year, including the century (for example, 1988).
%%	Replaced by %.

Modified Directives

Some directives can be modified by the E and O modifier characters to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified directive. If the alternative format or specification does not exist in the current locale, the behavior will be as if the unmodified directive were used.

%Ec	The locale's alternative appropriate date and time representation.
%EC	The name of the base year (period) in the locale's alternative representation.
%Ex	The locale's alternative date representation.
%EX	The locale's alternative time representation.
%Ey	The offset from %EC (year only) in the locale's alternative representation.
%EY	The full alternative year representation.
%Od	The day of the month using the locale's alternative numeric symbols; leading zeros are permitted but not required.
%Oe	The same as %Od.
%OH	The hour (24-hour clock) using the locale's alternative numeric symbols.
%OI	The hour (12-hour clock) using the locale's alternative numeric symbols.
%Om	The month using the locale's alternative numeric symbols.

%OS The seconds using the locale's alternative numeric symbols.
%OU The week number of the year (Sunday as the first day of the week) using the locale's alternative numeric symbols.
%Ow The number of the weekday (Sunday=0) using the locale's alternative numeric symbols.
%OW The week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.
%Oy The year (offset from %C) in the locale's alternative representation and using the locale's alternative numeric symbols.

A directive composed of white-space characters is executed by scanning input up to the first character that is not white space (which remains unscanned), or until no more characters can be scanned.

A directive that is an ordinary character is executed by scanning the next character from the buffer. If the character scanned from the buffer differs from the one comprising the directive, the directive fails, and the differing and subsequent characters remain unscanned.

A series of directives composed of %n, %t, white-space characters or any combination is executed by scanning up to the first character that is not white space (which remains unscanned), or until no more characters can be scanned.

Any other conversion specification is executed by scanning characters until a character matching the next directive is scanned, or until no more characters can be scanned. These characters, except the one matching the next directive, are ten compared to the locale values associated with the conversion specifier. If a match is found, values for the appropriate tm structure members are set to values corresponding to the locale information. Case is ignored when matching items in *buf* such as month or weekday names. If no match is found, strptime fails and no more characters are scanned.

RETURN VALUES

Upon successful completion, strptime returns a pointer to the character following the last character parsed. Otherwise, a null pointer is returned.

SEE ALSO

scanf(3C), strftime(3C), time.h(3C)

NAME

strtod, strtold, strtof, atof, wcstod – Converts string to double, long double, or float

SYNOPSIS

```
#include <stdlib.h>
double strtod (const char *nptr, char **endptr);
long double strtold (const char *nptr, char **endptr);
float strtof (const char *nptr, char **endptr);
double atof (const char *nptr);

#include <wchar.h>
double wcstod (const wchar_t *nptr, wchar_t **endptr);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (strtod and atof only)

XPG4 (wcstod only)

CRI extension (strtold and strtof)

DESCRIPTION

The strtod, strtold, and strtof functions convert the initial portion of the string to which *nptr* points to double, long double, or float representation, respectively. The wcstod function is similar to the strtod function, except that it converts a wide character string rather than a character string. First, each function breaks the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the isspace or iswspace macros); a subject sequence that resembles a floating-point constant; and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then they try to convert the subject sequence to a floating-point number and return the result.

The expected form of the subject sequence is an optional plus or minus sign, then a nonempty sequence of digits optionally containing a decimal-point character, then an optional exponent part, but no floating suffix. The subject sequence is defined as the longest initial subsequence of the input string, starting with the first nonwhite-space character that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first nonwhite-space character is other than a sign, a digit, or a decimal-point character.

If the subject sequence has the expected form, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) is interpreted as a floating constant, except that the decimal-point character is used in place of a period, and if neither an exponent part nor a decimal-point character appears, a decimal point is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object to which *endptr* points, provided that *endptr* is not a null pointer.

In a locale other than the "C" locale, additional implementation-defined subject sequence forms may be accepted. (See `locale.h`(3C).)

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object to which *endptr* points, provided that *endptr* is not a null pointer.

The `atof` function converts the initial portion of the string to which *nptr* points to `double` representation. It is equivalent to the following:

```
strtod(nptr, (char **)NULL)
```

RETURN VALUES

These functions return the converted value, if any. If no conversion could be performed, 0 is returned. If the correct value is outside the range of representable values, plus or minus `HUGE_VAL` (`HUGE_VALL` for `strtold`, and `HUGE_VALF` for `strtof`) is returned (according to the sign of the value), and the value of the `ERANGE` macro is stored in `errno`. `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL` are defined in `math.h`. If the correct value would cause underflow, 0 is returned, and the value of the `ERANGE` macro is stored in `errno`.

SEE ALSO

`ecvt`(3C), `errno.h`(3C), `float.h`(3C), `math.h`(3C), `isspace` (see `ctype`(3C)), `iswspace` (see `wctype`(3C)), `locale.h`(3C), `strtol`(3C)

NAME

strtol, strtoll, strtoul, strtoull, atol, atoll, atoi, wcstol, wcstoll, wcstoul, wcstoull – Converts string to integer

SYNOPSIS

```
#include <stdlib.h>

long int strtol (const char *nptr, char **endptr, int base);

long long int strtoll (const char * restrict nptr, char ** restrict
endptr, int base);

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

unsigned long long int strtoull (const char * restrict nptr, char **
restrict endptr, int base);

long int atol (const char *nptr);

long long int atol (const char *nptr);

int atoi (const char *nptr);


#include <wchar.h>

long int wcstol (const wchar_t *nptr, wchar_t **endptr, int base);

long long int wcstoll (const wchar_t * restrict nptr, wchar_t **
restrict endptr, int base);

unsigned long int wcstoul (const wchar_t *nptr, wchar_t **endptr, int
base);

unsigned long long int wcstoull (const wchar_t * restrict nptr, wchar_t
** restrict endptr, int base);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (all except wcstol and wcstoul)

XPG4 (wcstol and wcstoul only)

DESCRIPTION

The `strtol`, `strtoul`, `strtoll`, and `strtoull` functions convert the initial portion of the string to which *nptr* points to `long int`, `unsigned long int`, `long long int`, and `unsigned long long int` representation, respectively. The `wcstol`, `wcstoul`, `wcstoll`, and `wcstoull` functions operate similarly to `strtol`, `strtoul`, `strtoll`, and `strtoull`, respectively; however, they operate on wide character strings rather than character strings. First they break the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the `isspace` or `iswspace` macros); a subject sequence that resembles an integer represented in some radix determined by the value of *base*; and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then they try to convert the subject sequence to an integer, and return the result.

If the value of *base* is 0, the expected form of the subject sequence is that of an integer constant, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from a (or A) through z (or Z) are ascribed the values 10 through 35; only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first nonwhite-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first nonwhite-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form, and the value of *base* is 0, the sequence of characters starting with the first digit is interpreted as an integer constant. The base is determined by the string itself, as follows: After an optional leading sign, a leading 0 indicates octal conversion, and a leading `0x` or `0X` hexadecimal conversion. Otherwise, decimal conversion is used.

If the subject sequence has the expected form, and the value of *base* is between 2 and 36, it is used as the base for conversion, assigning to each letter its value. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object to which *endptr* points, provided that *endptr* is not a null pointer.

In a locale other than the C locale, additional implementation-defined subject sequence forms may be accepted. (See `locale.h`(3C).)

If the subject sequence is empty or does not have the expected form, no conversion is performed; in that case, the value of *nptr* is stored in the object to which *endptr* points, provided that *endptr* is not a null pointer.

The `atol` function converts the initial portion of the string to which *nptr* points to `long int` representation. Except for the behavior on error, it is equivalent to the following:

```
strtol(nptr, (char **)NULL, 10)
```

The `atoll` function converts the initial portion of the string to which *nptr* points to `long long int` representation. Except for the behavior on error, it is equivalent to the following:

```
strtoll(nptr, (char **)NULL, 10)
```

The `atoi` function converts the initial portion of the string to which *nptr* points to `int` representation. Except for the behavior on error, it is equivalent to the following:

```
(int)strtol(nptr, (char **)NULL, 10)
```

(Unlike `strtol`, functions `atoi`, `atol`, and `atoll` ignore overflow conditions and do not set `errno`.)

RETURN VALUES

These functions return the converted value, if any. If no conversion can be performed, 0 is returned. If the correct value is outside the range of representable values, `strtol` and `wcstol` return `LONG_MAX` or `LONG_MIN`, according to the sign of the value, and `strtoul` and `wcstoul` return `ULONG_MAX`. For all functions, the value of the macro `ERANGE` is stored in `errno`.

SEE ALSO

`atof`(3C), `errno.h`(3C), `isspace` (see `ctype`(3C)), `iswspace` (see `wctype`(3C)), `locale.h`(3C), `scanf`(3C), `strtod`(3C)

NAME

swab – Swaps bytes

SYNOPSIS

```
#include <unistd.h>
void swab (const void *from, void *to, ssize_t nbytes);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The swab function copies *nbytes* bytes pointed to by *from* to the array pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between machines. The *nbytes* argument should be even and positive. If *nbytes* is odd and positive, swab uses *nbytes* – 1 instead. If *nbytes* is negative, swab does nothing.

NAME

sysctl – Gets or sets system information

SYNOPSIS

```
#include <sys/sysctl.h>

int sysctl (int *name, u_int namelen, void *oldp, size_t *oldlenp, void *newp,
size_t newlen
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The sysctl function retrieves system information and allows processes with appropriate privileges to set system information. The information available from sysctl consists of integers, strings, and tables. Information may be retrieved and set from the command interface using the sysctl(8) utility.

Unless explicitly noted below, sysctl returns a consistent snapshot of the data requested. Consistency is obtained by locking the destination buffer into memory so that the data may be copied out without blocking. Calls to sysctl are serialized to avoid deadlock.

The state is described using a Management Information Base (MIB) style name, listed in *name*, which is a *namelen* length array of integers.

The information is copied into the buffer specified by *oldp*. The size of the buffer is given by the location specified by *oldlenp* before the call, and that location gives the amount of data copied after a successful call. If the amount of data available is greater than the size of the buffer supplied, the call supplies as much data as fits in the buffer provided and returns with the error code ENOMEM. If the old value is not desired, *oldp* and *oldlenp* should be set to NULL.

The size of the available data can be determined by calling sysctl with a NULL parameter for *oldp*. The size of the available data will be returned in the location pointed to by *oldlenp*. For some operations, the amount of space may change often. For these operations, the system attempts to round up so that the returned size is large enough for a call to return the data shortly thereafter.

To set a new value, *newp* is set to point to a buffer of length *newlen* from which the requested value is to be taken. If a new value is not to be set, *newp* should be set to NULL and *newlen* set to 0.

The top level names are defined with a CTL_ prefix in <sys/sysctl.h>, and are as follows. The next and subsequent levels down are found in the include files listed here, and described in separate sections below.

Name	Next level names	Description
CTL_MBUF	sys/mbuf.h	Network Memory Management
CTL_NSEC	sys/sysctl.h	Network Security
CTL_NET	sys/socket.h	Networking
CTL_USER	sys/sysctl.h	User-level

CTL_MBUF

The table and integer information available for the CTL_MBUF level is detailed below. The changeable column shows whether a process with appropriate privilege may change the value.

Second level name	Type	Changeable
MBUFCTL_STAT	struct	no
MBUFCTL_MBREQ	struct	no
MBUFCTL_MBDENIED	struct	no
MBUFCTL_SLEEPS	struct	no
MBUFCTL_HEADER	struct	no
MBUFCTL_DATA	struct	no
MBUFCTL_NMBSPACE	integer	no
MBUFCTL_MHBASE	integer	no
MBUFCTL_MDBASE	integer	no

The second level of the CTL_MBUF level is used mainly by netstat(1B) to display information about mbuf allocation.

CTL_NSEC

The table and integer information available for the CTL_NSEC level is detailed below. The changeable column shows whether a process with appropriate privileges may change the value.

Second level name	Type	Changeable
NSEC_NAL	struct	no
NSEC_WAL	struct	no
NSEC_MAP	struct	no
NSEC_ENABLED	integer	no

NSEC_NAL

Returns a copy of the Network Access List (NAL)

NSEC_WAL
Returns a copy of the Workstation Access List (WAL)

NSEC_MAP
Returns a copy of the IP Security Option (IPSO) map

NSEC_ENABLED
Returns a value of 1 if Network Security is enabled (otherwise 0)

CTL_NET The second level for the CTL_NET level is the following protocol families: PF_INET, PF_LINK, PF_ROUTE, PF_SOCK, PF_TRACE, and PF_UNIX.

PF_INET The third level name for the PF_INET level are the following internet protocols: IPPROTO_IP, IPPROTO_ICMP, IPPROTO_IGMP, IPPROTO_TCP, and IPPROTO_UDP

IPPROTO_IP
The table and integer information for the IPPROTO_IP level is detailed below. The changeable column shows whether a process with appropriate privilege may change the value.

Fourth level name	Type	Changeable
IPCTL_FORWARDING	integer	yes
IPCTL_SENDREDIRECTS	integer	yes
IPCTL_STAT	struct	no
IPCTL_QMAXLEN	integer	yes
IPCTL_SUBNETSARELOCAL	integer	yes
IPCTL_MRTPROTO	integer	no
IPCTL_MRTSTAT	struct	no
IPCTL_VIFTABLE	struct	no
IPCTL_MRTTABLE	struct	no
IPCTL_MRTDEBUG	integer	yes
IPCTL_IPQ	struct	no
IPCTL_IPINTRQ	struct	no
IPCTL_DYNAMIC_MTU	integer	yes
IPCTL_ADMIN_OVERRIDE_MTU	integer	yes
IPCTL_IPMAXPKTS	integer	yes
IPCTL_IPLOADLEVELING	integer	yes

IPCTL_FORWARDING
Returns a value of 1 when IP forwarding is enabled for the host, meaning that the host is acting as a router.

IPCTL_SENDREDIRECTS
Returns a value of 1 when ICMP redirects may be sent by the host.

IPCTL_STAT
: Returns a copy of the `ipstat` structure. See `/usr/include/netinet/ip_var.h` for greater detail.

IPCTL_QMAXLEN
: Returns the *ipqmaxlen* kernel variable. This is the maximum size of the IP input queue.

IPCTL_SUBNETSARELOCAL
: Returns the value of 1 when the system is treating subnets as local networks.

IPCTL_MRTPROTO
: Returns the *ip_mrtproto* kernel variable. This variable is used by `netstat`(1B) to determine if the kernel is performing multicast routing.

IPCTL_MRTSTAT
: Returns a copy of the `mrtstat` structure. See `/usr/include/netinet/ip_mroute.h` for greater detail.

IPCTL_VIFTABLE
: Returns a copy of the multicast virtual interface table

IPCTL_MRTTABLE
: Returns a copy of the multicast routing tables

IPCTL_MRTDEBUG
: Returns a value of 1 if `mrouted` kernel debugging is enabled

IPCTL_IPQ
: Returns a copy of the IP reassembly queue structure.

IPCTL_IPINTRQ
: Returns a copy of the IP input queue (`ipintrq`)

IPCTL_DYNAMIC_MTU
: Returns a value of 1 if dynamic network `mtu` discovery is enabled

IPCTL_ADMIN_OVERRIDE_MTU
: Returns a value of 1 if the administrator maximum transmission unit (`mtu`) override option is enabled

IPCTL_IPMAXPKTS
: Returns the maximum number of packets `ipintr` will process on a single pseudo interrupt.

IPCTL_IPLOADLEVELING
: Returns a non-zero if IP load leveling is enabled for ATM.

IPPROTO_ICMP

The table and integer information for the IPPROTO_ICMP level is detailed below. The changeable column indicates whether a process with appropriate privilege may change the value.

Fourth level name	Type	Changeable
ICMPCTL_MASKREPL	integer	yes
ICMPCTL_STAT	struct	no

ICMPCTL_MASKREPL

Returns a value of 1 if ICMP network mask requests are to be answered

ICMPCTL_STAT

Returns a copy of the icmpstat structure. See /usr/include/netinet/icmp_var.h for greater detail.

IPPROTO_IGMP

The table and integer information for the IPPROTO_IGMP level is detailed below. The changeable column indicates whether a process with appropriate privilege may change the value.

Fourth level name	Type	Changeable
IGMPCTL_STAT	struct	no

IGMPCTL_STAT

Returns a copy of the igmpstat structure. See usr/include/netinet/igmp_var.h for greater detail.

IPPROTO_TCP

The table and integer information for the IPPROTO_TCP level is detailed below. The changeable column indicates whether a process with appropriate privilege may change the value.

Fourth level name	Type	Changeable
TCPCTL_STAT	struct	no
TCPCTL_PRINTFS	integer	yes
TCPCTL_REXMTTHRESH	integer	yes
TCPCTL_DEFTTL	integer	yes
TCPCTL_SENDSPACE	integer	yes

Fourth level name	Type	Changeable
TCPCTL_RECVSPACE	integer	yes
TCPCTL_KEEPIDLE	integer	yes
TCPCTL_PCB	struct	no
TCPCTL_DEBUG	struct	no
TCPCTL_DEBX	integer	no
TCPCTL_NDEBUG	integer	no
TCPCTL_AUTOWINSHFT	integer	yes

TCPCTL_STAT
: Returns a copy of the tcpstat structure. See /usr/include/netinet/tcp_var.h for greater detail.

TCPCTL_PRINTFS
: Returns a value of 1 if TCP printf debugging is enabled

TCPCTL_REXMTTHRESH
: Returns the value of the retransmission threshold

TCPCTL_DEFTTL
: Returns the default IP time-to-live for TCP sockets

TCPCTL_SENDSPACE
: Returns the default TCP send space for socket output buffering

TCPCTL_RECVSPACE
: Returns the default TCP receive space for socket input buffering

TCPCTL_KEEPIDLE
: Returns the default IP time-to-live for TCP sockets

TCPCTL_PCB
: Returns a copy of the TCP transmission control blocks

TCPCTL_DEBUG
: Returns the TCP trace records when a socket is marked for debugging

TCPCTL_BEBX
: Returns the actual number of TCP trace records collected

TCPCTL_NDEBUG
: Returns the maximum number of TCP trace records that can be collected

TCPCTL_AUTOWINSHFT
Returns a value of 1 if the system is allowed to calculate the windowshift based on the recv socket buffer size

IPPROTO_UDP
The table and integer information for the IPPROTO_UDP level is detailed below. The changeable column indicates whether a process with appropriate privilege may change the value.

Fourth level name	Type	Changeable
UDPCTL_CHECKSUM	integer	yes
UDPCTL_STAT	struct	no
UDPCTL_SENDSPACE	integer	yes
UDPCTL_RECVSPACE	integer	yes
UDPCTL_DEFTTL	integer	yes
UDPCTL_PCB	struct	no

UDPCTL_CHECKSUM
Returns a value of 1 if UDP checksums are being performed

UDPCTL_STAT
Returns a copy of the udpstat structure. See /usr/include/netinet/udp_var.h for greater detail.

UDPCTL_SENDSPACE
Returns the default UDP send space for socket output buffering

UDPCTL_RECVSPACE
Returns the default UDP receive space for socket input buffering

UDPCTL_DEFTTL
Returns the default IP time-to-live for UDP sockets

UDPCTL_PCB
Returns a copy of the UDP transmission control blocks

PF_LINK The third level is an index into the "array" of ifnet structures or a pointer to a specific ifnet structure. The fourth level integer information for the PF_LINK level is detailed below. The changeable column indicates whether a process with appropriate privilege may change the value.

Fourth level name	Type	Changeable
IFCTL_IFQMAXLEN	integer	yes
IFCTL_METRIC	integer	no

Fourth level name	Type	Changeable
IFCTL_MTU	integer	no
IFCTL_FLAGS	integer	no
IFCTL_IF	struct	no
IFCTL_IFADDR	struct	no
IFCTL_BUFSTATLEN	integer	no
IFCTL_BUFSTATSIZE	integer	no
IFCTL_IBUFSTAT	struct	no
IFCTL_OBUFSTAT	struct	no
IFCTL_BBGINFO	struct	no

IFCTL_IFQMAXLEN
: Returns the interface's maximum size of the send queue

IFCTL_METRIC
: Returns the interface's metric value

IFCTL_MTU
: Returns the size of the interface's MTU

IFCTL_FLAGS
: Returns the interface's flag values

IFCTL_IF
: Returns a particular interface's `ifnet` structure. See `/usr/include/net/if.h` for greater detail.

IFCTL_IFADDR
: Returns a particular interface's `ifaddr` structure. See `/usr/include/net/if.h` for greater detail.

IFCTL_BUFSTATLEN
: Returns the `ifc_bufstatlen` kernel variable

IFCTL_BUFSTATSIZE
: Returns the `ifc_bufstatsize` kernel variable.

IF_CTL_IBUFSTAT
: Returns an array of integers that contains statistics about the number of input packets for a given size

IFCTL_OBUFSTAT
: Returns an array of integers that contains statistics about the number of output packets for a given size

IFCTL_BBGINFO
Returns a copy of the bbg_info structure. See /usr/include/crayif/if_bbg.h for greater detail.

PF_ROUTE
Returns the entire routing table or a subset of it. The data is returned as a sequence of routing messages (see route(4P) for the header file, format, and meaning). The length of each message is contained in the message header. The third level name is a protocol number, which is currently always zero. The fourth level name is an address family, which may be set to zero to select all address families. The fifth level names are as follows:

Fifth level name	Type	Changeable
NET_RT_DUMP	struct	no
NET_RT_FLAGS	struct	no
NET_RT_IFLIST	struct	no
NET_RT_STATS	struct	no

NET_RT_DUMP
Returns a copy of the rtentry structures. See /usr/include/net/route.h for greater detail.

NET_RT_FLAGS
Returns a copy of the rtentry structures. See /usr/include/net/route.h for greater detail.

NET_RT_IFLIST
Returns a copy of the routing information from the interface structures (ifnet).

NET_RT_STATS
Returns a copy of the rstat structure. See /usr/include/net.route.h for greater detail.

PF_SOCK The integer information for the PF_SOCK level is detailed below. The changeable column indicates whether a process with appropriate privilege may change the value.

Third level name	Type	Changeable
SOCKCTL_MAXSOCK	integer	yes
SOCKCTL_SBMAX	integer	yes
SOCKCTL_PRINTDELAY	integer	yes

SOCKCTL_MAXSOCK
Returns the maximum number of sockets the system will allow to be opened

SOCKCTL_SBMAX
Returns the system-wide maximum send and receive space for socket buffering. This value is the maximum number of bytes that can be set on the SO_SNDBUF and SORECVBUF socket options.

SOCKCTL_PRINTDELAY
Returns the default minimum interval between operator messages of the same type

PF_TRACE
The table and integer information for the PF_TRACE level is detailed below. The changeable column indicates whether a process with appropriate privilege may change the value.

Third level name	Type	Changeable
TRCTL_TCPWAITQ	integer	no
TRCTL_UDPWAITQ	integer	no
TRCTL_PCB	struct	no
TRCTL_RECVSPACE	integer	yes

TRCTL_TCPWAITQ
Returns the kernel variable tcpwaitq. This value is used by netstat(1B) to display trace information.

TRCTL_UDPWAITQ
Returns the kernel variable udpwaitq. This value is used by netstat(1B) to display trace information.

TRCTL_PCB
Returns a copy of the TRACE transmission control blocks

TRCTL_RECVSPACE
Returns the default TRACE receive space for socket input buffering

PF_UNIX The third level name for the PF_UNIX level is SOCK_STREAM or SOCK_DGRAM. The table and integer information for the PF_UNIX level is detailed below. The changeable column indicates whether a process with appropriate privilege may change the value.

Fourth level name	Type	Changeable
UNPCTL_SENDSPACE	integer	yes
UNPCTL_RECVSPACE	integer	yes
UNPCTL_PCB	struct	no

UNPCTL_SENDSPACE
: Returns the default Unix-domain send space for stream or datagram socket output buffering

UNPCTL_RECVSPACE
: Returns the default Unix-domain receive space for stream or datagram socket input buffering

UNPCTL_PCB
: Returns a copy of the UNIX transmission control blocks

CTL_USER
: The string and integer information available for the CTL_USER level is detailed below. The changeable column shows whether a process with appropriate privilege may change the value.

Second level name	Type	Changeable
USER_BC_BASE_MAX	integer	no
USER_BC_DIM_MAX	integer	no
USER_BC_SCALE_MAX	integer	no
USER_BC_STRING_MAX	integer	no
USER_COLL_WEIGHTS_MAX	integer	no
USER_CS_PATH	string	no
USER_EXPR_NEST_MAX	integer	no
USER_LINE_MAX	integer	no
USER_POSIX2_CHAR_TERM	integer	no
USER_POSIX2_C_BIND	integer	no
USER_POSIX2_C_DEV	integer	no
USER_POSIX2_FORT_DEV	integer	no
USER_POSIX2_FORT_RUN	integer	no
USER_POSIX2_LOCALEDEF	integer	no
USER_POSIX2_SW_DEV	integer	no
USER_POSIX2_UPE	integer	no
USER_POSIX2_VERSION	integer	no
USER_RE_DUP_MAX	integer	no
USER_STREAM_MAX	integer	no

Second level name	Type	Changeable
USER_TZNAME_MAX	integer	no

USER_BC_BASE_MAX
: The maximum ibase/obase values in the bc(1) utility.

USER_BC_DIM_MAX
: The maximum array size in the bc(1) utility.

USER_BC_SCALE_MAX
: The maximum scale value in the bc(1) utility.

USER_BC_STRING_MAX
: The maximum string length in the bc(1) utility.

USER_COLL_WEIGHTS_MAX
: The maximum number of weights that can be assigned to any entry of the LC_COLLATE order keyword in the locale definition file.

USER_CS_PATH
: Return a value for the PATH environment variable that finds all the standard utilities.

USER_EXPR_NEST_MAX
: The maximum number of expressions that can be nested within parenthesis by the expr(1) utility.

USER_LINE_MAX
: The maximum length in bytes of a text-processing utility's input line.

USER_POSIX2_CHAR_TERM
: Return 1 if the system supports at least one terminal type capable of all operations described in POSIX 1003.2, otherwise 0.

USER_POSIX2_C_BIND
: Return 1 if the system's C-language development facilities support the C-Language Bindings Option, otherwise 0.

USER_POSIX2_C_DEV
: Return 1 if the system supports the C-Language Development Utilities Option, otherwise 0.

USER_POSIX2_FORT_DEV
: Return 1 if the system supports the FORTRAN Development Utilities Option, otherwise 0.

USER_POSIX2_FORT_RUN
: Return 1 if the system supports the FORTRAN Runtime Utilities Option, otherwise 0.

USER_POSIX2_LOCALEDEF
: Return 1 if the system supports the creation of locales, otherwise 0.

USER_POSIX2_SW_DEV
: Return 1 if the system supports the Software Development Utilities Option, otherwise 0.

USER_POSIX2_UPE
: Return 1 if the system supports the User Portability Utilities Option, otherwise 0.

USER_POSIX2_VERSION
: The version of POSIX 1003.2 with which the system attempts to comply.

USER_RE_DUP_MAX
: The maximum number of repeated occurrences of a regular expression permitted when using interval notation.

USER_STREAM_MAX
: The minimum maximum number of streams that a process may have open at any one time.

USER_TZNAME_MAX
: The minimum maximum number of types supported for the name of a timezone.

NOTES

If the executable using this library routine is installed with a privilege assignment list (PAL), a user with one of the following active categories is allowed to perform the actions shown:

Active Category	Action
system, secadm, sysadm	Allowed to change the "changeable" MIBs.

If the PRIV_SU configuration option is enabled, the super user is allowed to change the "changeable" MIBs.

ERRORS

The following errors may be reported:

[EFAULT] The buffer *name*, *oldp*, *newp*, or length pointer *oldlenp* contains an invalid address.

[EINVAL] The *name* array is less than two or greater than CTL_MAXNAME.

[EINVAL] A non-null *newp* is given and its specified length in *newlen* is too large or too small.

[ENOMEM] The length pointed to by *oldlenp* is too short to hold the requested value.

[ENOTDIR] The *name* array specifies an intermediate rather than terminal name.

[EOPNOTSUPP] The *name* array specifies a value that is unknown.

[EPERM] An attempt is made to set a read-only value.

[EPERM] A process without appropriate privilege attempts to set a value.

RETURN VALUES

If the call to `sysctl` is successful, the number of bytes copied out is returned. Otherwise –1 is returned and *errno* is set appropriately.

EXAMPLES

Example 1:

To retrieve the standard search path for the system utilities, use the following:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/sysctl.h>
main() {

   int     mib[2];
   size_t  len;
   char    *p;

   mib[0] = CTL_USER;
   mib[1] = USER_CS_PATH;
   if (sysctl(mib, 2, NULL, &len, NULL, 0) < 0)
      perror("sysctl");
   p = (char *)malloc(len);
   if (sysctl(mib, 2, p, &len, NULL, 0) < 0)
      perror("sysctl");

   printf("Standard Search Path = < %s >0, p);
}
```

Example 2:

To retrieve the size of the kernel's networking IP input queue, use the following:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/sysctl.h>
#include <sys/socket.h>
#include <netinet/in.h>
main() {

   int     ipintrq_size, mib[4];
   size_t  len = sizeof(ipintrq_size);

   mib[0] = CTL_NET;
   mib[1] = PF_INET;
   mib[2] = IPPROTO_IP;
   mib[3] = IPCTL_QMAXLEN;
   if (sysctl(mib, 4, &ipintrq_size, &len, NULL, 0) < 0)
      perror("sysctl");
   printf("IP input queue size is %d0, ipintrq_size);
}
```

FILES

<sys/sysctl.h>	Definitions for top level identifiers, second level kernel and hardware identifiers, and user level identifiers
<sys/socket.h>	Definitions for second level network identifiers
<sys/un.h>	Definitions for fourth level UNIX domain identifiers
<sys/tr_pcb.h>	Definitions of fourth level TRACE domain identifiers
<net/if.h>	Definitions of fourth level IF identifiers
<netinet/in.h>	Definitions for third level Internet identifiers and fourth level IP identifiers
<netinet/icmp_var.h>	Definitions for fourth level ICMP identifiers
<netinet/igmp_var.h>	Definitions for fourth level IGMP identifiers
<netinet/tcp_var.h>	Definitions for fourth level TCP identifiers
<netinet/udp_var.h>	Definitions for fourth level UDP identifiers

SEE ALSO

sysctl(8)

NAME

syslog, setloghost, setlogport, openlog, closelog, setlogmask – Controls system log

SYNOPSIS

```
#include <syslog.h>
int syslog (int priority, char *message, ... );
int setloghost (char *host);
int setlogport (int port);
int openlog (char *ident, int logopt, int facility);
int closelog (void);
int setlogmask (int maskpri);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The syslog function arranges to write *message* onto the system log maintained by syslogd(8). The message is tagged with *priority*. The message looks like a printf(3C) string except that %m is replaced by the current error message (collected from errno). Up to five parameters are supported. A trailing newline character is added if needed. This message is read by syslogd(8). It is then written to the system console or log files, or it is forwarded to syslogd on another host as appropriate.

The *priority* arguments are encoded as a *facility* and a *level*. The *facility* describes the part of the system generating the message, as follows:

Facility	Description
LOG_AUTH	Messages generated by the authorization system, such as login(1), su(1), or getty(8).
LOG_DAEMON	Messages generated by system daemons, such as ftpd(8) or route(8).
LOG_KERN	Messages generated by the kernel. These messages cannot be generated by any user processes.
LOG_LOCAL0	Reserved for local use. (Similarly for LOG_LOCAL1 through LOG_LOCAL7.)
LOG_MAIL	Messages generated by the mail system.
LOG_USER	Messages generated by random user processes. If none is specified, this is the default facility identifier.

The *level* is selected from the following ordered list:

Facility	Level
LOG_ALERT	A condition that should be corrected immediately (such as a corrupted system database).
LOG_CRIT	Critical conditions (for example, hard device errors).
LOG_DEBUG	Messages that contain information typically useful only when debugging a program.
LOG_EMERG	A panic condition, which usually is broadcast to all users.
LOG_ERR	Errors.
LOG_INFO	Informational messages.
LOG_NOTICE	Conditions that are not error conditions, but they may need to be handled specially.
LOG_WARNING	Warning messages.

If syslog cannot pass the message to syslogd(8), it tries to write the message on /dev/console if the LOG_CONS option is set (see below).

If the log messages will be sent to another system on the network, call setloghost(*host*). If the system log daemon, on either the local or remote host, is waiting on a port other than the one specified in /etc/services, call setlogport(*port*). You must call both setloghost(*host*) and setlogport(*port*) before you call either openlog or syslog.

If special processing is needed, you can call openlog(3C) to initialize the log file. The *ident* argument is a string that is prepended to every message. The *logopt* argument is a bit field that indicates logging options. Current values for *logopt* are as follows:

Value	Description
LOG_CONS	If openlog cannot send the message to syslog, LOG_CONS forces messages to be written to the console. This option is safe to use in daemon processes that have no controlling terminal because syslog forks before opening the console.
LOG_DELAY	Opens the pipe to syslogd(8) without the O_NDELAY flag.
LOG_NOWAIT	Indicates not to wait for child processes forked to log messages on the console. This option should be used by processes that enable notification of child termination using SIGCHLD, because syslog can otherwise block waiting for a child process whose exit status has already been collected.
LOG_PID	Logs the process ID with each message; this is useful for identifying instantiations of daemons.
LOG_USETCP	Forces messages to be sent to the local syslog daemon by using the TCP/IP socket interface.

The *facility* argument encodes a default facility to be assigned to all messages that do not have an explicit facility encoded, as previously described.

You can use the closelog function to close the log file.

The setlogmask function sets the log priority mask to *maskpri* and returns the previous mask. Calls to syslog with a priority not set in *maskpri* are rejected. The mask for an individual priority *pri* is calculated by macro LOG_MASK(*pri*); the mask for all priorities up to and including *toppri* is given by macro LOG_UPTO(*toppri*). The default allows all priorities to be logged.

NOTES

The named pipe interface, /dev/log, is not available when FORCED_SOCKET configuration option is ON. In this case, the LOG_USETCP option is forced by the syslog routine.

RETURN VALUES

The syslog and openlog functions return 0 after successful completion. Otherwise, -1 is returned, which indicates that an error occurred.

EXAMPLES

The following example shows execution of the syslog function:

```
#include <syslog.h>

syslog(LOG_ALERT, "who: internal error 23")

openlog("ftpd", LOG_PID, LOG_DAEMON)
setlogmask(LOG_UPTO(LOG_ERR))
syslog(LOG_INFO, "Connection from host %d", CallingHost)

setloghost("secure_log_system")
openlog("login", LOG_PID, LOG_AUTH)
syslog(LOG_WARNING, "%s logged in with a null password.", UserName)

syslog(LOG_INFO|LOG_LOCAL2, "error: %m")
```

SEE ALSO

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

log(4) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

ftpd(8), getty(8), route(8), and syslogd(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

system – Passes string to host for execution

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The system function passes the string to which *string* points to the host environment to be executed by a "command processor" (sh(1)) as input, as if *string* had been typed as a command at a terminal. The current process then performs a waitpid(2) system call, and it waits until the shell terminates. The system function then returns the exit status returned by the waitpid(2) system call. Unless the shell was interrupted by a signal, its termination status is contained in the 8 bits higher up from the low-order 8 bits of the value returned by the waitpid(2) system call. The system function ignores the SIGINT and SIGKILL signals, and it blocks the SIGCHLD signal while waiting for the command to terminate.

To inquire whether a command processor exists, use a null pointer for *string*.

RETURN VALUES

If the argument is a null pointer, the system function returns a nonzero value only if a command processor is available. If the argument is not a null pointer, the system function does a vfork(2) system call to create a child process that, in turn, executes /bin/sh to execute *string*. If the vfork(2) or exec(2) system calls fail, system returns –1 and sets errno.

SEE ALSO

errno.h(3C)

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

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

NAME

sys_types.h – Library header for system type definitions

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

TYPES

Header sys/types.h defines the following types. Unless noted as ISO/ANSI, all items are CRI extensions.

Type	Description
word	The integral type that represents a machine word. Equivalent to long on CRI systems.
ulong	Shorthand notation for unsigned long.
uint	Shorthand notation for unsigned int.
ushort	Shorthand notation for unsigned short.
size_t	The unsigned integral type of the result of the sizeof operator. ISO/ANSI.
time_t	Arithmetic type capable of representing time. ISO/ANSI.
clock_t	Arithmetic type capable of representing time. ISO/ANSI.

MACROS

None

FUNCTION DECLARATIONS

None

SEE ALSO

stddef.h(3C), time.h(3C)

NAME

tcgetattr, tcsetattr – Gets or sets terminal attributes

SYNOPSIS

```
#include <termios.h>
int tcgetattr (int fildes, struct termios *termios_p);
int tcsetattr (int fildes, int optional_actions, const struct termios
*termios_p);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX

DESCRIPTION

The tcgetattr function gets the parameters associated with the object referred to by *fildes* and stores them in the termios structure referenced in termios_p.

The tcsetattr function sets the parameters associated with the terminal (unless the necessary support from the underlying hardware is not available) from the termios structure referenced bytermios_p, as follows:

- If *optional_actions* is TCSANOW, the changes occur immediately.
- If *optional_actions* is TCSADRAIN, the change occurs after all output written to *fildes* has been transmitted. This function should be used when changing parameters that affect output.
- If *optional_actions* is TCSAFLUSH, the change occurs after all output written to the object referred to by *fildes* has been transmitted, and all input that has been received but not read is discarded before the change is made.

The symbolic constants for the values of *optional_actions* are defined in header termios.h.

NOTES

This function is allowed from a background process; however, the terminal attributes may subsequently be changed by a foreground process.

SEE ALSO

terminal(3C), cfgetospeed(3C)

termio(4) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

tcgetpgrp, tcsetpgrp – Gets or sets terminal foreground process group ID

SYNOPSIS

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

pid_t tcgetpgrp(int fildes);

int tcsetpgrp(int fildes, pid_t pgid);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX

DESCRIPTION

The tcgetpgrp routine returns the foreground process group ID of the terminal specified by *fildes*. tcgetpgrp is allowed from a process that is a member of a background process group; however, the information may be subsequently changed by a process that is a member of a foreground process group.

The tcsetpgrp routine sets the foreground process group ID of the terminal specified by *fildes* to *pgid*. The file associated with *fildes* must be the controlling terminal of the calling process and the controlling terminal must be currently associated with the session of the calling process. *pgid* must match a process group ID of a process in the same session as the calling process.

MESSAGES

On success, tcgetpgrp returns the process group ID of the foreground process group associated with the specified terminal. Otherwise, it returns –1 and sets errno to indicate the error.

On success, tcsetpgrp returns a value of 0. Otherwise, it returns –1 and sets errno to indicate the error.

These functions fail if one of more of the following is true:

EBADF The *fildes* argument is not a valid file descriptor.

ENOTTY The file associated with *fildes* is not a terminal.

tcgetpgrp also fails if the following is true:

ENOTTY The calling process does not have a controlling terminal, or *fildes* does not refer to the controlling terminal.

`tcsetpgrp` also fails if the following is true:

`EINVAL` *pgid* is not a valid process group ID.

`ENOTTY` The calling process does not have a controlling terminal, or *fildes* does not refer to the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.

`EPERM` *pgid* does not match the process group of an existing process in the same session as the calling process.

SEE ALSO

`terminal`(3C)

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

`termio`(4) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

tcsendbreak, tcdrain, tcflush, tcflow – Performs terminal control functions

SYNOPSIS

```
#include <termios.h>
int tcsendbreak (int fildes, int duration);
int tcdrain (int fildes);
int tcflush (int fildes, int queue_selector);
int tcflow (int fildes, int action);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX

DESCRIPTION

Function `tcsendbreak` transmits a continuous stream of zero-valued bits for a specific duration, if the terminal is using asynchronous serial data transmission. If *duration* is 0, zero-valued bits are transmitted for at least 0.25 seconds, and not more than 0.5 seconds. If *duration* is not 0, zero-valued bits are transmitted for an implementation-defined period of time.

If the terminal is not using asynchronous serial data transmission, it is implementation-defined whether the `tcsendbreak` function sends data to generate a break condition (as defined by the implementation) or returns without taking any action.

The `tcdrain` function waits until all output written to the object referred to by the *fildes* has been transmitted.

The `tcflush` function discards data written to the object referred to by *fildes* but not transmitted, or data received but not read, depending on the value of *queue_selector*, as follows:

- If *queue_selector* is `TCIFLUSH`, `tcflush` flushes data received but not read.
- If *queue_selector* is `TCOFLUSH`, `tcflush` flushes data written but not transmitted.
- If *queue_selector* is `TCIOFLUSH`, `tcflush` flushes both data received but not read, and data written but not transmitted.

The `tcflow` function suspends transmission or reception of data on the object referred to by *fildes*, depending on the value of *action*, as follows:

- If *action* is `TCOOFF`, `tcflow` suspends output.
- If *action* is `TCOON`, `tcflow` restarts suspended output.

- If *action* is `TCIOFF`, the system transmits a `STOP` character, which is intended to cause the terminal device to stop transmitting data to the system.
- If *action* is `TCION`, the system transmits a `START` character, which is intended to cause the terminal device to start transmitting data to the system.

The symbolic constants for the values of *queue_selector* and *action* are defined in `<termios.h>`.

The default on open of a terminal file is that neither its input nor its output is suspended.

NAME

terminal – Introduction to terminal screen functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

This section describes the terminal screen functions. These functions describe a general terminal interface that is provided to control asynchronous communications ports. A more detailed overview of the terminal interface can be found on the termio(4) man page, which also describes an ioctl(2) interface that can be used to access the same functionality. However, the function interface described here is the preferred user interface.

Many of the functions described here have a termios_p argument that is a pointer to a termios structure. This structure contains the following members:

```
tcflag_t   c_iflag;        /* input modes */
tcflag_t   c_oflag;        /* output modes */
tcflag_t   c_cflag;        /* control modes */
tcflag_t   c_lflag;        /* local modes */
cc_t       c_cc[NCCS];     /* control chars */
```

These structure members are described in detail in termio(4).

ASSOCIATED HEADERS

<termios.h>

<X/Xlib.h>

ASSOCIATED FUNCTIONS

cfgetispeed – Gets terminal input baud rates (see cfgetospeed)
cfgetospeed – Gets terminal output baud rates
cfsetispeed – Sets terminal input baud rates (see cfgetospeed)
cfsetospeed – Sets terminal output baud rates (see cfgetospeed)
tcdrain – Waits until all output written has been transmitted (see tcsendbreak)
tcflow – Suspends transmission or reception of data (see tcsendbreak)
tcflush – Discards data written but not transmitted, or data received but not read (see tcsendbreak)
tcgetattr – Gets terminal attributes
tcsendbreak – Sends data to generate a break condition
tcsetattr – Sets terminal attributes (see tcgetattr)
tcgetpgrp – Gets terminal foreground process group ID
tcsetpgrp – Sets terminal foreground process group ID (see tcgetpgrp)
Xlib – C language X Window System interface library (see xlib)

SEE ALSO

`cfgetospeed`(3C) `tcgetattr`(3C), `tcsendbreak`(3C), `tcgetpgrp`(3C) for descriptions of individual terminal screen functions

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

`termio`(4) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014, for a more detailed overview of the terminal interface

NAME

t_exit – Exits multitasking process

SYNOPSIS

```
#include <tfork.h>
void t_exit (int value);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

CRI extension

DESCRIPTION

Function t_exit is used to exit a process created by t_fork(3C). This function frees the stack associated with the process. t_exit does not release any locks held by that task.

The *value* is returned using the wait(2) system call, but, as this will change in future releases, it is not valuable at this time.

SEE ALSO

tfork(3C), tid(3C), tlock(3C)

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

NAME

tfork, t_fork – Creates a multitasking sibling

SYNOPSIS

```
#include <tfork.h>
int t_fork (int nframes);
int tfork (void);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

CRI extension

DESCRIPTION

The t_fork function uses the _tfork(2) system call to create a new multitasking sibling. It allocates a stack for the new sibling, as well as assigning it a task ID. The stack is created with *nframes* frame packages copied from the elder stack. When the new sibling returns from its base stack frame, an implicit t_exit is called.

The tfork macro expands to a call to t_fork with *nframes* set to 1. If #undef is used to remove the macro definition from tfork, the behavior is undefined.

NOTES

Currently, *nframes* must be set to 1.

This method of multitasking does not work correctly with the Flowtrace (see flowtrace(7)) performance tool.

RETURN VALUES

On failure, –1 is returned. On success, the elder sibling is returned a nonzero value, and the younger sibling is returned a 0.

SEE ALSO

t_exit(3C), tid(3C), tlock(3C)

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

flowtrace(7) (available only online) for information about Flowtrace

CF77 Optimization Guide, Cray Research publication SG–3773

NAME

t_id, t_tid, t_gettid – Return task IDs

SYNOPSIS

```
#include <tfork.h>
int t_id (void);
int t_tid (void);
int t_gettid (int pid);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

CRI extension

DESCRIPTION

Function t_id returns the process identification number (PID) of the caller. It is valid only after a t_fork(3C) call has been placed. Because t_id is not a system call, it is much faster than getpid(2).

Function t_tid returns the task identification number (TID) of the called function. Currently, this value is a small integer and can be used as an index into an array for task-specific data. This value should be used instead of the PID to reference the task.

Function t_gettid returns the TID for the process specified by *pid*. Argument *pid* must reference a task within the current multiprocessing group.

RETURN VALUES

These functions return –1 if the task is not the result of a t_fork call.

SEE ALSO

t_exit(3C), tfork(3C), tlock(3C)

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

NAME

`time` – Determines the current calendar time

SYNOPSIS

```
#include <time.h>
time_t time (time_t *timer);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The `time` function determines the current calendar time. The encoding of the value is unspecified.

NOTES

In general, the specific value returned should not be of concern to you; you should use it only for passing to other time functions.

Under UNICOS, `time`(2) is implemented as a system call, but the `time` function is also 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 `time` function returns the implementation's best approximation to the current calendar time. The value `(time_t)-1` is returned if the calendar time is not available. If `timer` is not a null pointer, the return value is also assigned to the object to which it points.

SEE ALSO

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

NAME

`time.h` – Library header for date and time functions

IMPLEMENTATION

All Cray Research systems

TYPES

The types defined in `time.h` are as follows:

Type	Standards	Description
`size_t`	ISO/ANSI	The unsigned integral type of the result of the `sizeof` operator.
`clock_t`	ISO/ANSI	Arithmetic type capable of representing time.
`altzone`	ISO/ANSI	Difference in seconds between universal coordinated time (UTC) and local daylight saving time.
`timezone`	XPG4	Difference, in seconds, between UTC and the local standard time. The function `tzset` uses the contents of the environment variable `TZ` to override the default value of `timezone`.
`daylight`	XPG4	Indicates whether time should reflect daylight savings time. The function `tzset` uses the contents of the environment variable `TZ` to override the default value of daylight.
`tzname`	POSIX	Contains time zone names. The function tzset uses the contents of the environment variable `TZ` to override the default value of `tzname`.
`time_t`	ISO/ANSI	Arithmetic type capable of representing time.
`struct tm`	ISO/ANSI	Structure that holds the components of a calendar time, called the *broken-down time*. The structure contains at least the members shown in the following structure, in any order. The semantics of the members and their normal ranges are expressed in the comments. The value of `tm_isdst` is positive if daylight saving time is in effect, zero if daylight saving time is not in effect, and negative if the information is not available.

The members of structure `struct tm` are as follows:

```
int tm_sec;   /* seconds after the minute [0, 61] */
int tm_min;   /* minutes after the hour [0, 59] */
int tm_hour;  /* hours since midnight [0, 23] */
int tm_mday;  /* day of the month [1, 31] */
int tm_mon;   /* months since January [0, 11] */
int tm_year;  /* years since 1900 */
int tm_wday;  /* days since Sunday [0, 6] */
int tm_yday;  /* days since January 1 [0, 365] */
int tm_isdst; /* Daylight Saving Time flag */
```

MACROS

The macros defined in header `time.h` are as follows:

Macro	Standards	Description
`NULL`	ISO/ANSI	An implementation-defined null pointer constant, equal to zero on CRI systems.
`CLK_TCK`	POSIX	Clock ticks / second
`CLOCKS_PER_SEC`	ISO/ANSI	Number per second of the value returned by the `clock` function. On Cray Research systems, this macro expands to 1,000,000 because the CPU time is reported in microseconds.

FUNCTION DECLARATIONS

Functions declared in header `time.h` are as follows:

```
ascftime      ascftime_r    asctime       cftime        clock
cpused        ctime         ctime_r       difftime      gmtime
gmtime_r      localtime     localtime_r   mktime        rtclock
strftime      strptime      time          tzset
```

SEE ALSO

`ctime`(3C), `strftime`(3C), `strptime`(3C)

NAME

t_lock, t_unlock, t_testlock, t_nlock, t_nunlock – Lock routines for multitasking

SYNOPSIS

```
#include <tfork.h>
t_lock (lock_t *lock);
t_unlock (lock_t *lock);
long t_testlock (lock_t *lock);
t_nlock (nlock_t *lock);
t_nunlock (nlock_t *lock);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

CRI extension

DESCRIPTION

Lock routines protect critical regions of code and shared memory. A *lock* is a statically allocated word in memory. When the word is 0, the lock is clear; when the word is nonzero, the lock is set. The following paragraphs describe each lock routine.

Routine t_lock sets a lock. If the lock is already set when t_lock is called, the task is suspended until another task clears the lock. Calls to t_lock do not nest. The lock word must be 0 before t_lock is called the first time; otherwise the program may crash.

Routine t_unlock clears a lock. There must be one call to t_unlock for every call to t_lock.

Function t_testlock tests a lock and locks it if it is not already locked. t_testlock returns the original value of the lock (0 if the lock was clear, nonzero if the lock was set). Function t_testlock cannot be used on nested locks.

Routine t_nlock sets a nested lock. If the lock is already set when t_nlock is called, the task ID is checked. If this task holds the lock, the nesting level is incremented. If this task does not hold the lock, the task is suspended until the task that holds the lock has cleared it.

Routine t_nunlock clears a nested lock. When t_nunlock is called, the nesting level is decremented. If this is the last active nest level, the lock is cleared. There must be one call to t_nunlock for every call to t_nlock.

SEE ALSO

t_exit(3C), tfork(3C), tid(3C)

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

NAME

tmpfile – Creates a temporary binary file

SYNOPSIS

```
#include <stdio.h>
FILE *tmpfile (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The tmpfile function creates a temporary binary file (using a name generated by tmpnam(3C)) and it returns a corresponding FILE pointer. If the file cannot be opened, a null pointer is returned. The file is removed automatically when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined. In the Cray Research implementation, the file is removed if the program terminates abnormally. The file is opened for update ("wb+").

FORTRAN EXTENSIONS

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

```
INTEGER*8 TMPFILE, I
I = TMPFILE( )
```

SEE ALSO

fopen(3C), mktemp(3C), perror(3C), tmpnam(3C)

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

NAME

tmpnam, tempnam – Creates a name for a temporary file

SYNOPSIS

```
#include <stdio.h>
char *tmpnam (char *s);
char *tempnam (const char *dir, const char *pfx);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (tmpnam only)
XPG4 (tempnam only)

DESCRIPTION

Functions tmpnam and tempnam generate valid file names that are not the same as the name of an existing file and that can be used safely for the name of a temporary file.

The tmpnam function generates a file name by using the path prefix defined as P_tmpdir in the header file stdio.h unless the TMPDIR environment variable is defined. If TMPDIR is provided, and its length is less than L_tmpnam–16, it is used as the path prefix. If *s* is null, tmpnam leaves its result in an internal static area and returns a pointer to that area. The next call to tmpnam may destroy the contents of the area. If *s* is not null, it is assumed to be the address of an array of at least L_tmpnam bytes (L_tmpnam is a constant defined in header file stdio.h); tmpnam places its result in that array and returns *s*.

The tempnam function lets you control the choice of a directory. The *dir* argument points to the name of the directory in which the file will be created. If *dir* is null or points to a string that is not a name for an appropriate directory, the path prefix defined as P_tmpdir in the header file stdio.h is used. If that directory is not accessible, /tmp is used as a last resort. To override this entire sequence, provide a TMPDIR environment variable in the user's environment; the variable's value is the name of the desired temporary-file directory, rather than the directory specified on the call to tempnam. (This use of TMPDIR with the tempnam function is a CRI extension; for XPG4 conformance, the TMPDIR environment variable must not be set.)

Many applications prefer their temporary files to have certain initial letter sequences in their names. Use the *pfx* argument for this. This argument may be null or may point to a string of up to 5 characters to be used as the first few characters of the temporary-file name.

The tempnam function uses malloc(3C) to get space for the constructed file name, and it returns a pointer to this area. Thus, any pointer value returned from tempnam may serve as an argument to free (see malloc(3C)). If tempnam cannot return the expected result for any reason (that is, if malloc failed, or none of the preceding attempts to find an appropriate directory was successful), a null pointer is returned.

NOTES

These functions generate a different file name each time they are called.

Files created using these functions and either fopen(3C) or creat(2) are temporary only in the sense that they reside in a directory intended for temporary use, and their names are unique. To remove the file when its use is ended, you must use remove(3C).

The tmpnam and tempnam functions generate a different string each time they are called, up to TMP_MAX times. If they are called more than TMP_MAX times, these functions start recycling previously used names.

CAUTIONS

Between the time a file name is created and the file is opened, some other process can create a file with the same name. This situation can never happen if that other process is using these functions or mktemp(3C), and the file names are chosen so as to render duplication by other means unlikely.

FORTRAN EXTENSIONS

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

```
CHARACTER dir *m, pfx *n
INTEGER*8 TEMPNAM, I
I = TEMPNAM(dir,pfx)
```

Arguments *dir* and *pfx* may be of type integer. In that case, the data must be packed 8 characters per word and terminated with a null (0) byte.

SEE ALSO

fopen(3C), malloc(3C), mktemp(3C), tmpfile(3C)

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

NAME

`tracebk` – Prints a traceback

SYNOPSIS

```
include <stdlib.h>
int tracebk ([int depth[, FILE *optunit]]);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The `tracebk` function prints a traceback (on `stdout`, or *optunit* if specified) beginning with its caller and ending at "Start-up" or when *depth* is reached. The traceback includes the arguments and their values, and some data about the run-time stack.

The `tracebk` function has the following optional argument:

depth Trace depth; if no argument is used, or *depth* is not in the range 3 to 50, 25 is used.

- <0 Trace depth = 25
- 0 Prints one line "Where am I" message
- 1 Prints one line trace (caller's name and line number)
- 2 Prints "Where am I" message and one line trace
- >2 Trace depth (25 if above 50)

NOTES

A maximum of 12 different active recursive functions may be in the traceback. The `tracebk` function fails (with a message on `stdout`) if the trace data has been detectably corrupted.

RETURN VALUES

The `tracebk` function returns 0 if the traceback can be completed; otherwise, it is 1.

FORTRAN EXTENSIONS

The `tracebk` function can also be called from Fortran, as follows:

```
CALL TRACEBK
```

or

```
INTEGER*8 depth
CALL TRACEBK(depth)
```

or

```
INTEGER*8 depth
CHARACTER*n filenm
CALL TRACEBK(depth, filenm)
```

The optional filenm argument in the call to TRACEBK is a character variable containing the name of a file in which TRACEBK will write the tracebk. The *filenm* argument must not be open as a Fortran file when TRACEBK is called because the results are unpredictable. The *filenm* argument may be read with standard Fortran I/O. The *depth* must be present if *filenm* is present.

SEE ALSO

STKSTAT(3C)

REPRIEVE(3F), TRACEBK(3F) in the

NAME

tsearch, tfind, tdelete, twalk – Manages binary search trees

SYNOPSIS

```
#include <search.h>

void *tsearch (const void *key, void **rootp, int (*compar)(const void *,
const void *));

void *tfind (const void *key, void *const *rootp,
int (*compar)(const void *, const void *));

void *tdelete (const void *key, void **rootp, int (*compar)(const void *,
const void *));

void twalk (const void *root, void (*action)(const void *, VISIT, int));
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

Functions tsearch, tfind, tdelete, and twalk manipulate binary search trees. All comparisons are done with a user-supplied function, which is called with two arguments, the pointers to the elements being compared. It returns an integer less than, equal to, or greater than 0, according to whether the first argument is considered to be less than, equal to, or greater than the second argument. The comparison function does not have to compare every byte, therefore arbitrary data may be contained in the elements in addition to the values being compared.

The tsearch function builds and accesses the tree. *key* is a pointer to data that will be accessed or stored. If data in the tree is equal to **key* (the value to which *key* points), a pointer to this found data is returned; otherwise, **key* is inserted, and a pointer to it is returned. Only pointers are copied; therefore, the calling function must store the data.

The *rootp* argument points to a variable that points to the root of the tree. A null value for the variable to which *rootp* points denotes an empty tree; in this case, the variable is set to point to the data that will be at the root of the new tree.

Like tsearch, tfind searches for data in the tree, returning a pointer to it if found. If it is not found, however, tfind returns a null pointer. The arguments for tfind are the same as for tsearch.

The `tdelete` function deletes a node from a binary search tree. The arguments are the same as for `tsearch`. The variable to which *rootp* points is changed if the deleted node was the root of the tree. The `tdelete` function returns a pointer to the parent of the deleted node, or a null pointer if the node is not found.

The `twalk` function traverses a binary search tree. The *root* argument is the root of the tree to be traversed. (You can use any node in a tree as the root for a walk below that node.) *action* is the name of a function to be invoked at each node. This function, in turn, is called with three arguments. The first argument is a pointer to the node being visited. The second argument is a value from an enumeration data type, as follows:

```
typedef enum { preorder, postorder, endorder, leaf } VISIT;
```

(defined in the header file `search.h`), depending on whether this is the first, second, or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level 0.

The pointers to the key and the root of the tree may be pointers of any type.

The value required should be cast into type pointer-to-element.

CAUTIONS

If the calling function alters the pointer to the root, results are unpredictable.

NOTES

The *root* argument to `twalk` is one level of indirection less than the *rootp* arguments to `tsearch` and `tdelete`.

Two nomenclatures are used to refer to the order in which tree nodes are visited. The `tsearch` function uses preorder, postorder, and endorder, respectively, to refer to visting a node before any of its child processes, after its left child process and before its right, and after both its child processes. The alternate nomenclature uses preorder, inorder, and postorder, respectively, to refer to the same visits, which could result in some confusion over the meaning of postorder.

RETURN VALUES

If not enough space is available to create a new node, `tsearch` returns a null pointer.

If *rootp* is null on entry, `tsearch`, `tfind`, and `tdelete` return a null pointer.

If the data is found, both `tsearch` and `tfind` return a pointer to it. If not, `tfind` returns null, and `tsearch` returns a pointer to the inserted item.

EXAMPLES

The following example reads in strings and stores structures that contains a pointer to each string and a count of its length. It then walks the tree, printing out the stored strings and their lengths in alphabetical order.

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

struct node {              /* pointers to these are stored in tree */
       char *string;
       int length;
};
char string_space[10000];  /* space to store strings */
struct node nodes[500];    /* nodes to store */
void *root = NULL;         /* this points to the root */

main( )
{
   char *strptr = string_space;
   struct node *nodeptr = nodes;
   int i = 0;
   int node_compare(const void *node1, const void *node2);
   void print_node(void *node, VISIT order, int level);
```

```
   while (gets(strptr) != NULL && i++ < 500)  {
      /* set node */
      nodeptr->string = strptr;
      nodeptr->length = strlen(strptr);
      /* put node into the tree */
      (void) tsearch((void *)nodeptr, &root, node_compare);
      /* adjust pointers, so we don't overwrite tree */
      strptr += nodeptr->length + 1;
      nodeptr++;
   }
   twalk(root, print_node);
}
/*
   This function compares two nodes, based on an
   alphabetical ordering of the string field.
*/
int node_compare(const void *node1, const void *node2);
{
   return (strcmp(((struct node *)node1)->string,
                  ((struct node *)node2)->string));
}

/*
   This function prints out a node, the first time
   twalk encounters it.
*/
void print_node(void *node, VISIT order, int level)
{
   if (order == postorder || order == leaf)  {
      (void)printf("string = %20s,  length = %d\n",
            (*(struct node **)node)->string,
            (*(struct node **)node)->length);
   }
}
```

SEE ALSO

bsearch(3C), hsearch(3C), lsearch(3C)

NAME

TSKLIST – Lists the status of each existing task

SYNOPSIS

```
CALL TSKLIST
```

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

TSKLIST lists the status of each existing task, indicating whether the task is running, ready to run, or waiting. TSKLIST also provides useful information about blocked tasks and lists several timing parameters for the tasks and the program as a whole.

NAME

TSKSTART – Initiates a task

SYNOPSIS

CALL TSKSTART(*task-array*, *name* [,*list*])

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

The TSKSTART routine initiates a task. The following is a list of valid arguments for this routine.

Argument **Description**

task-array Each user-created task is represented by an integer task control array, constructed by the user program. Words 1 through 3 contain the following information:

LENGTH Length of the array in words. On SPARC systems, the length must be set to a value of 2 or 3, depending on the optional presence of the task value field. Set the LENGTH field before creating the task. On Cray PVP systems, the length must be set to a value of 2, 3, 5, 7, or 9, depending on the presence of optional arguments.

TASK ID Task identifier assigned by the multitasking library when a task is created. This identifier is unique among active tasks. The multitasking library uses this field for task identification, but the task identifier is of limited use to the user program and must not be modified.

TASK VALUE (optional field)
This field can be set to any value before the task is created. The task value can be used for any purpose. Suggested values include a programmer-generated task name or identifier or a pointer to a task local-storage area. During execution, a task can retrieve this value by using the TSKVALUE(3F) subroutine.

Words 4 through 9 are optional and may be used only on Cray PVP systems. They can be used to specify the initial stack size, the stack increment, and the task priority. Words 4 through 9 must be used in pairs; that is, if a *string* is used in word 4, word 5 must contain an *integer*; if a *string* is used in word 6, word 7 must contain an *integer*; and if a *string* is used in word 8, word 9 must contain an *integer*.

Words 4/5, 6/7, and 8/9 can consist of one of the following *string*s with a corresponding *integer*:

String (words 4, 6, or 8) **Integer (words 5, 7, or 9)**

´STACKSZ´L	Initial stack size in 256-word blocks
´STACKSZW´L	Initial size in words
´STACKIN´L	Stack increment in 256-word blocks
´STACKINW´L	Stack increment in words
´PRIORITY´L	Task priority

If specified, the *string*s must be left-justified and zero-filled.

name External entry point at which task execution begins. Declare this name EXTERNAL in the program or subroutine making the call to TSKSTART. (Fortran does not allow a program unit to use its own name in this parameter.)

list List of arguments being passed to the new task when it is entered. This list can be of any length.

EXAMPLES

```
      PROGRAM MULTI
      INTEGER TASK1ARY(3),TASK2ARY(3)
      EXTERNAL PLLEL
      REAL DATA(40000)
C
C     LOAD DATA ARRAY FROM SOME OUTSIDE SOURCE
C     ...
C
C     CREATE TASK TO EXECUTE FIRST HALF OF THE DATA
C
      TASK1ARY(1)=3
      TASK1ARY(3)='TASK 1'
C
      CALL TSKSTART(TASK1ARY,PLLEL,DATA(1),20000)
C
C     CREATE TASK TO EXECUTE SECOND HALF OF THE DATA
C
      TASK2ARY(1)=3
      TASK2ARY(3)='TASK 2'
C
      CALL TSKSTART(TASK2ARY,PLLEL,DATA(20001),20000)
C     ...
      END
```

The following is an example of a task control array:

```
INTEGER TASKARY(9)
TSKARY(1) = 9
TSKARY(3) = 1
TSKARY(4) = 'STACKSZW'L
TSKARY(5) = 10000
TSKARY(6) = 'STACKINW'L
TSKARY(7) = 1000
TSKARY(8) = 'PRIORITY'L
TSKARY(9) = 45
```

SEE ALSO

TSKVALUE(3F)

NAME

TSKTEST – Returns a value indicating whether the indicated task exists

SYNOPSIS

```
LOGICAL  TSKTEST
return  =  TSKTEST(task-array)
```

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

TSKTEST returns a value that indicates whether the specified task exists.

The following is a list of valid arguments for this routine.

Argument	Description
return	A logical .TRUE. if the indicated task exists. A logical .FALSE. if the task was never created or has completed execution.
task-array	Task control array. TSKTEST and *return* must be declared type LOGICAL in the calling module.

NAME

TSKTUNE – Modifies tuning parameters within the library scheduler

SYNOPSIS

CALL TSKTUNE(*keyword*$_1$, *value*$_1$, *keyword*$_2$, *value*$_2$,...)

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

The multitasking system software design allows you to perform tuning without the need to rebuild libraries or other system software. TSKTUNE, which can be called multiple times within a program, performs the tuning, modifying tuning parameters within the library scheduler. Each of these parameters has a default setting within the library and can be modified at any time to other valid settings.

Each keyword is an integer variable containing an ASCII string, left-justified, blank-filled, and in uppercase. Each value is an integer. The parameters must be specified in pairs, but the pairs can occur in any order. The following subsections describe the legal keywords.

Keyword	Description
DBACTIVE	Deadband for activation or acquisition of logical CPU. This keyword specifies the number of additional tasks that must be readied before an additional logical CPU is required. The default is 0.
DBRELEAS	Deadband for release of logical CPUs. This keyword specifies the number of logical CPUs retained by the job if there are more CPUs than tasks. The default is 1 less than the number of physical CPUs.
HOLDTIME	Number of clock periods to hold a CPU, waiting for tasks to become ready, before releasing it to the operating system. This parameter lets you hold additional logical CPUs in a program while executing a nonmultitasked section of code and to have these CPUs quickly available when the program reenters a multitasked mode.
MAXCPU	Maximum number of logical CPUs allowed for the program. The initial value is set to the number of physical CPUs available on the system. The value of MAXCPU can range from 1 to a site installation parameter, which limits the number of tasks in the system.
PRIORITY	Scheduling priority of macrotasks. Legal values are 0 to 63, with 0 having the lowest priority. The default is 31.
SAMPLE	Number of clock periods between checks of the ready queue. Use this parameter with the HOLDTIME parameter. SAMPLE adjusts the frequency of sampling the ready queue when a logical CPU is waiting for a task to become ready. If the ready queue is sampled too often, excess memory contention may result.
STACKIN	Stack increment in 256-word blocks.
STACKINW	Stack increment in words.

STACKSZ	Initial stack size (on SPARC systems, the total stack size in words). Specified in 256-word blocks. Only the STACKSZ keyword has an effect on SPARC platforms.
STACKSZW	Initial stack size in words. Only the STACKSZW keyword has an effect on SPARC platforms.

NOTES

Because of variability between and during runs, the effects of this routine are not reliably measurable in a batch environment.

EXAMPLES

```
CALL TSKTUNE('DBACTIVE'L,1,'MAXCPU'L,2)

CALL TSKTUNE('HOLDTIME'L,0)

CALL TSKTUNE('MAXCPU'L,1)
```

NAME

TSKVALUE – Retrieves user identifier specified in task control array

SYNOPSIS

CALL TSKVALUE(*return*)

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

TSKVALUE retrieves the user identifier (if any) specified in the task control array used to create the executing task.

return Integer value that was in word 3 of the task control array when the calling task was created. A value of 0 is returned if the task control array length is less than 3 or if the task is the initial task.

EXAMPLES

```
      SUBROUTINE PLLEL(DATA,SIZE)
      REAL DATA(SIZE)
C
C     DETERMINE WHICH OUTPUT FILE TO USE
C
      CALL TSKVALUE(IVALUE)
      IF (IVALUE .EQ. 'TASK 1') THEN
         IUNITNO=3
      ELSEIF (IVALUE .EQ. 'TASK 2') THEN
         IUNITNO=4
      ELSE
         STOP           (!Error condition; do not continue.)
      ENDIF
C     ...
      END
```

NAME

TSKWAIT – Waits for the indicated task to complete execution

SYNOPSIS

CALL TSKWAIT(*task-array*)

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

TSKWAIT waits for the indicated task to complete execution.

task-array Task control array.

EXAMPLES

```
      PROGRAM MULTI
      INTEGER TASK1ARY(3),TASK2ARY(3)
      EXTERNAL PLLEL
      REAL DATA(40000)
C
C     LOAD DATA ARRAY FROM SOME OUTSIDE SOURCE
C     ...
C
C     CREATE TASK TO EXECUTE FIRST HALF OF THE DATA
C
      TASK1ARY(1)=3
      TASK1ARY(3)='TASK 1'
C
      CALL TSKSTART(TASK1ARY,PLLEL,DATA(1),20000)
C
C     CREATE TASK TO EXECUTE SECOND HALF OF THE DATA
C
      TASK2ARY(1)=3
      TASK2ARY(3)='TASK 2'
C
      CALL TSKSTART(TASK2ARY,PLLEL,DATA(20001),20000)
C     ...
C     NOW WAIT FOR BOTH TO FINISH
C
      CALL TSKWAIT(TASK1ARY)
      CALL TSKWAIT(TASK2ARY)
```

```
C
C     AND PERFORM SOME POST-EXECUTION CLEANUP
C     ...
C
      END
```

NAME

ttyname, ttyname_r, isatty – Finds the name of a terminal

SYNOPSIS

```
#include <unistd.h>
char *ttyname (int fildes);
int ttyname_r (int fildes, char *ttyname, size_t namesize);
int isatty (int fildes);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX (ttyname and isatty)
PThreads (ttyname_r)

DESCRIPTION

The ttyname function returns a pointer to a string that contains the null-terminated path name of the terminal device associated with file descriptor *fildes*.

The ttyname_r function provides the same functionality but with an interface that is safe for multitasked applications. It specifies a buffer to store the tty name, and *namesize* specifies this buffer's size. The maximum buffer size is determined with the _SC_TTYNAME_R_SIZE_MAX sysconf parameter.

If *fildes* is associated with a terminal device, isatty returns 1; otherwise, it returns 0.

NOTES

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

RETURN VALUES

If *fildes* does not describe a terminal device in directory /dev, ttyname returns a null pointer.

On success, the ttyname_r function returns 0; otherwise, it returns an error number:

Error	Description
EBADF	The *fildes* argument is not a valid file descriptor.
ENOTTY	The *fildes* argument does not refer to a tty.
ERANGE	The value of *namesize* is smaller than the length of the string to be returned, including the terminating null character.

FILES

/dev/*

NAME

ungetc, ungetwc – Pushes a character back into the input stream

SYNOPSIS

```
#include <stdio.h>
int ungetc (int c, FILE *stream);
#include <wchar.h>
wint_t ungetwc (wint_t wc, FILE *stream);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (ungetc only)
XPG4 (ungetwc only)

DESCRIPTION

The ungetc and ungetwc functions push the character specified by *c* or *wc* (converted to an unsigned char) back onto the input stream to which *stream* points. The pushed-back characters are returned by subsequent reads on that stream in the reverse order of their pushing. ungetwc is used with wide-character arguments. A successful intervening call (with the stream to which *stream* points) to a file positioning function (fseek, fsetpos, or rewind) discards any pushed-back characters for the stream. The external storage corresponding to the stream is unchanged.

One character of pushback is guaranteed. If the ungetc or ungetwc function is called too many times on the same stream without an intervening read or file positioning operation on that stream, the operation may fail.

If the value of *c* equals that of macro EOF or if *wc* equals WEOF, the operation fails and the input stream is unchanged.

A successful call to the ungetc or ungetwc function clears the end-of-file (EOF) indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed-back characters shall be the same as it was before the characters were pushed back. For a text stream, the value of its file position indicator after a successful call to the ungetc or ungetwc function is unspecified until all pushed-back characters are read or discarded. For a binary stream, its file position indicator is decremented by each successful call to the ungetc or ungetwc function; if its value was 0 before a call, it is indeterminate after the call.

The `fseek`, `fflush`, `fsetpos`, and `rewind` functions erase all memory of inserted characters.

RETURN VALUES

If successful, the `ungetc` and `ungetwc` functions return the character pushed back after conversion. Otherwise, `ungetc` returns `EOF` and `ungetwc` returns `WEOF`.

SEE ALSO

`fseek`(3C), `getc`(3C), `setbuf`(3C)

NAME

utilities – Introduction to general utility functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The general utility functions provide various means for converting strings to numbers, for converting numbers to strings, for managing memory, for communicating with the environment, for searching, and so on.

ASSOCIATED HEADERS

```
<malloc.h>
<nlist.h>
<regexp.h>
<search.h>
<stdlib.h>
<syslog.h>
```

ASSOCIATED FUNCTIONS

Communication with the Environment

abort	Generates an abnormal process termination
atabort(3C)	Calls specified function on abnormal process termination
atexit	Calls specified function on normal termination
endusershell	Gets legal user shells (see getusershell)
exit	Terminates a program
getenv	Returns value for environment name
getlogin	Gets login name
getopt	Gets option letter from argument vector
getoptlst	Gets option argument list
getusershell	Gets legal user shells
isatty	Finds the name of a terminal (see ttyname)
logname	Returns the login name of the user
nlimit	Provides an interface to setting or obtaining resource limit values
nlist	Gets entries from name list
putenv	Changes or adds value to the environment
setusershell	Gets legal user shells (see getusershell)
setenv	Sets value of an environment variable
sleep	Suspends execution for a specified interval

system	Passes string to host for execution
ttyname	Finds the name of a terminal
unsetenv	Removes value of an environment variable (see setenv)

Configuration Values

freeconfval	Frees memory allocated for obtaining run-time configuration values
getconfval	Gets configuration value

Database Management

dbmclose	Closes a database (see dbm)
dbminit	Opens a database (see dbm)
delete	Removes a key and its associated contents in a database (see dbm)
fetch	Accesses data stored under a key in a database (see dbm)
firstkey	Returns the first key in a database (see dbm)
nextkey	Returns the next key in the database (see dbm)
store	Places data in a database under a key (see dbm)

Encryption Functions

crypt	Generates DES encryption
encrypt	Generates DES encryption (see crypt)
setkey	Generates DES encryption (see crypt)

Integer Arithmetic Functions

abs	Returns the integer absolute value
div	Computes quotient and remainder
labs	Returns the long integer absolute value (see abs)
ldiv	Computes long integer quotient and remainder (see div)

Memory Management Functions

calloc	Allocates memory space for array (see malloc)
free	Deallocates memory space (see malloc)
mallinfo	Provides memory allocation information (see malloc)
malloc	Allocates memory space
malloc_check	Checks memory arena for corruption (see malloc)
malloc_error	Memory manager error value (see malloc)
malloc_expand	Expands memory block as large as possible without sbreak (see malloc)
malloc_extend	Returns words that could be reallocated in memory without copying (see malloc)
malloc_howbig	Returns size of memory block (see malloc)
malloc_inplace	Reallocates memory without copying, or fails (see malloc)
malloc_isvalid	Checks validity of memory block (see malloc)
malloc_dtrace	Traces calls to the memory manager (see malloc)
malloc_etrace	Traces calls to the memory manager (see malloc)
malloc_space	Returns memory to the system (see malloc)
malloc_limit	Controls free when last memory block in arena is freed (see malloc)

malloc_stats	Prints memory manager statistics (see malloc)
malloc_troff	Traces call to the memory manager (see malloc)
malloc_tron	Traces call to the memory manager (see malloc)
mallopt	Sets various options in the memory manager (see malloc)
realloc	Changes size in memory of a defined object (see malloc)

Multibyte Character Functions

mblen	Finds bytes in multibyte character
mbtowc	Determines code for value in multibyte characters
_pack	Packs 8-bit bytes to/from Cray 64-bit words
_unpack	Unpacks 8-bit bytes to/from Cray 64-bit words (see _pack)
wctomb	Determines bytes needed for multibyte character

Multibyte String Functions

mbstowcs	Converts array of multibyte characters
wcstobmbs	Converts codes in array into multibyte character

Numeric String Conversion Functions

a64l	Converts from base-64 ASCII to long integer
atof	Converts initial part of string to floating-point number (see strtod)
atoi	Converts string to integer (see strtol)
atol	Converts string to long integer (see strtol)
ecvt	Converts a floating-point number to a string
fcvt	Converts a floating-point number to a string rounded for printf (see ecvt)
gcvt	Converts a floating-point number to a string (see ecvt)
l3tol	Converts 3-byte integers to long integers
ltol3	Converts long integers to 3-byte integers (see l3tol)
l64a	Converts from long integer to base-64 ASCII string (see a64l)
strtod	Converts string to double
strtol	Converts string to long int
strtold	Converts string to long double (see strtod)
strtoul	Converts string to unsigned long int (see strtol)

Pseudo-random Sequence Generation Functions

drand48	Generates uniformly distributed pseudo-random numbers over the interval (0.0, 1.0)
erand48	Generates uniformly distributed pseudo-random numbers over the interval (0.0, 1.0) (see drand48)
jrand48	Generates uniformly distributed pseudo-random numbers (over the interval $(-2^{31}, 2^{31})$ (see drand48)
lcong48	Provides initialization entry points for drand48, lrand48, or mrand48 (see drand48)
lrand48	Generates uniformly distributed pseudo-random numbers (see drand48)
mrand48	Generates uniformly distributed pseudo-random numbers (over the interval $(-2^{31}, 2^{31})$ (see drand48)

nrand48	Generates uniformly distributed pseudo-random numbers (see drand48)
rand	Generates pseudo-random integers
seed48	Provides initialization entry points for drand48, lrand48, or mrand48 (see drand48)
srand48	Provides initialization entry points for drand48, lrand48, or mrand48 (see drand48)
srand	Generates seed for new sequence of pseudo-random numbers (see rand)

Regular Expression Functions

regcmp	Compiles and executes a regular expression
regex	Compiles and executes a regular expression (see regcmp)
re_comp	Matches regular expressions
re_exec	Matches regular expressions (see re_comp)

Searching and Sorting Utilities

bsearch	Performs a binary search of an array
hcreate	Allocates space for hash search tables (see hsearch)
hdestroy	Destroys hash search tables (see hsearch)
hsearch	Manages hash search tables
lfind	Performs a linear search and update (see lsearch)
lsearch	Performs a linear search and update
qsort	Performs quicker sort
tdelete	Deletes a node from a binary search tree (see tsearch)
tfind	Searches for datum in binary search trees (see tsearch)
tsearch	Builds and accesses binary search trees
twalk	Traverses binary search trees (see tsearch)

System Log

airlog	Logs messages to system log using syslog
closelog	Controls system log (see syslog)
openlog	Controls system log (see syslog)
setloghost	Controls system log (see syslog)
setlogmask	Controls system log (see syslog)
setlogport	Controls system log (see syslog)
syslog	Controls system log

SEE ALSO

`str_han`(3C) for more string handling functions

`locale`(3C) for more functions related to environment

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

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

NAME

utimes – Sets file times

SYNOPSIS

```
#include <sys/time.h>
int utimes (char *path, struct timeval *times);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The utimes call uses the "accessed" and "updated" times, in that order, from the *times* vector to set the corresponding recorded times for the file specified by *path*.

The caller must be the owner of the file or the super user. The "inode-changed" time of the file is set to the current time.

RETURN VALUES

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

Error Code	Decsription
ENOTDIR	A component of the path prefix is not a directory.
EINVAL	The path name contains a character with the high-order bit set.
ENOENT	The named file *path* does not exist.
EPERM	The calling process is not super user and not the owner of the file.
EACCES	Search permission is denied for a component of the path prefix.
EROFS	The file system containing the file is mounted read-only.
EFAULT	Argument *path* or *times* points outside the process' allocated address space.
EIO	An I/O error occurred during the reading or writing of the affected inode.

SEE ALSO

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

NAME

values.h – Library header for machine-dependent values

IMPLEMENTATION

All Cray Research systems

STANDARDS

AT&T extension

TYPES

None

MACROS

The header values.h defines a set of manifest constants defined for Cray Research processor architectures. The model assumed for integers is binary representation (twos complement), where the sign is represented by the value of the high-order bit. Many of these macros are similar to macros in float.h and limits.h; use of these two headers is preferable to the use of macros in values.h.

There are a number of macros defined in values.h; the most commonly used macros are as follows:

Macro	Definition
_BIAS	Bias to add to exponent to get bit representation
BITSPERBYTE	Number of bits in a byte
_DEXPLEN	The number of bits for the exponent of a type double
DMAXEXP	The maximum exponent of a type double
DMAXPOWTWO	The largest power of two exactly representable as a type double
DMINEXP	The minimum exponent of a type double
DSIGNIF	Number of significant bits in the mantissa of a double-precision, floating-point number
_EXPBASE	The exponent base
_FEXPLEN	The number of bits for the exponent of a float
FMAXEXP	The maximum exponent of a float
FMAXPOWTWO	The largest power of two exactly representable as a float
FMINEXP	The minimum exponent of a float
FSIGNIF	Number of significant bits in the mantissa of a single-precision, floating-point number
HIBITI	Value of a regular integer with only the high-order bit set
HIBITL	Value of a long integer with only the high-order bit set
HIBITS	Value of a short integer with only the high-order bit set
_HIDDENBIT	Value is 1 if high-significance bit of mantissa is implicit
MAXDOUBLE and LN_MAXDOUBLE	Maximum value of a double-precision, floating-point number and its natural logarithm
MAXFLOAT and LN_MAXFLOAT	Maximum value of a single-precision, floating-point number and its natural logarithm

MAXINT	Maximum value of a signed regular integer (usually the same as MAXSHORT or MAXLONG)
MAXLONG	Maximum value of a `signed long` integer
MAXSHORT	Maximum value of a `signed short` integer
MINDOUBLE and LN_MINDOUBLE	Minimum positive value of a double-precision, floating-point number and its natural logarithm
MINFLOAT and LN_MINFLOAT	Minimum positive value of a single-precision, floating-point number and its natural logarithm
M_PI	The best machine representation of *pi*
M_SQRT2	The best machine representation of the square root of 2
NBPD	Number of bytes in a `double`
NBPF	Number of bytes in a `float`
NBPI	Number of bytes in an `int`
NBPL	Number of bytes in a `long`
NBPS	Number of bytes in a `short`
NBPW	Number of bytes in a machine word

FUNCTIONS

None

NOTES

Header `float.h` is included in header `values.h`. Thus, all of the macros defined in header `float.h` are available in addition to the macros listed here. See `float.h`(3C) for a list of macros that are available in header `float.h`.

SEE ALSO

`float.h`(3C), `limits.h`(3C)

NAME

`var_arg` – Introduction to variable argument functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The variable argument functions provide various means for advancing through a list of arguments whose number and types are not known to the called function when it is translated.

While most functions have a fixed number of arguments and each argument has a fixed type, there are some functions that have a fixed number of arguments of specific types followed by a variable number of arguments (zero or more) of unspecified types. A function can be called with a variable number of arguments of varying types; its parameter list contains one or more parameters. The rightmost parameter plays a special role in the access mechanism; it is designated *parmN* in this section.

Two variations on the variable argument functions are provided. One method conforms to the ISO/ANSI standard and the other conforms to the System V Interface standard (SVID).

ASSOCIATED HEADERS

`<stdarg.h>`
`<varargs.h>`

ASSOCIATED FUNCTIONS

None

NAME

varargs.h – Library header for variable arguments

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

TYPES

Header varargs.h defines the following types:

Type	Standards	Description
va_alist	AT&T extension	Used as the parameter list in a function header.
va_list	AT&T extension	Type defined for the variable used to traverse the list.

MACROS

Header varargs.h defines the following macros:

Macro	Standards	Description
va_dcl	AT&T extension	Declaration for va_alist. No semicolon should follow va_dcl.
va_start (*pvar*)	AT&T extension	Called to initialize *pvar* to the beginning of the list.
va_arg (*pvar*,*type*)	AT&T extension	Returns the next argument in the list pointed to by *pvar*. Argument *type* is the type the argument is expected to be. Different types can be mixed, but it is up to the function to know what type of argument is expected, as it cannot be determined at run time.
va_end (*pvar*)	AT&T extension	Cleans up.

Multiple traversals, each bracketed by va_start ... va_end, are possible.

FUNCTION DECLARATIONS

None

NOTES

The preceding describes the SVID approach to variable-length argument list processing. The Cray Standard C (and ISO/ANSI) approach is defined in `stdarg.h`. The two approaches are not compatible; if both headers are included in the same compilation unit, the compiler issues redefinition error messages. The Cray Standard C (and ISO/ANSI) approach defined in `stdarg.h` is the preferred method of variable-length argument list processing.

It is up to the calling function to specify how many arguments there are, because it is not always possible to determine this from the stack frame. In the following example, `execl` is passed a zero pointer to signal the end of the list.

CAUTIONS

It is nonportable to specify a second argument of `char`, `short`, or `float` to `va_arg`, since arguments seen by the called function are not `char`, `short`, or `float`. C converts `char` and `short` arguments to `int` and converts `float` arguments to `double` before passing them to a function. However, the Cray implementation of the C language treats arguments of type `float` and `double` identically.

EXAMPLES

The following example is a possible implementation of system call `execl` (see `exec`(2)) using the macros defined in this header:

```
#include <varargs.h>
#define MAXARGS          100

/*    execl is called in this manner
                execl(file, arg1, arg2, ...,(char *)0);
*/
execl(va_alist)
va_dcl
{
      va_list ap;
      char *file;
      char *args[MAXARGS];
      int argno=0;

      va_start(ap);
      file=va_arg(ap, char *);
      while ((args[argno++]=va_arg(ap, char *)) != (char *)0)
               ;
      va_end(ap);
      return execv(file, args);
}
```

SEE ALSO

stdarg.h(3C)

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

NAME

vprintf, vfprintf, vsprintf, vsnprintf – Prints formatted output of a varargs argument list

SYNOPSIS

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

int vprintf (const char *format, va_list arg);

int vfprintf (FILE *stream, const char *format, va_list arg);

int vsprintf (char *s, const char *format, va_list arg);

int vsnprintf (char * restrict s, const char * restrict format, va_list
arg);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The vprintf function is equivalent to printf, with the variable argument list replaced by *arg*. *arg* is initialized by the va_start macro (and possibly subsequent va_arg calls).

The vfprintf function is equivalent to fprintf, with the variable argument list replaced by *arg*. *arg* is initialized by the va_start macro (and possibly subsequent va_arg calls).

The vsprintf function is equivalent to sprintf, with the variable argument list replaced by *arg*. *arg* is initialized by the va_start macro (and possibly subsequent va_arg calls). If copying occurs between objects that overlap, the behavior is undefined.

The vsnprintf function is equivalent to snprintf, with the variable argument list replaced by *arg*. *arg* is initialized by the va_start macro (and possibly subsequent va_arg calls). If copying occurs between objects that overlap, the behavior is undefined.

None of these functions invokes the va_end macro.

RETURN VALUES

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

The `vsprintf` and `vsnprintf` functions return the number of characters written in the array, not counting the terminating null character.

EXAMPLES

The following code shows the use of `vprintf` to print a `varargs` list:

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

void emit_message(int msg_number, int line_number, const char *msg_text, ...)
{
   va_list arg_ptr;

   va_start(arg_ptr, msg_text);

   if (msg_number < 100)
      fprintf(stderr, "* * * ERROR on line %d:  ", line_number);
   else
      fprintf(stderr, "* * * WARNING on line %d:  ", line_number);

   vfprintf(stderr, msg_text, arg_ptr);
   putc('0, stderr);

   va_end(arg_ptr)'
}  /* emit_message */

#define NUM_ARGS 5

main(int argc, char *argv[])
{
   emit_message(30, _LINE_, "Out of memory.");
   emit_message(120, _LINE_, "Name of executable file is
   emit_message(150, _LINE_, "Number of arguments is %d; should be %d.",
                              argc, NUM_ARGS);
}

sn2024:  cc x.c
sn2024:  a.out
* * * ERROR on line 26:  Out of memory.
* * * WARNING on line 27:  Name of executable file is "a.out"
* * * WARNING on line 28:  Number of arguments is 1; should be 5.
sn2024:
```

SEE ALSO

printf(3C), stdarg.h(3C)

NAME

towupper, towlower – Translates wide-characters

SYNOPSIS

```
#include <wchar.h>
wint_t towupper (wint_t wc);
wint_t towlower (wint_t wc);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The towupper and towlower functions have as domains a type wint_t, the value of which must be a character representable as a wchar_t, and must be a wide-character code corresponding to a valid character in the current locale or the value of WEOF. If the argument of towupper represents a lowercase wide-character code, and there exists a corresponding uppercase wide-character code in the program's locale, the result is the corresponding uppercase wide-character code. If the argument of towlower represents an uppercase wide-character code, and there exists a corresponding lowercase wide-character code in the program's locale, the result is the corresponding lowercase wide-character code. All other arguments in the domain are returned unchanged.

NOTES

The behavior of functions towupper and towlower may be affected by the current locale.

SEE ALSO

conv(3C) getwc(3C), locale.h(3C)

NAME

iswalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, iswxdigit, isphonogram, isideogram, isenglish, isnumber, isspecial, iswctype, wctype – Classifies wide characters

SYNOPSIS

```
#include <wchar.h>
int iswalnum (wint_t wc);
int iswalpha (wint_t wc);
int iswcntrl (wint_t wc);
int iswdigit (wint_t wc);
int iswgraph (wint_t wc);
int iswlower (wint_t wc);
int iswprint (wint_t wc);
int iswpunct (wint_t wc);
int iswspace (wint_t wc);
int iswupper (wint_t wc);
int iswxdigit (wint_t wc);
int isphonogram (wint_t wc);
int isideogram (wint_t wc);
int isenglish (wint_t wc);
int isnumber (wint_t wc);
int isspecial (wint_t wc);
int iswctype (wint_t wc, wctype_t charclass);
wctype_t wctype (const char *charclass);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4 (All except isphonogram, isideogram, isenglish, isnumber, isspecial)
CRI Extension (isphonogram, isideogram, isenglish, isnumber, isspecial only)

DESCRIPTION

For all of these functions, the *wc* argument has a domain of type `wint_t`, the value of which must be a wide character code corresponding to a valid character in the current locale or must equal the value of the macro `WEOF`. If the argument *wc* has any other value, the behavior is undefined.

The `iswalnum` function tests whether *wc* is a wide character representing a character of class `alpha` or `digit` in the program's current locale.

The `iswalpha` function tests whether *wc* is a wide character representing a character of class `alpha` in the program's current locale.

The `iswcntrl` function tests whether *wc* is a wide character representing a character of class `cntrl` in the program's current locale.

The `iswdigit` function tests whether *wc* is a wide character representing a character of class `digit` in the program's current locale.

The `iswgraph` function tests whether *wc* is a wide character representing a character of class `graph` in the program's current locale.

The `iswlower` function tests whether *wc* is a wide character representing a character of class `lower` in the program's current locale.

The `iswprint` function tests whether *wc* is a wide character representing a character of class `print` in the program's current locale.

The `iswpunct` function tests whether *wc* is a wide character representing a character of class `punct` in the program's current locale.

The `iswspace` function tests whether *wc* is a wide character representing a character of class `space` in the program's current locale.

The `iswupper` function tests whether *wc* is a wide character representing a character of class `upper` in the program's current locale.

The `iswxdigit` function tests whether *wc* is a wide character representing a character of class `xdigit` in the program's current locale.

The `isphonogram` function tests whether *wc* is a wide character representing a character of class `phonogram` in the program's current locale.

The `isideogram` function tests whether *wc* is a wide character representing a character of class `ideogram` in the program's current locale.

The `isenglish` function tests whether *wc* is a wide character representing a character of class `english` in the program's current locale.

The `isnumber` function tests whether *wc* is a wide character representing a character of class `number` in the program's current locale.

The `isspecial` function tests whether *wc* is a wide character representing a character of class `special` in the program's current locale.

The `iswctype` function determines whether the wide character *wc* has the character class *charclass*, returning true or false.

The `wctype` function is defined for valid character class names as defined in the current locale. The *charclass* is a string identifying a generic character class for which codeset-specific type information is required. The following character class names are defined in all locales - "alnum", "alpha", "blank", "cntrl", "digit", "graph", "lower", "print", "punct", "space", "upper", "xdigit". Additional character class names defined in the locale definition file can also be specified. The function returns a value of type `wctype_t`, which can be used as the second argument to subsequent calls of `iswtype`. The values returned by `wctype` are valid until a call to `setlocale` that modifies the category `LC_CTYPE`.

RETURN VALUES

The `is*` functions return nonzero if the value of the argument conforms to that in the description of the function, and zero otherwise. The `wctype` function returns zero if the given character class name is not valid for the current locale; otherwise, it returns an object of type `wctype_t` that can be used in calls to `iswctype`.

SEE ALSO

`ctype`(3C), `locale.h`(3C), `localedef`(1)

NAME

wcwidth – Number of column positions of a wide-character code

SYNOPSIS

```
#include <wchar.h>
int wcwidth(wint_t wc);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The wcwidth function determines the number of column positions required for the wide character *wc*. The value of *wc* must be a character representable as a wchar_t, and must be a wide-character code corresponding to a valid character in the current locale.

RETURN VALUES

The wcwidth function either returns zero (if *wc* is a null wide-character code), or returns the number of column positions to be occupied by the wide-character code *wc*, or returns –1 (if *wc* does not correspond to a printing wide-character code).

SEE ALSO

conv(3C) getwc(3C), wctype(3C),

NAME

wordexp, wordfree – Performs word expansions

SYNOPSIS

```
#include <wordexp.h>
int wordexp (const char *words, wordexp_t *pwordexp, int flags);
void wordfree (wordexp_t *pwordexp);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The wordexp function performs word expansions and places the list of expanded words into *pwordexp*. The expansions are the same as the shell would perform on *words* when used as an argument to a utility command. Therefore, the newline character and shell special characters can appear in *words* only when quoted, or when used in command substitution. An unquoted # symbol at the beginning of a token is treated as a comment character.

The header file wordexp.h defines the wordexp_t structure type, which includes at least the following members:

Type	Field	Description
size_t	we_wordc	Count of words matched by *words*
char**	we_wordv	Pointer to list of expanded words
size_t	we_offs	Slots to reserve at the beginning of we_wordv

The *words* argument is a pointer to a string that contains one or more words to be expanded. The wordexp function stores the number of generated words to we_wordc and stores a pointer to a list of word pointers in we_wordv. Each field created during field splitting or path name expansion is a separate word in the we_wordv list. The first pointer after the last pointer is null.

The caller creates the structure to which pwordexp points. The wordexp function allocates other space as needed, including memory to which we_wordv points. The wordfree function frees memory associated with pwordexp from a previous call.

The *flags* argument controls the behavior of wordexp. The value of *flags* is the bitwise inclusive OR of any of the following constants:

Constant	Description
WRDE_APPEND	Appends words generated to those from a previous wordexp.
WRDE_DOOFFS	Checks the we_offs flag; if it is set, it specifies how many null pointers to prepend to we_wordv; thus, w_wordv points to we_offs null pointers, followed by we_wordc word pointers, followed by a null pointer.
WRDE_NOCMD	Fails if command subsitution is requested.
WRDE_REUSE	The *pwordexp* argument was passed to a previous successful call to wordexp and has not been passed to wordfree. The result is the same as if the application had called wordfree and then called wordexp without WRDE_REUSE.
WRDE_SHOWERR	Does not redirect stderr to /dev/null.
WRDE_UNDEF	Reports error on an attempt to expand an undefined shell variable.

You can use the WRDE_APPEND flag to append a new set of words to those generated by a previous call to wordexp. The following rules apply when two or more calls to wordexp are made with the same value of pwordexp and without intervening calls to wordfree:

1. The first such call does not set WRDE_APPEND. All subsequent calls set it.
2. All of the calls treat WRDE_DOOFFS the same, either setting or not setting it.
3. After the second and later calls, we_wordv points to a list that contains the following:
   a. Zero or more nulls, specified by WRDE_DOOFFS and we_offs.
   b. Pointers to the words that were in the we_wordv list before the call, in the same order as before.
   c. Pointers to the new words generated by the latest call, in the specified order.
4. The count returned in we_wordc is the total number of words from all of the calls.

After a call to wordexp, the application can change any field shown at the beginning of this DESCRIPTION section, but if it does, it must reset the field to its original value before a subsequent call, using the same *pwordexp* value, to wordfree or wordexp with the WRDE_APPEND or WRDE_REUSE flag.

If *words* contains an unquoted |, &, ;, <, >, (,), {, }, or newline character in an inappropriate context, wordexp fails and returns zero words.

Unless WRDE_SHOWERR is set in *flags*, wordexp redirects stderr to /dev/null for any utilities executed as a result of command substitution while expanding *words*. If WRDE_SHOWERR is set, wordexp writes messages to stderr concerning any syntax errors detected during the expansion of *words*.

If WRDE_DOOFFS is set, we_offs has the same value for each wordexp call and the wordfree call using a given *pglob*.

RETURN VALUES

If no errors are encountered during word expansion, `wordexp` returns 0; otherwise, it returns a nonzero value. If it terminates due to an error, it returns one of the following values, defined in the header file `wordexp.h`:

Constant	Description
`WRDE_BADCHAR`	An unquoted `|`, `&`, `;`, `<`, `>`, `(`, `)`, `{`, `}`, or newline character appears in an inappropriate context in *words.*
`WRDE_BADVAL`	Reference to undefined shell variable when `WRDE_UNDEF` is set in *flags*.
`WRDE_CMDSUB`	Command substitution requested when `WRDE_NOCMD` was set in *flags*.
`WRDE_NOSPACE`	Attempt to allocate memory failed.
`WRDE_SYNTAX`	Shell syntax error, such as unbalanaced parentheses or unterminated string.

If `wordexp` returns the value `WRDE_NOSPACE`, then the `we_wordc` and `we_wordv` members of the structure to which *pwordexp* points are updated to reflect any words that were successfully expanded. Otherwise, they are left unchanged.

NAME

wcscat, wcsncat, wcscmp, wcsncmp, wcscpy, wcsncpy, wcslen, wcschr, wcsrchr, wcspbrk, wcsspn, wcscspn, wcstok, wcswcs, wcscoll, wcsxfrm – Performs wide-character string operations

SYNOPSIS

```
#include <wchar.h>
wchar_t *wcscat (wchar_t *ws1, const wchar_t *ws2);
wchar_t *wcsncat (wchar_t *ws1, const wchar_t *ws2, size_t n);
int wcscmp (const wchar_t *ws1, const wchar_t *ws2);
int wcsncmp (const wchar_t *ws1, const wchar_t *ws2, size_t n);
wchar_t *wcscpy (wchar_t *ws1, const wchar_t *ws2);
wchar_t *wcsncpy (wchar_t *ws1, const wchar_t *ws2, size_t n);
size_t wcslen (const wchar_t *ws);
wchar_t *wcschr (const wchar_t *ws, wint_t wc);
wchar_t *wcsrchr (const wchar_t *ws, wint_t wc);
wchar_t *wcspbrk (const wchar_t *ws1, const wchar_t *ws2);
size_t wcsspn (const wchar_t *ws1, const wchar_t *ws2);
size_t wcscspn (const wchar_t *ws1, const wchar_t *ws2);
wchar_t *wcstok (wchar_t *ws1, const wchar_t *ws2);
wchar_t *wcswcs (const wchar_t *ws1, const wchar_t *ws2);
int wcscoll (const wchar_t *ws1, const wchar_t *ws2);
size_t wcsxfrm (wchar_t *ws1, const wchar_t *ws2, size_t n);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

With any of the wide character string functions, if copying takes place between objects that overlap, the behavior is undefined.

The `wcscat` function appends a copy of the wide character string pointed to by *ws2* (including the terminating null wide-character) to the end of the wide character string pointed to by *ws1*. The `wcsncat` function appends not more than *n* wide-characters (a null wide-character and wide-characters that follow it are not appended) from the wide character string pointed to by *ws2* to the end of the wide character string pointed to by *ws1*. With both functions, the initial wide-character of *ws2* overwrites the null wide-character at the end of *ws1*. Function `wcsncat` always appends a terminating null wide-character to the result.

The `wcscmp` function returns an integer that is greater than, equal to, or less than 0, according to whether the wide character string pointed to by *ws1* is greater than, equal to, or less than the wide character string pointed to by *ws2*. The `wcsncmp` function compares not more than *n* wide-characters (wide-characters that follow a null wide-character are not compared) from the wide character string pointed to by *ws1* to the wide character string pointed to by *ws2*.

The `wcscpy` function copies the wide character string pointed to by *ws2* (including the terminating null wide-character) into the array pointed to by *ws1*. The `wcsncpy` function copies not more than *n* wide characters (wide-characters that follow a null wide-character are not copied) from the array pointed to by *ws2* to the array pointed to by *ws1*. If the array pointed to by *ws2* is a string that is shorter than *n* wide characters, null wide-characters are appended to the copy in the array pointed to by *ws1*, until *n* wide characters in all have been written.

The `wcslen` function returns the number of wide-characters in the wide character string pointed to by *s*, not including the terminating null wide-character.

The `wcschr` function locates the first occurrence of *wc* in the wide character string pointed to by *ws*. The `wcsrchr` function locates the last occurrence of *wc* in the wide character string pointed to by *ws*. With both functions, the terminating null wide-character is considered to be part of the string.

The `wcspbrk` function locates the first occurrence in the wide character string pointed to by *ws1* of any wide-character from the wide character string pointed to by *ws2*.

The `wcsspn` function computes the length of the maximum initial segment of the wide character string pointed to by *ws1* that consists entirely of wide characters from the wide character string pointed to by *ws2*. The `wcscspn` function computes the length of the maximum initial segment of the wide character string pointed to by *ws1* that consists entirely of wide characters *not* from the wide character string pointed to by *ws2*.

A sequence of calls to the `wcstok` function breaks the wide character string pointed to by *ws1* into a sequence of tokens, each of which is delimited by a wide character from the wide character string pointed to by *ws2*. The first call in the sequence has *ws1* as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by *ws2* may be different from call to call. The first call in the sequence searches the wide character string pointed to by *ws1* for the first wide character that is *not* contained in the current separator string pointed to by *ws2*. If no such wide character is found, then there are no tokens in the wide character string pointed to by *ws1*, and the `wcstok` function returns a null pointer. If such a wide character is found, it is the start of the first token.

The wcstok function then searches for a wide character that *is* contained in the current separator string (*ws2*). If no such wide character is found, the current token extends to the end of the wide character string pointed to by *ws1*, and subsequent searches for a token will return a null pointer. If such a wide character is found, it is overwritten by a null wide character, which terminates the current token. The wcstok function saves a pointer to the following wide character, from which the next search for a token starts. Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as previously described.

The wcswcs function locates the first occurrence in the wide character string pointed to by *ws1* of the sequence of wide characters (excluding the terminating null wide character) in the wide character string pointed to by *ws2*.

The wcscoll function compares the wide character string pointed to by *ws1* to the wide character string pointed to by *ws2*, both interpreted as appropriate to the LC_COLLATE category of the current locale.

The wcsxfrm function transforms the wide character string pointed to by *ws2* and places the resulting wide character string into the array pointed to by *ws1*. The transformation is such that if the wcscmp function is applied to two transformed strings, it returns a value greater than, equal to, or less than 0, corresponding to the result of the wcscoll function applied to the same two original wide character strings. No more than *n* wide characters, including the terminating null wide character, are placed into the resulting array pointed to by *ws1*. If *n* is zero, *ws1* is permitted to be a null pointer.

RETURN VALUES

The wcscat, wcsncat, wcscpy, wcsncpy functions return the value of *ws1*.

The wcscmp function returns an integer that is greater than, equal to, or less than 0, according to whether the wide character string pointed to by *ws1* is greater than, equal to, or less than the wide character string pointed to by *ws2*.

The wcsncmp function returns an integer that is greater than, equal to, or less than 0, according to whether the possibly null-terminated array pointed to by *ws1* is greater than, equal to, or less than the possibly null-terminated array pointed to by *ws2*.

The wcslen function returns the number of wide characters that precede the terminating null wide character.

The wcschr and wcsrchr functions return a pointer to the located wide character, or a null pointer if *wc* does not occur in the wide character string.

The wcspbrk function returns a pointer to the wide character, or a null pointer if no wide character from *ws2* occurs in *ws1*.

The wcsspn function returns the length of the maximum initial segment of the wide character string pointed to by *ws1* that consists entirely of wide characters from the wide character string pointed to by *ws2*. The wcscspn function returns the length of the maximum initial segment of the wide character string pointed to by *ws1* that consists entirely of wide characters *not* from the wide character string pointed to by *ws2*.

The `wcstok` function returns a pointer to the first wide character of a token, or a null pointer if there is no token.

The `wcscoll` function returns an integer that is greater than, equal to, or less than 0, according to whether the wide character string pointed to by *ws1* is greater than, equal to, or less than the wide character string pointed to by *ws2*, when both are interpreted as appropriate to the current locale.

The `wcswcs` function returns a pointer to the located wide character string, or a null pointer if the wide character string is not found. If *ws2* points to a wide character string with zero length, the function returns *ws1*.

The `wcsxfrm` function returns the length of the transformed wide character string (not including the terminating null wide character). If the value returned is *n* or more, the contents of the array pointed to by *ws1* are indeterminate.

SEE ALSO

`locale`(3C), `string`(3C)

NAME

xdr – Achieves machine-independent data transformation

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The external data representation (XDR) functions allow C programmers to describe arbitrary data structures in a machine-independent fashion. These functions transmit data for remote procedure calls.

The XDR library functions are described in the following list:

Function	Description
xdr_array	Translates arrays to/from external representation
xdr_bool	Translates Boolean values to/from external representation
xdr_bytes	Translates counted byte strings to/from external representation
xdr_char	Translates characters to/from external representation
xdr_destroy	Destroys an XDR stream and frees the associated memory
xdr_double	Translates double-precision values to/from external representation
xdr_enum	Translates enumerated types to/from external representation
xdr_float	Translates floating-point values to/from external representation
xdr_getpos	Returns the current position in an XDR stream
xdr_inline	Invokes the in-line functions associated with an XDR stream
xdr_int	Translates integers to/from external representation
xdr_long	Translates long integers to/from external representation
xdr_opaque	Translates fixed-size opaque data to/from external representation
xdr_pointer	Translates pointers to/from external representation
xdr_reference	Follows pointers within structures
xdr_setpos	Changes current position in XDR stream
xdr_short	Translates short integers to/from external representation
xdr_string	Translates counted strings to/from external representation
xdr_u_char	Translates unsigned characters to/from external representation
xdr_u_int	Translates unsigned integers to/from external representation
xdr_u_long	Translates unsigned long integers to/from external representation
xdr_u_short	Translates unsigned short integers to/from external representation
xdr_union	Translates discriminated unions to/from external representation
xdr_vector	Translates fixed-length arrays to/from external representation
xdr_void	Always returns one (1)
xdr_wrapstring	Translates null-terminated strings to/from external representation

`xdrmem_create`	Initializes an XDR stream
`xdrrec_create`	Initializes an XDR stream with record boundaries
`xdrrec_endofrecord`	Marks an XDR record stream with an end-of-record marker
`xdrrec_eof`	Marks an XDR record stream with an end-of-file marker
`xdrrec_skiprecord`	Skips the remaining record in an XDR record stream
`xdrstdio_create`	Initializes an XDR stream as a standard I/O file stream

FILES

`/lib/libc.a`

SEE ALSO

`rpc`(3C)

Remote Procedure Call (RPC) Reference Manual, Cray Research publication SR–2089

"External Data Representation Protocol Specification" in Networking on the Sun Workstation, part #800–1324–03, Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94043.

XDR: External Data Representation Standard, RFC 1014

NAME

`Xlib` – C language X Window System interface library

SYNOPSIS

```
#include <X/Xlib.h>
```

IMPLEMENTATION

All Cray Research systems (except Cray MPP systems)

STANDARDS

Other (see the following description)

DESCRIPTION

This library is the low-level C interface to the X protocol, which supports the X Window System, Version 11, Release 4, from M.I.T.

SEE ALSO

UNICOS X Window System Reference Manual, Cray Research publication SR–2101

NAME

XSELFADD, XCRITADD – Allows performance of *xvar* = *xvar*+*xvalue* under the protection of a hardware semaphore

SYNOPSIS

var = XSELFADD(*xvar*, *xvalue*)

CALL XCRITADD(*xvar*, *xvalue)*

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

XSELFADD is a function, and XCRITADD is a routine.

The following is a list of valid arguments:

Argument	**Description**
xvar	Real variable to be incremented by *xvalue.*
xvalue	Real variable by which *xvalue* is incremented.

A call to XSELFADD is functionally equivalent to, but considerably faster than, the following code block:

```
CALL LOCKON(lockvar)
var = xvar
xvar = xvar+xvalue
CALL LOCKOFF(lockvar)
```

A call to XCRITADD is functionally equivalent to, but considerably faster than, the following code block:

```
CALL LOCKON (lockvar)
xvar = xvar+xvalue
CALL LOCKOFF (lockvar)
```

SEE ALSO

ISELFADD(3F)

NAME

XSELFMUL, XCRITMUL – Allows performance of *xvar* = *xvar**XVALUE under the protection of a hardware semaphore

SYNOPSIS

var = XSELFMUL(*xvar*, *xvalue*)

CALL XCRITMUL(*xvar*, *xvalue*)

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

XSELFMUL is a function, and XCRITMUL is a routine.

The following is a list of valid arguments.

Argument	Description
xvar	Real variable to be multiplied by *xvalue*.
xvalue	Real variable multiplied by *xvar*.

A call to XSELFMUL is functionally equivalent to, but considerably faster than, the following code block:

```
CALL LOCKON(lockvar)

var = xvar

xvar = xvar*xvalue

CALL LOCKOFF(lockvar)
```

A call to XCRITMUL is functionally equivalent to, but considerably faster than, the following code block:

```
CALL LOCKON(lockvar)

xvar = xvar*xvalue

CALL LOCKOFF(lockvar)
```

SEE ALSO

ISELFMUL(3F)

NAME

yp_get_default_domain, yp_bind, yp_unbind, yp_match, yp_first, yp_next, yp_all, yp_order, yp_master, yperr_string, ypprot_err – Network information service (NIS) client interface

SYNOPSIS

```
#include <rpcsvc/ypclnt.h>

int yp_bind (char *indomain);

void yp_unbind (char *indomain);

int yp_get_default_domain (char *outdomain);

int yp_match (char *indomain, char *inmap, char *inkey, int inkeylen,
char **outval, int *outvallen);

int yp_first (char *indomain, char *inmap, char **outkey, int *outkeylen,
char **outval, int *outvallen);

int yp_next (char *indomain, char *inmap, char *inkey, int inkeylen,
char **outkey, int *outkeylen, char **outval, int *outvallen);

int yp_all (char *indomain, char *inmap, struct ypall_callback incallback);

int yp_order (char *indomain, char *inmap, int *outorder);

int yp_master (char *indomain, char *inmap, char **outname);

char *yperr_string (int incode);

int ypprot_err (unsigned int incode);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

This package of functions, formerly known as yellow pages (YP), provides an interface to the network information service (NIS) network look-up service. The package can be loaded from the standard library, /lib/libc.a. See "UNICOS Network Information Service" in *UNICOS Networking Facilities Administrator's Guide*, Cray Research publication SG–2304, and ypfiles(5) and ypserv(8) for an overview of the network information service, including the definitions of map and domain, and a description of the various servers, databases, and commands that comprise the network information service.

The names of all input parameters begin with `in`. Names of output parameters begin with `out`. Output parameters of type `char **` should be addresses of uninitialized character pointers. The NIS client package uses `malloc`(3C) to allocate memory, and it may be freed if the user code has no continuing need for it. For each *outkey* and *outval* allocated, there are 2 extra bytes of memory that contain `NEWLINE` and `NULL`, respectively. These 2 bytes, however, are not reflected in *outkeylen* or *outvallen*. The *indomain* and *inmap* strings must be nonnull and null-terminated. String parameters that are accompanied by a count parameter cannot be null, but can point to null strings; the *count* argument indicates this. Counted strings need not be null-terminated.

All functions of type `int` in this package return 0 if they succeed; otherwise, they return a failure code (`YPERR_` *xxxx*). The MESSAGES section describes possible error codes.

The NIS look-up calls require, at minimum, a map name and a domain name. It is assumed that the client process knows the name of the map of interest. Client processes should fetch the node's default domain by calling `yp_get_default_domain()` and use the returned *outdomain* value as the *indomain* argument to successive NIS calls.

To use the NIS services, the client process must be bound to a NIS server that serves the appropriate domain; this is accomplished by a call to `yp_bind`. Binding need not be done explicitly by user code; rather it is done automatically whenever a NIS look-up function is called. When NIS services are unavailable, the `yp_bind` function can be called directly for processes that make use of a back-up strategy (for example, a local file)

Each binding allocates (uses up) one client process socket descriptor; each bound domain requires one socket descriptor. However, multiple requests to the same domain use the same descriptor. The `yp_unbind`() function is available at the client interface for processes that explicitly manage their socket descriptors while accessing multiple domains. The call to `yp_unbind`() makes the domain unbound, and it frees all per-process and per-node resources used to bind it.

If an RPC failure results upon use of a binding, the associated domain is unbound automatically. At that point, the `ypclnt` layer retries forever or until the operation succeeds, If `yp_bind` is running, and either the client process cannot bind a server for the proper domain or RPC requests to the server fail.

If an error is not related to RPC, or if `yp_bind` is not running, or if a bound `ypserv`(8) process returns any answer (success or failure), the `ypclnt` layer returns control to the user code, with either an error code or a success code and any results.

The `yp_match` function returns the value associated with a passed key. This key must provide an exact match; no pattern matching is available.

The `yp_first` function returns the first key-value pair from the specified map in the specified domain.

The `yp_next`() function returns the next key-value pair in a specified map. The *inkey* argument should be the *outkey* returned from an initial call to `yp_first`() (to get the second key-value pair) or the one returned from the *n*th call to `yp_next`() (to get the *n*th + next key-value pair).

The concept of *first* (and, for that matter, of *next*) is specific to the structure of the NIS map being processed; the retrieval order is not related to either the lexical order within any original (non-NIS) database, or to any obvious numerical sorting order on the keys, values, or key-value pairs. The only ordering guarantee made is that if the `yp_first`() function is called on a particular map, and then the `yp_next`() function is repeatedly called on the same map at the same server until the call fails with a reason of `YPERR_NOMORE`, every entry in the database will be seen exactly once. Further, if the same sequence of operations is performed on the same map at the same server, the entries will be seen in the same order.

Under conditions of heavy server load or server failure, the domain may become unbound, then bound once again (perhaps to a different server), while a client is running. This can cause a break in one of the enumeration rules; specific entries may either be seen twice by the client or not seen at all. This approach protects the client from error messages that would otherwise be returned in the midst of the enumeration. The next paragraph describes a better solution to enumerating all entries in a map.

The `yp_all` function provides a way to transfer an entire map from server to client in a request by using TCP (rather than UDP, as with other functions in this package). The entire transaction occurs as a single RPC request and response. You can use `yp_all` just like any other NIS procedure: identify the map in the normal manner, and supply the name of a function that will be called to process each key-value pair within the map. You return from the call to `yp_all` only when the transaction is completed (successfully or unsuccessfully), or your `foreach` function decides that it does not want to see any more key-value pairs.

The third parameter to `yp_all` is as follows:

```
struct ypall_callback *incallback {
        int (*foreach)();
        char *data;
};
```

The function `foreach` is called as follows:

```
foreach (int instatus, char *inkey, int inkeylen,
         char *inval, int invallen, char *indata);
```

The *instatus* argument holds one of the return status values defined in header file `rpcsvc/yp_prot.h`, either `YP_TRUE` or an error code. (The `ypprot_err` function, described later in this entry, converts an NIS protocol error code to a `ypclnt` layer error code.)

The key and value parameters are somewhat different from the definition in the SYNOPSIS section. First, the memory to which the *inkey* and *inval* arguments point is private to the `yp_all` function, and it is overwritten with the arrival of each new key-value pair. The `foreach` function must do something useful with the contents of that memory, but it does not own the memory itself. Key and value objects presented to the `foreach` function look exactly as they do in the server's map; if they were not terminated by a new line or null-terminated in the map, they are not here either.

The *indata* argument is the contents of the `incallback->data` element passed to `yp_all`. The `data` member of the `yp_callback` structure can be used to share state information between the `foreach` function and the main-line code. Its use is optional, and no part of the NIS client package inspects its contents; you can cast it to something useful or ignore it.

The `foreach` function is a Boolean. It should return 0 to indicate that it wants to be called again for further received key-value pairs, or a nonzero value to stop the flow of key-value pairs. If `foreach` returns a nonzero value, it is not called again; the functional value of `yp_all` is then 0.

The `yp_order` function returns the order number for a map.

The `yp_master` function returns the machine name of the master NIS server for a map.

The `yperr_string` function returns a pointer to an error message string that is null-terminated, but it contains no period or new line.

The `ypprot_err` function takes an NIS protocol error code as input and returns a `ypclnt` layer error code, which can be used, in turn, as input to `yperr_string`.

MESSAGES

All integer functions return 0 if the requested operation is successful, or one of the following errors if the operation fails:

Error Code	Description
YPERR_BADARGS	The function's arguments are bad.
YPERR_RPC	RPC failure; the domain has been unbound.
YPERR_DOMAIN	You cannot bind to the server on this domain.
YPERR_MAP	No such map is in the server's domain.
YPERR_KEY	No such key is in the map.
YPERR_YPERR	An internal NIS server or client error occurred.
YPERR_RESRC	A resource allocation failure occurred.
YPERR_NOMORE	No more records are in the map database.
YPERR_PMAP	You cannot communicate with the portmapper.
YPERR_YPBIND	You cannot communicate with `ypbind`.
YPERR_YPSERV	You cannot communicate with `ypserv`.
YPERR_NODOM	Local domain name has not been set.

FILES

`/usr/include/rpcsvc/ypclnt.h`	Header file for the `ypclnt` interface
`/usr/include/rpcsvc/yp_prot.h`	Header file that holds return status values

SEE ALSO

`malloc`(3C)

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

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

UNICOS Networking Facilities Administrator's Guide, Cray Research publication SG–2304