UNICOS® System Libraries Reference Manual

SR–2080 10.0

Portions of this product may still be in development. The existence of those portions still in development is not a commitment of actual release or support by Cray Research, Inc. Cray Research, Inc. assumes no liability for any damages resulting from attempts to use any functionality or documentation not officially released and supported. If it is released, the final form and the time of official release and start of support is at the discretion of Cray Research, Inc.

Portions of the text in this document are taken directly from documents written by Jim Thomas and are used by permission of Taligent, Inc. The documents from which the text is taken are works in progress.

New Features

This rewrite of the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080, supports the 10.0 release of the UNICOS operating system. The following man page changes have been made since the UNICOS 9.0 release:

The following are new man pages to support the Array Services library (libarray):

- `intro_libarray`(3X) - Introduction to the Array Services library
- `aserrorcode`(3X), `asmakeerror`(3X), `asperror`(3X), `asstrerror`(3X) – Error messages
- `asopenserver`(3X), `asopenserver_from_optinfo`(3X), `asparseopts`(3X), `assetserveropt`(3X) – Connections to the Array Services daemon
- `asgetattr`(3X), `asgetdfltarray`(3X), `aslistarrays`(3X), `aslistmachines`(3X) – Database interrogation
- `asallocash`(3X), `asashisglobal`(3X), `asashofpid`(3X), `aslistashs`(3X), `aspidsinash`(3X) – Array session handle management and interrogation
- `asfreearray`(3X), `asfreearraylist`(3X), `asfreearraypidlist`(3X), `asfreeashlist`(3X), `asfreecmdrsltlist`(3X), `asfreemachinelist`(3X), `asfreemachinepidlist`(3X), `asfreeoptinfo`(3X), `asfreepidlist`(3X) – Data structure release
- `ascommand`(3X), `askillash_array`(3X), `askillpid_server`(3X), `asrcmd`(3X) – Array Command Execution

Because the following tools or functions are not supported in UNICOS 10.0, all references to them have been deleted:

- `cdbx` debugging tool
- `getnetinfo` function

Information has been added on the double l (el) function to the following man pages: `abs`, `div`, `printf`, `strtol`, `vprintf`.

The following are new man pages that support the IEEE floating-point implemenation:

- `ieee_float` – Introductory page
- `fp.h` – Header page for macros and functions to support general IEEE floating-point programming
- `copysign` – Copies the sign of its second argument to the value of its first argument

- `fpclassify` – Identifies its argument as NaN, infinite, normal, subnormal, or zero
- `isgreater` – Determines the relationship between two IEEE floating-point arguments
- `logb` – Returns the signed exponent of its argument
- `nextafter` – Returns the next value in the direction of the second argument
- `remainder` – Divides its arguments and returns the remainder
- `rint` – Rounds its argument to an integral value in IEEE floating-point format
- `rinttol` – Rounds a floating-point number to a long integer value
- `scalb`, `scalbf`, `scalbl` – Computes x * `FLT_RADIX`*n* efficiently
- `signbit` – Determines if its argument value is negative
- `fenv.h` – Header page for IEEE floating-point environment
- `feclearexcept` – Manages IEEE floating-point exception flags
- `fegetround` – Manages the rounding direction of IEEE floating-point numbers
- `fegetenv` – Manages the IEEE floating-point environment
- `fedisabletrap` – Manages IEEE floating-point traps

The following man pages have been updated to include IEEE information:

- `asin`, `bessel`, `erf`, `exp`, `frexp`, `lgamma`, `math`, `pow`, `sin`, `sinh`, `sqrt` – Explains error handling on IEEE floating-point systems
- `float.h` – IEEE floating-point values added to table
- `math.h` – Defines new IEEE floating-point macros
- `strtod` – Uses new IEEE floating-point macros

The following man pages contain changes:

- `mtimesx`(3F) – *rarray* argument was changed to CALL; description of *overlap* argument was updated
- `regexec` – Added `REG_WORDS` function
- `cpused` – Updated wording of functionality
- `gethost` – Added `int sethostlookup (`*int lookup_type*`)` function
- `scanf` – Added description for `B` as valid conversion directive character; added information on double l (el) option
- `syslog` – Added RETURN VALUES section

- `strptime` – Updated for year 2000 compliance

The following are new man pages:

- `pthread_atfork` – Register fork handlers (supports threading feature)
- `ndbm`, `dbm_open`, `dbm_close`, `dbm_fetch`, `dbm_store`, `dbm_delete`, `dbm_firstkey`, `dbm_nextkey`, `dbm_error`, `dbm_clearerr` – Database subroutines (maintains key/content pairs in a database)

Record of Revision

Version	*Description*
	June 1990 Original Printing. This manual supports the version of the UNICOS C library released with the Cray Standard C compiler releases 1.0 and 2.0.
6.0	January 1991 This manual supports the UNICOS 6.0 release.
7.0	August 1992 This manual supports the C library functions provided with the UNICOS 7.0 release.
8.0	January 1994 This manual supporst the C library functions provided with the UNICOS 8.0 release.
9.0	September 1995 This manual supports the UNICOS 9.0 release.
10.0	November 1997 This manual supports the UNICOS 10.0 release. The New Features page provides information about the changes documented in this manual.

Preface

This publication documents the UNICOS system library functions provided with the UNICOS 10.0 release running on Cray Research computer systems.

This is a reference manual for programmers. Readers should have a working knowledge of either the UNICOS or the UNIX operating system.

For closely related routines, two more more routines may be described on the same page.

Related publications

The following man page manuals contain additional information that may be helpful.

Note: For the UNICOS 10.0 release, man page reference manuals are not orderable in printed book form. Instead, they are available as printable PostScript files provided on the same DynaWeb CD as the rest of the supporting documents for this release. Individual man pages are still available online and can be accessed by using the `man`(1) command.

- *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011
- *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012
- *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014
- *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

The following ready references are available in printed form from the Distribution Center:

- *UNICOS User Commands Ready Reference*, Cray Research publication SQ–2056
- *UNICOS System Libraries Ready Reference*, Cray Research publication SQ–2147
- *UNICOS System Calls Ready Reference*, Cray Research publication SQ–2215
- *UNICOS Administrator Commands Ready Reference*, Cray Research publication SQ–2413

The following documents contain additional information that may be helpful:

- *Cray Standard C Reference Manual*, Cray Research publication SR–2074
- *Cray Standard C Ready Reference*, Cray Research publication SQ–2076
- *Scientific Libraries Reference Manual*, Cray Research publication SR–2081
- *Remote Procedure Call (RPC) Reference Manual*, Cray Research publication SR–2089
- *Intrinsic Procedures Reference Manual*, Cray Research publication SR–2138
- *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165
- *Compiler Information File (CIF) Reference Manual*, Cray Research publication SR–2401
- *UNICOS Macros and Opdefs Reference Manual*, Cray Research publication SR–2403
- *Cray Assembly Language (CAL) for Cray PVP Systems Reference Manual*, Cray Research publication SR–3108
- *CF90 Ready Reference*, Cray Research publication SQ–3900
- *CF90 Commands and Directives Reference Manual*, Cray Research publication SR–3901
- *Fortran Language Reference Manual, Volume 1*, Cray Research publication SR–3902
- *Fortran Language Reference Manual, Volume 2*, Cray Research publication SR–3903
- *Fortran Language Reference Manual, Volume 3*, Cray Research publication SR–3905

Ordering Cray Research publications

The *User Publications Catalog*, Cray Research publication CP–0099, describes the availability and content of all Cray Research hardware and software documents that are available to customers. Cray Research customers who subscribe to the Cray Inform (CRInform) program can access this information on the CRInform system.

To order a document, either call the Distribution Center in Mendota Heights, Minnesota, at +1–612–683–5907, or send a facsimile of your request to fax number +1–612–452–0141. Cray Research employees may send electronic mail to `orderdsk` (UNIX system users).

Customers who subscribe to the CRInform program can order software release packages electronically by using the `Order Cray Software` option.

Customers outside of the United States and Canada should contact their local service organization for ordering and documentation information.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
`manpage`(*x*)	Man page section identifiers appear in parentheses after man page names. The following list describes the identifiers: 1 User commands 1B User commands ported from BSD 2 System calls 3 Library routines, macros, and opdefs 4 Devices (special files) 4P Protocols 5 File formats 7 Miscellaneous topics 7D DWB-related information 8 Administrator commands Some internal routines (for example, the `_assign_asgcmd_info`() routine) do not have man pages associated with them.

variable	Italic typeface denotes variable entries and words or concepts being defined.
user input	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.

The following machine naming conventions may be used throughout this document:

Term	Definition
Cray PVP systems	All configurations of Cray parallel vector processing (PVP) systems.
Cray MPP systems	All configurations of the CRAY T3D series. The UNICOS operating system is not supported on CRAY T3E systems. CRAY T3E systems run the UNICOS/mk operating system.
All Cray Research systems	All configurations of Cray PVP and Cray MPP systems that support this release.

The default shell in the UNICOS and UNICOS/mk operating systems, referred to in Cray Research documentation as the *standard shell*, is a version of the Korn shell that conforms to the following standards:

- Institute of Electrical and Electronics Engineers (IEEE) Portable Operating System Interface (POSIX) Standard 1003.2–1992
- X/Open Portability Guide, Issue 4 (XPG4)

The UNICOS and UNICOS/mk operating systems also support the optional use of the C shell.

Cray UNICOS Version 10.0 is an X/Open Base 95 branded product.

Man page sections

The entries in this document are based on a common format. The following list shows the order of sections in an entry and describes each section. Most entries contain only a subset of these sections.

Section heading	Description
NAME	Specifies the name of the entry and briefly states its function.
SYNOPSIS	Presents the syntax of the entry.
IMPLEMENTATION	Identifies the Cray Research systems to which the entry applies.
STANDARDS	Provides information about the portability of a utility or routine.
DESCRIPTION	Discusses the entry in detail.
NOTES	Presents items of particular importance.
CAUTIONS	Describes actions that can destroy data or produce undesired results.
WARNINGS	Describes actions that can harm people, equipment, or system software.
ENVIRONMENT VARIABLES	Describes predefined shell variables that determine some characteristics of the shell or that affect the behavior of some programs, commands, or utilities.
RETURN VALUES	Describes possible return values that indicate a library or system call executed successfully, or identifies the error condition under which it failed.
EXIT STATUS	Describes possible exit status values that indicate whether the command or utility executed successfully.
MESSAGES	Describes informational, diagnostic, and error messages that may appear. Self-explanatory messages are not listed.
ERRORS	Documents error codes. Applies only to system calls.

FORTRAN EXTENSIONS	Describes how to call a system call from Fortran. Applies only to system calls.
BUGS	Indicates known bugs and deficiencies.
EXAMPLES	Shows examples of usage.
FILES	Lists files that are either part of the entry or are related to it.
SEE ALSO	Lists entries and publications that contain related information.

Reader comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. You can contact us in any of the following ways:

- Send us electronic mail at the following address:

  `publications@cray.com`

- Contact your customer service representative and ask that an SPR or PV be filed. If filing an SPR, use `PUBLICATIONS` for the group name, `PUBS` for the command, and `NO-LICENSE` for the release name.

- Call our Software Publications Group in Eagan, Minnesota, through the Customer Service Call Center, using either of the following numbers:

  1–800–950–2729 (toll free from the United States and Canada)

  +1–612–683–5600

- Send a facsimile of your comments to the attention of "Software Publications Group" in Eagan, Minnesota, at fax number +1–612–683–5599.

We value your comments and will respond to them promptly.

CONTENTS

NAME

`intro` – Introduction to the UNICOS C library

IMPLEMENTATION

All Cray Research systems

INTRODUCTION

This manual describes the UNICOS C library functions used with the Cray Standard C compiler on all Cray Research, Inc. (CRI) computer systems running the UNICOS operating system release 9.0 or higher. It also describes two sets of Fortran library functions - the sort routines and multitasking routines. In addition, four Network Queuing Environment (NQE) functions have been added to the Network Access section.

This manual describes a rich selection of user-level functions. These libraries are supplemented by other UNICOS libraries that may be useful to programmers. These include the following:

- The UNICOS Fortran library, documented in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165
- The UNICOS math library, documented in the *Intrinsic Procedures Reference Manual*, Cray Research publication SR–2138
- The UNICOS scientific library, documented in the *Scientific Libraries Reference Manual*, Cray Research publication SR–2081
- The UNICOS specialized libraries, documented in the *Compiler Information File (CIF) Reference Manual*, Cray Research publication SR–2401, and the *Remote Procedure Call (RPC) Reference Manual*, Cray Research publication SR–2089

In addition, the UNICOS operating system performs many functions for the user through system calls that are called in the same manner as library functions. (System calls are documented in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012.)

Some library functions are specific to certain groups of CRI mainframes. These are identified by one or more of the mainframe designations under the heading IMPLEMENTATION on each man page.

STANDARDS

This manual describes functions that are defined by several important standards. The libraries also contain additional functions ported (with permission) from other sources or written by CRI. The relevant standards for each man page are listed under the STANDARDS heading. Do not infer, however, from the reference to a standard that the entire library has necessarily been validated to conform to that standard. Validation of conformance to these standards is an issue discussed in other Cray Research documents.

In this manual, the reference to a standard provides you with information about the portability of code using that function. For example, if the entry for the function states that the function is defined in the ISO/ANSI standard, you can expect a given function to be found in any vendor's system that conforms to the ISO/ANSI standard. On the other hand, if the entry for the function states that it is a CRI extension, you cannot expect it to be found in other vendor's systems.

The specific meanings of the terms in the STANDARDS section are as follows:

Term	Description
ISO/ANSI	Defined in the ISO and ANSI standard. In this manual, the term *the standard* refers to this combined standard.
POSIX	Defined in the POSIX standards IEEE Std 1003.1-1990, or IEEE Std 1003.2-1992, but not defined in the ISO/ANSI standard. The POSIX 1003.1 standard embraces the ISO/ANSI standard for C but includes more. For brevity, POSIX is not stated in the STANDARDS section if ISO/ANSI has been specified.
PThreads	Defined in the POSIX standards IEEE Std 1003.1c-1994, but not defined in the ISO/ANSI, POSIX 1003.1, or POSIX 1003.2 standards.
XPG4	Defined as part of the X/Open Common Applications Environment Specification, Issue 4. The XPG4 standard embraces the ISO/ANSI standard for C, as well as the POSIX 1003.1 and 1003.2 standards, but includes more. For brevity, XPG4 is not stated in the STANDARDS section if either ISO/ANSI or POSIX has been specified.
AT&T extension	Not defined in any of the previous standards; it originated from one or more of the software releases from AT&T.
BSD extension	Not defined in any of the previous standards; it originated from the Fourth Berkeley Software Distribution under license from The Regents of the University of California.
CRI extension	Not defined in any of the previous standards; added by CRI.

LOADING THE UNICOS LIBRARIES

UNICOS libraries are automatically available on all UNICOS systems when you compile your C program.

All library functions necessary to support a strictly conforming Standard C program are located in several distinct libraries; the `cc`(1) command automatically issues the appropriate directives to load the program with the appropriate functions. If your program strictly conforms to the standard, you do not need to know library names and locations.

However, if your program requires other libraries or if you want direct control over the loading process, more knowledge of loaders and libraries is necessary.

There is no library search order dependency. Default libraries on PVP systems are as follows:

```
libc.a          libf.a
libfi.a         libm.a
libsci.a        libu.a
```

Default libraries on MPP systems are as follows:

```
libc.a          libf.a
libfi.a         libm.a
libpvm3.a       libsci.a
libsma.a        libu.a
```

If you specify personal libraries by using the `-l` loader option, as in the following example, those libraries are added to the top of the preceding list.

```
cc -h intrinsics target.c -l mine
```

When the preceding command line is issued, the loader searches for a library named `libmine.a` (following the naming convention) and adds it to the search list. Whenever additional libraries are specified on the command line, the loader first searches for the named library in directory `/lib`, then in directory `/usr/lib`, unless a full path name is specified. If the library name begins with a "`.`" or a "`/`", the loader assumes that a full path name is given, and looks there first.

HEADERS FOR THE UNICOS C LIBRARY

Associated with the UNICOS C library are a set of headers that are helpful as an interface to the library.

The headers contain function declarations in function prototype format for all the C library functions defined by the standard. If you include these headers in your C program, the function prototype information is automatically provided to the Standard C compiler. The compiler uses the information to ensure that the functions are called with the proper number and type of arguments and that the function call has a proper interface with the library function. Note, however, that nonstandard functions may not have declarations in any header; in that case, refer to the individual function descriptions.

Headers can be included in any order. Each can be included more than once in a given scope; if so, the effect is no different from that produced when the header is included only once. The only exception to this rule is the header <`assert.h`>; the effect of including `<assert.h>` depends on the definition of the `NDEBUG` macro at the time of each inclusion.

If a header is used, include it outside of any external declaration or definition. Be sure to include it before the first reference to any of the functions or objects it declares, or to any of the types or macros it defines. If an identifier is declared or defined in more than one header, however, the second and subsequent associated headers can be included after the initial reference to the identifier. The program cannot contain any macros with names lexically identical to keywords currently defined prior to the inclusion.

The UNICOS headers also provide the following:

- Types definitions that declare a name synonymous with a type.
- Macros that have no parameters and define useful values.
- Macros with parameters, some of which are macro versions of library functions; this places the function code inline, which saves the overhead of the function calling sequence.

The compiler automatically searches, by default, the following directory:

```
/usr/include
```

However, if your program requires other headers, you may also specify other directories containing headers by using the `cc -I` option.

The following headers, called the "standard headers," are associated with the UNICOS C library, as required by the standard:

```
<assert.h>    <locale.h>    <stddef.h>    <ctype.h>
<math.h>      <stdio.h>     <errno.h>     <setjmp.h>
<stdlib.h>    <float.h>     <signal.h>    <string.h>
<limits.h>    <stdarg.h>    <time.h>
```

The following headers are CRI extensions to the UNICOS C library, in addition to those required by the standard:

```
<complex.h>    <fortran.h>
```

The following headers are associated with the UNICOS C library, as required by the AT&T System V Interface Definition (SVID):

```
<assert.h>    <malloc.h>    <signal.h>    <ctype.h>
<nlist.h>     <stdio.h>     <errno.h>     <prof.h>
<string.h>    <ftw.h>       <pwd.h>       <time.h>
<grp.h>       <regexp.h>    <utmp.h>      <math.h>
<search.h>    <varargs.h>   <memory.h>    <setjmp.h>
```

Any header required by the SVID that is not a standard header is located in the `/usr/include` directory.

Note that headers appearing in more than one of the preceding lists may behave differently in different compilation modes; see `cc`(1).

The combination of those headers in the preceding lists are collectively called the UNICOS C library headers. There are other headers for other purposes in directory `/usr/include`; most of these other headers are not listed or described in this manual. (See the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014, for descriptions of these headers.)

RESERVED IDENTIFIERS IN STANDARD C

Each header declares or defines all identifiers listed in its associated section. The following identifiers are reserved by the standard for use by the UNICOS C library and headers. Standard-conforming programs must not declare or define the following identifiers:

- All identifiers that begin with an underscore and an uppercase letter or with two underscores are always reserved for library use.
- All identifiers that begin with an underscore are always reserved for use as identifiers with file scope in both the ordinary identifier and tag name spaces.

- Each macro name listed in any of the following header description pages is reserved for any use if any of its associated headers is included.
- All identifiers with external linkage in any of the following header description pages are always reserved for use as identifiers with external linkage.
- Each identifier with file scope listed in any of the following header introduction pages is reserved for use as an identifier with file scope in the same name space if any of its associated headers is included.

No other identifiers are reserved. If the program declares or defines an identifier with the same name as an identifier reserved in that context, the behavior is undefined.

USE OF LIBRARY FUNCTIONS

In this manual, the terms *function* and *subroutine* generally mean a block of code that performs a specific, documented task. The function may exist in a header as the definition of a macro, in the C library as compiled code, or in both. Unless there is a specific reason to state how a function is implemented, you need not know how it is implemented (as a macro or as compiled code), just that the task will be performed when the function is called.

If you desire access to a function, as opposed to a macro (for example, to take the address of a function to pass to another function), you need to include an `#undef` *function_name* directive following the `#include` directive for the header. Note that the standard prohibits the use of `#undef` directives with some of the standard functions. See the documentation for a particular function if you are not sure whether such a restriction exists.

Any function declared in a header can also be implemented as a macro defined in the header, so a library function should not be declared explicitly if its corresponding header is included. Any macro definition of a function can be suppressed locally by enclosing the name of the function in parentheses, because the name is then not followed by the left parenthesis that indicates expansion of a macro function name. For the same syntactic reason, it is permitted to take the address of a library function even if it is also defined as a macro.

The use of an `#undef` directive to remove any macro definition also ensures that you refer to an actual function, with exceptions identified in this manual (for example, `putc` and `getc`). Unless otherwise noted, any invocation of a library function that is implemented as a macro expands to code that evaluates each of its arguments exactly once, fully protected by parentheses where necessary, so it is generally safe to use arbitrary expressions as arguments. Likewise, those function-like macros described in the following pages can be invoked in an expression anywhere a function with a compatible return type could be called.

All object-like macros listed as expanding to integral constant expressions are suitable for use in `#if` preprocessing directives.

Provided that a library function can be declared without reference to any type defined in a header, you can also declare the function, either explicitly or implicitly, and use it without including its associated header. If a function that accepts a variable number of arguments is not declared, explicitly or by including its associated header, the behavior is undefined.

Each of the following statements applies in the detailed standard header descriptions that follow unless explicitly stated otherwise:

- If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer outside the address space of the program, or a null pointer), the behavior is undefined.
- If a function argument is described as being an array, the pointer actually passed to the function must have a value which ensures that all address computations and accesses to objects that would be valid if the pointer did point to the first element of such an array are in fact valid.

The following examples show how the `atoi` function can be used in any of several ways:

- By use of its associated header (possibly generating a macro expansion):

```
#include <stdlib.h>
const char *str;
/* . . . */
i=atoi(str);
```

- By use of its associated header (generating a true function reference):

```
#include <stdlib.h>
#undef atoi
const char *str;
/* . . . */
i=atoi(str);
```

or

```
#include <stdlib.h>
const char *str;
/* . . . */
i=(atoi)(str);
```

- By explicit declaration:

```
extern int atoi (const char *);
const char *str;
/* . . . */
i=atoi(str);
```

- By implicit declaration:

```
const char *str;
/* . . . */
i=atoi(str);
```

NAME

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

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

AT&T extension

DESCRIPTION

Use a64l and l64a to maintain numbers stored in base-64 ASCII characters. This is a notation by which long integers can be represented by up to 11 characters; each character represents a digit in a radix-64 notation.

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

The a64l function takes a pointer to a null-terminated base-64 representation and returns a corresponding long value. If the string pointed to by *s* contains more than 11 characters, a64l uses the first 10 plus the right 4 bits of the value of the eleventh character; all characters beyond the eleventh are discarded.

The l64a function takes a long argument and returns a pointer to the corresponding base-64 representation. If the argument is 0, l64a returns a pointer to a null string.

CAUTIONS

The value returned by l64a is a pointer into a static buffer, which is overwritten by each call.

Results are not portable because a long value on Cray Research computer systems is 64 bits, while a long value on most other machines is 32 bits.

NAME

abort – Generates an abnormal process termination

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The abort function causes abnormal program termination to occur, unless the signal SIGABRT is being caught and the signal handler does not return.

If the SIGABRT signal is being caught, the abort function sends the signal before performing any other actions.

The abort function calls all functions that are registered by the atabort(3C) function in the reverse order of their registration. Each function is called as many times as it was registered. The abort function flushes all output streams and closes all open streams.

The SIGABRT signal is set to its default action (if it was formerly being caught or ignored), and the signal is unblocked before sending it to the calling process by calling the raise(3C) function with the SIGABRT signal.

RETURN VALUES

The abort function does not return to the caller.

MESSAGES

If the current directory is writable, the abort function produces a core dump and the shell writes an informational message.

SEE ALSO

atexit(3C), raise(3C)

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

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

NAME

abs, labs, llabs – Returns the integer or long integer absolute value

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The abs function returns the absolute value of an integer *j*. If the result cannot be represented, the behavior is undefined.

The labs function is similar to the abs function, except that the argument and the returned value each have type long int.

The llabs function is similar to the abs function, except that the argument and the returned value each have type long long int.

NOTES

In twos complement representation, the absolute value of the negative integer with the largest magnitude cannot be represented. The behavior in this case is undefined.

SEE ALSO

floor(3C)

NAME

airlog – Logs messages to system log using syslog(3)

SYNOPSIS

```
#include <airlog.h>
int airlog (int severity, int productid, char *subproductid, char *message);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The airlog function is a part of the automated incident reporting (AIR) system. The airlog function formats messages and passes them to the syslog(3C) function, which then arranges to write the message onto a UNICOS system log maintained by syslogd(8).

The *severity* is selected from the following list:

Severity	Description
AIR_START	Normal daemon initiation
AIR_TERM	Normal daemon termination
AIR_PANIC	Abnormal daemon termination
AIR_CRIT	A disaster has occurred
AIR_WARN	Warning information; while not fatal, this information may be a precursor to disaster
AIR_ATTEN	Information to be displayed for the operator
AIR_INFO	Useful information to be logged
AIR_PULSE	Daemon heartbeat
AIR_FORK	Daemon has spawned a child process
AIR_USER	User-entered message
AIR_CONF	Configuration information

The *productid* is selected from the following list:

Product ID	Description
AIR_GUEST	UNICOS under UNICOS (guest)
AIR_UNICOS	Kernel
AIR_NQS	Network Queuing System
AIR_NEWQS	New Network Queuing System
AIR_TCP	Internet Transmission Control Protocol
AIR_TAPE	Tape subsystem

AIR_DMF	Data Migration Facility
AIR_NFS	Network file system
AIR_ACCT	Accounting
AIR_DISK	CRI disk farm
AIR_SUPERL	OSI-based networking
AIR_SHARE	UNICOS share scheduler
AIR_CRON	cron daemon

The *subproductid* is a comma-delimited string that further delineates the origin of the message. For example, if the *productid* is NQS, a possible string for the *subproductid* field could be "qfdaemon,readq,end".

The *message* string denotes the actual textual information to be logged.

The airlog function creates the format of the log entry by ordering the given arguments and adding an identifying key whose actual contents are defined based upon the *severity* argument, as follows:

Severity	Description
AIR_START	Process, job, parent process, user, group, and account IDs of the daemon
All others	Process and job IDs of the daemon

NOTES

The /etc/syslog.conf file must have the following entry:

```
local7.debug          /usr/logs/airlog
```

FILES

/usr/logs/airlog	AIR system log file.

RETURN VALUES

The airlog function returns –1 if it is unable to allocate memory for the message buffer; otherwise, it returns 0.

SEE ALSO

syslog(3C)

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

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

NAME

intro_libarray – Introduces the Array Services library (libarray)

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The Array Services library (libarray) provides functions that allow you to interrogate the Array Services configuration database and call on the services of the Array Services daemon, arrayd(8). The library is used by several array software products for the IRIX and UNICOS operating systems. For more information, see the array_sessions(7) and array_services(7) man pages.

The programming interface to Array Services is declared in the arraysvcs.h header file. The IRIX libarray.so and the UNICOS libarray.a libraries contain the functions. You can load the library by using the -larray option with cc(1) or ld(1).

The following subsections summarize the functions.

Error Messages

aserrorcode(3x)	Provides Array Services error information
asmakeerror(3x)	Generates an Array Services error code
asperror(3x)	Prints an Array Services error message
asstrerror(3x)	Gets an Array Services error message string

Connections to the Array Services Daemon

ascloseserver(3x)	Destroys an array server token
asdfltserveropt(3x)	Retrieves the standard default value for options when a new server token is created using asopenserver(3x)
asgetserveropt(3x)	Returns the local options currently used by an instance of arrayd(8)
asopenserver(3x)	Creates an array server token
asopenserver_from_optinfo(3x)	Creates and modifies an array server token using parameters taken from an asoptinfo_t structure
asparseopts(3x)	Parses standard Array Services command line options
assetserveropt(3x)	Returns the default options in effect at an instance of arrayd(8)

Database Interrogation

asgetattr(3x)	Searches an attribute list for a particular name
asgetdfltarray(3x)	Gets information about the default array
aslistarrays(3x)	Enumerates known arrays
aslistmachines(3x)	Enumerates machines in an array

Array Session Handle Management and Interrogation

asallocash(3x)	Allocates a global array session handle
asashisglobal(3x)	Determines if an array session handle is global
asashofpid(3x)	Obtains the array session handle of a process
aspidsinash(3x)	Returns a list of processes that belong to the specified array session handle
aspidsinash_array(3x)	Returns a list of processes in the specified array session for all of the machines in the specified array
aspidsinash_local(3x)	Returns only those processes in the array session that are running on the local machine
aspidsinash_server	Returns the list of processes in the specified array session that are running on the specified server
aslistashs(3x)	Returns a list of array session handles
aslistashs_array(3x)	Returns a list of array session handles that are currently active on the specified array
aslistashs_local(3x)	Returns a list of array session handles that are currently active on the local machine
aslistashs_server(3x)	Returns a list of array session handles that are currently active on the specified server

Data Structure Release

asfreearray(3x)	Releases array information structure
asfreearraylist(3x)	Releases array information structures
asfreearraypidlist(3x)	Releases array-wide process identification enumeration structures
asfreeashlist(3x)	Releases array session handle enumeration structures
asfreecmdrsltlist(3x)	Releases array command result structures
asfreemachinelist(3x)	Releases machine information structures
asfreemachinepidlist(3x)	Releases process identification enumeration structures
asfreeoptinfo(3x)	Releases command line options information structure

asfreepidlist(3x)	Releases process identification enumeration structures

Array Command Execution

ascommand(3x)	Executes an array command
askillash_array(3x)	Sends a signal to an array session on the specified array
askillash_local(3x)	Sends a signal to an array session on the local machine
askillash_server(3x)	Sends a signal to an array session on the specified server
askillpid_server(3x)	Sends a signal to a remote process
asrcmd(3x)	Executes a command on a remote machine using a single string that contains the entire command line
asrcmdv(3x)	Executes a command on a remote machine using pointers

SEE ALSO

asallocash(3x), asashisglobal(3x), asashofpid(3x), ascommand(3x), aserrorcode(3x), asfreearray(3x), asfreearraylist(3x), asfreearraypidlist(3x), asfreeashlist(3x), asfreecmdrsltlist(3x), asfreemachinelist(3x), asfreemachinepidlist(3x), asfreeoptinfo(3x), asfreepidlist(3x), asgetattr(3x), asgetdfltarray(3x), askillash_array(3x), askillpid_server(3x), aslistarrays(3x), aslistashs(3x), aslistmachines(3x), asmakeerror(3x), asopenserver(3x), asopenserver_from_optinfo(3x), asparseopts(3x), asperror(3x), aspidsinash(3x), asrcmd(3x), assetserveropt(3x), asstrerror(3x)

array(1), cc(1), ld(1)

array_services(7), array_sessions(7)

arrayd(8)

NAME

asallocash – Allocates a global array session handle

SYNOPSIS

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

ash_t asallocash(asserver_t Server, const char *Array);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The asallocash function allocates a global array session handle in the specified array. The resulting array session handle is guaranteed to be unique across all of the machines in that array.

The formal parameters are as follows:

Server Specifies an optional array server token, which can be used to direct the request to a specific Array Services daemon. If you specify a null pointer, the request is processed by the default Array Services daemon. For information on how the default Array Services daemon is selected, see the array(1) man page. For information on creating an array server token, see the asopenserver(3x) man page.

Array Specifies the name of the array as an ordinary character string. If you specify a null pointer, the array session handle is allocated in the default array of the Array Services daemon.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain this function. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

RETURN VALUES

If successful, asallocash returns the newly allocated global array session handle. If unsuccessful, asallocash returns a value of –1 and sets aserrorcode(3x) accordingly.

SEE ALSO

asashisglobal(3x), aserrorcode(3x), asopenserver(3x) array(1), cc(1), ld(1)

setash(2)

array_services(7), array_sessions(7)

arrayd(8)

NAME

asashisglobal – Determines if an array session handle is global

SYNOPSIS

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

int asashisglobal(ash_t ASH)
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The asashisglobal function determines if an array session handle is global. A global array session handle is guaranteed to be unique across all machines in an array.

The formal parameter is as follows:

ASH Specifies an array session handle.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain this function. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

RETURN VALUES

The asashisglobal function returns a nonzero value if the specified array session handle is global. If it is not global, asashisglobal returns a value of 0.

SEE ALSO

asallocash(3x)

cc(1), ld(1)

array_services(7), array_sessions(7)

NAME

asashofpid – Obtains the array session handle of a process

SYNOPSIS

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

ash_t *asashofpid(pid_t PID);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The asashofpid function returns the array session handle of the process with the specified process identification number. The process is assumed to run on the local machine.

The formal parameter is as follows:

PID Specifies the process identification number.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain this function. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

RETURN VALUES

If successful, asashofpid returns the array session handle of the specified process. If *PID* is a negative number, asashofpid returns the array session handle of the current process. If unsuccessful, asashofpid returns a value of –1 and sets aserrorcode(3) accordingly.

SEE ALSO

asashisglobal(3x), aserrorcode(3x), aspidsinash(3x)

cc(1), ld(1)

getash(2)
array_services(7), array_sessions(7)

NAME

ascommand – Executes an array command

SYNOPSIS

```
#include <arraysvcs.h>
ascmdrsltlist_t *ascommand(asserver_t Server, ascmdreq_t *Command);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The ascommand function executes an array command. The command request is processed by an Array Services daemon. That Array Services daemon is responsible for translating the array command into an actual IRIX or UNICOS command, running it on one or more machines in the requested array, and returning the results.

The formal parameters are as follows:

Server Specifies an optional array server token that can be used to direct the request to a specific Array Services daemon. If you set *Server* to a null pointer, the request is processed by the default Array Services daemon. For information on how the default Array Services daemon is selected, see the array(1) man page.

Command Points to an ascmdreq_t structure (defined in the arraysvcs.h file) that describes the command request. For more information, see the Structure Members subsection of this man page.

Structure Members

The ascmdreq_t structure that the *Command* formal parameter points to has the following format:

```
typedef struct ascmdreq {
        char     *array;
        uint32_t flags;
        int      numargs;
        char     **args;
        uint32_t ioflags;
} ascmdreq_t;
```

The members in the structure are as follows:

array Specifies the name of the array on which the command should be executed. If you set *array* to a null pointer, the server's default destination is used if one has been specified, otherwise the command request is rejected.

flags Specifies various control options for the command. It is constructed from the logical OR of zero or more of the following flags. (If you do not specify a flag, set the value to 0 so that no control options are set.) The flags are as follows:

`ASCMDREQ_LOCAL`	Runs the command on the server machine only, rather than broadcasting it to the machines in an array. If you specify this flag, the *array* member of *Command* is ignored.
`ASCMDREQ_NEWSESS`	Runs the command in a new global array session. The Array Services daemon allocates a new global array session handle and ensures that each machine executes the array command in an array session using this handle.
`ASCMDREQ_OUTPUT`	Collects output from the array command. If you specify this flag, the standard output and standard error of the array command is saved on each machine. If any output is generated on a particular machine, the `ascmdrslt_t` structure for that machine contains a pathname to a temporary file containing the output.
`ASCMDREQ_NOWAIT`	Forces the Array Services daemon to return results immediately. Ordinarily, the Array Services daemon waits for the array command to complete before returning the results. The `ascmdrslt_t` structure for each machine indicates that the command has been initiated, but it does not have a valid exit status for the command.
`ASCMDREQ_INTERACTIVE`	Specifies that socket connections should be made to one or more of the command's standard I/O file descriptors: • Standard input (`stdin`) • Standard output (`stdout`) • Standard error (`stderr`) The exact connections to be made are specified in the *ioflags* member of *Command*. If successful, the `ascmdrslt_t` structure for each machine contains socket descriptors for each of the requested connections. If this flag is specified, then the `ASCMDREQ_OUTPUT` flag is ignored and the `ASCMDREQ_NOWAIT` flag is implied (that is, an interactive request never waits for the command to complete).

numargs Specifies the number of arguments in the *args* array. This member behaves similarly to the `argc` argument to a standard C program.

args Specifies the array command itself and any arguments to it. This member behaves similarly to the `argv` argument to a standard C program.

ioflags Indicates which of the command's standard I/O descriptors should be routed back to the caller through a socket connection. This member is examined only when the *flags* member has the ASCMDREQ_INTERACTIVE flag set. The *ioflags* member is constructed from the logical OR of one or more of the following flags:

ASCMDIO_STDIN	Requests a socket attached to the command's standard input.
ASCMDIO_STDOUT	Requests a socket attached to the command's standard output.
ASCMDIO_STDERR	Requests a socket attached to the command's standard error.
ASCMDIO_SIGNAL	Requests a socket that can be used to deliver signals to the command.
ASCMDIO_OUTERRSHR	Indicates that the command's standard error should be routed back over the standard output channel. This flag is ignored if you do not also specify ASCMDIO_STDERR.

A series of ascmdrslt_t structures summarize the results from each machine. An ascmdrsltlist_t structure bundles the list of these structures together. The ascommand function returns a pointer to an ascmdrsltlist_t structure.

The arraysvcs.h file defines the ascmdrslt_t and the ascmdrsltlist_t structures. An ascmdrslt_t structure has the following format:

```
typedef struct ascmdrslt {
        char       *machine;
        ash_t      ash;
        uint32_t   flags;
        aserror_t  error;
        int        status;
        char       *outfile;

        /* These fields only valid if ASCMDRSLT_INTERACTIVE set */
        uint32_t   ioflags;
        int        stdinfd;
        int        stdoutfd;
        int        stderrfd;
        int        signalfd;
} ascmdrslt_t;
```

The members are as follows:

machine Contains the name of the machine that generated this particular response. This is typically the network hostname of that machine, although the system administrator can override that value with a LOCAL HOSTNAME entry in the Array Services configuration file.

ash Contains the array session handle.

flags Contains flags that describe details about the command results. This member is constructed from the logical OR of zero or more of the following flags. The flags are as follows:

ASCMDRSLT_OUTPUT	Indicates that the command has generated output that has been saved in a temporary file. The *outfile* member contains the name of the temporary file.
ASCMDRSLT_MERGED	Indicates that although the array command may have been run on more than one machine, the results were merged together by a MERGE command on the Array Services daemon. The ascmdrslt_t structure describes the results of the MERGE command only.
ASCMDRSLT_ASH	Indicates that the array command was run using a global array session handle. The *ash* member of the ascmdrslt_t structure contains the array session handle.
ASCMDRSLT_INTERACTIVE	Indicates that one or more connections have been established with the standard I/O file descriptors of the running command. The ioflags member of the ascmdrslt_t structure describes the specific connections.

error Contains the results of the command on the particular machine. This member is a standard libarray error code. For details on libarray error codes, see the aserrorcode(3x) man page.

status Contains the final exit status of the array command's process on this machine, assuming that the errno subfield of error is ASE_OK and the what member is ASOK_COMPLETED.

outfile Contains the name of the temporary file.

ioflags Contains flags that describe which connections have been established with the running command. It is only valid if the ASCMDRSLT_INTERACTIVE flag is set in the *flags* member. This member is constructed from the logical OR of one or more of the following flags:

ASCMDIO_STDIN	Indicates that a socket connection has been established with the command's standard input. The *stdinfd* member of the ascmdrslt_t structure contains the socket descriptor. Data written to this descriptor is presented to the command's standard input.
ASCMDIO_STDOUT	Indicates that a socket connection has been established with the command's standard output. The *stdoutfd* member of the ascmdrslt_t structure contains the socket descriptor. Data that the command writes to its standard output can be read from this descriptor.

ASCMDIO_STDERR	Indicates that a socket connection has been established with the command's standard error. The *stderrfd* member of the ascmdrslt_t structure contains the socket descriptor. Data that the command writes to its standard error can be read from this descriptor.
ASCMDIO_SIGNAL	Indicates that a socket connection that can be used to deliver signals to the command has been established. The *signalfd* member of the ascmdrslt_t structure contains the socket descriptor. Any signal can be delivered to the running command by writing a single byte containing the desired signal number to this descriptor.

In some implementations, the same socket may be used to handle both the standard input and standard output connections or both the standard error and signal connections. Therefore, caution should be exercised before trying to close only one socket in either of those pairs.

stdinfd Specifies the standard input socket descriptor.

stdoutfd Specifies the standard output socket descriptor.

sterrfd Specifies the standard error socket descriptor.

signalfd Specifies the signal socket descriptor.

The libarray library uses the malloc(3C) function to allocate storage for the structures. To release the storage space, use the asfreecmdrsltlist(3x) function.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain this function. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

RETURN VALUES

If successful, ascommand returns a pointer to an ascmdrsltlist_t structure. If unsuccessful, ascommand returns a null pointer and sets aserrorcode accordingly.

SEE ALSO

aserrorcode(3x), asfreecmdrsltlist(3x), asopenserver(3x) malloc(3C)

array(1), cc(1), ld(1)

array_services(7), array_sessions(7)

arrayd(8)

NAME

aserrorcode – Provides Array Services error information

SYNOPSIS

```
#include <arraysvcs.h>

extern aserror_t aserrorcode;

#define aserrnoc(errorcode)
#define aserrwhatc(errorcode)
#define aserrwhyc(errorcode)
#define aserrextrac(errorcode)

#define aserrno      ...
#define aserrwhat   ...
#define aserrwhy     ...
#define aserrextra  ...
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

Upon completion, many Array Services library functions store status information in four fields of the *aserrorcode* global variable. You can extract information from the *aserrorcode* fields by using the following macros, which are defined in the arraysvcs.h file:

aserrno	Summarizes the results of the most recent Array Services function.
aserrwhat	Describes the particular component that experienced trouble. This macro only applies to certain values of aserrno.
aserrwhy	Describes why the error occurred. This macro only applies to certain values of aserrno.
aserrextra	Contains additional information supplied by certain combinations of aserrno, aserrwhat and aserrwhy. The exact information depends on the particular combination.

The arraysvcs.h file describes the specific values that can be stored in these fields.

You can extract the same type of information from the fields of a value of type aserror_t that you specify by using the the following macros:

- aserrnoc
- aserrwhatc
- aserrwhyc
- aserrextrac

NOTES

The IRIX `libarray.so` and the UNICOS `libarray.a` libraries contain this global variable. You can load the `libarray.so` or `libarray.a` library by using the `-larray` option with `cc`(1) or `ld`(1).

RETURN VALUES

The `aserrno`, `aserrwhat`, `aserrwhy`, and `aserrextra` macros return the corresponding field from the *aserrorcode* global variable.

The `aserrnoc`, `aserrwhatc`, `aserrwhyc`, and `aserrextrac` macros return the corresponding field from the specified `aserror_t` value.

SEE ALSO

`asmakeerror`(3x), `asperror`(3x), `asstrerror`(3x)

`cc`(1), `ld`(1)

`array_services`(7), `array_sessions`(7)

NAME

asfreearray – Releases array information structure

SYNOPSIS

```
#include <arraysvcs.h>
void asfreearray(asarray_t *ArrayInfo, uint32_t Flags);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The asfreearray function releases the resources used by the specified asarray_t structure. The asgetdfltarray(3x) function typically generates this structure.

The formal parameters are as follows:

ArrayInfo Specifies a pointer to the asarray_t structure whose resources are to be released.

Flags [Reserved for future enhancements.] Set this value to 0.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain this function. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

SEE ALSO

asgetdfltarray(3x), aslistarrays(3x)

cc(1), ld(1)

array_services(7), array_sessions(7)

NAME

asfreearraylist – Releases array information structures

SYNOPSIS

```
#include <arraysvcs.h>
void asfreearraylist(asarraylist_t *ArrayInfoList, uint32_t Flags);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The asfreearraylist function releases the resources used by the specified asarraylist_t structure. The aslistarrays(3x) function typically generates these structures.

The formal parameters are as follows:

ArrayInfoList Specifies a pointer to the asarraylist_t structure whose resources are to be released.

Flags Specifies the resources to be released. *Flags* can have one of the following values:

- ASFLF_FREEDATA — Releases the storage used by the individual asarray_t structure elements.
- 0 — Releases only the asarraylist_t structure.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain this function. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

SEE ALSO

aslistarrays(3x)

cc(1), ld(1)

array_services(7), array_sessions(7)

NAME

asfreearraypidlist – Releases array-wide process identification enumeration structures

SYNOPSIS

```
#include <arraysvcs.h>
void asfreearraypidlist(asarraypidlist_t *PIDList, uint32_t Flags);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The asfreearraypidlist function releases the resources used by the specified asarraypidlist_t structure. The aspidsinash_array(3x) function typically generates these structures.

The formal parameters are as follows:

PIDList Specifies a pointer to the asarraypidlist_t structure whose resources are to be released.

Flags Specifies the resources to be released. *Flags* can have one of the following values:

ASFLF_FREEDATA	Releases the storage used by the individual asmachinepidlist_t structure elements.
0	Releases only the storage used by the asarraypidlist_t structure itself.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain this function. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

SEE ALSO

aspidsinash_array(3x)

cc(1), ld(1)

array_services(7), array_sessions(7)

NAME

asfreeashlist – Releases array session handle enumeration structures

SYNOPSIS

```
#include <arraysvcs.h>
void asfreeashlist(asashlist_t *ASHlist, uint32_t Flags);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The asfreeashlist function releases the resources used by the specified asashlist_t structure. The aslistashs(3x) function typically generates these structures.

The formal parameters are as follows:

ASHlist Specifies a pointer to the asashlist_t structure whose resources are to be released.

Flags [Reserved for future expansion.] Set this value to 0.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain this function. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

SEE ALSO

aslistashs(3x)

cc(1), ld(1)

array_services(7), array_sessions(7)

NAME

asfreecmdrsltlist – Releases array command result structures

SYNOPSIS

```
#include <arraysvcs.h>
void asfreecmdrsltlist(ascmdrsltlist_t *CmdRsltList, uint32_t Flags);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The asfreecmdrsltlist function releases the resources used by the specified ascmdrsltlist_t structure. The ascommand(3x) function typically generates these structures.

The formal parameters are as follows:

CmdRsltList Specifies a pointer to the ascmdrsltlist_t structure whose resources are to be released.

Flags Specifies the resources to be released. The value of *Flags* is constructed from the logical OR of zero or more of the following flags. (If you do not specify a flag, set the value to 0.) The flags are as follows:

- ASFLF_FREEDATA — Releases the storage used by the individual ascmdrslt_t structure elements.
- ASFLF_UNLINK — Unlinks temporary files referenced by the ascmdrslt_t structure elements.
- ASFLF_CLOSEIO — Closes I/O sockets associated with the ascmdrslt_t structure elements.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain this function. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

SEE ALSO

ascommand(3x)

cc(1), ld(1)

array_services(7), array_sessions(7)

NAME

asfreemachinelist – Releases machine information structures

SYNOPSIS

```
#include <arraysvcs.h>
void asfreemachinelist(asmachinelist_t *MachineList, uint32_t Flags);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The asfreemachinelist function releases the resources used by the specified asmachinelist_t structure. The aslistmachines(3x) function typically generates these structures.

The formal parameters are as follows:

MachineList Specifies a pointer to the asmachinelist_t structure whose resources are to be released.

Flags Specifies the resources to be released. The *Flags* value can be one of the following:

ASFLF_FREEDATA	Releases the storage used by the individual asmachine_t structure elements.
0	Releases only the storage used by the asmachinelist_t structure.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain this function. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

SEE ALSO

aslistmachines(3x)

cc(1), ld(1)

array_services(7), array_sessions(7)

NAME

asfreemachinepidlist – Releases process identification enumeration structures

SYNOPSIS

```
#include <arraysvcs.h>
void asfreemachinepidlist(asmachinepidlist_t *PIDList, uint32_t Flags);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The asfreemachinepidlist function releases the resources used by the specified asmachinepidlist_t structure. The aspidsinash_server(3x) function typically generates these structures.

The formal parameters are as follows:

PIDList Specifies a pointer to the asmachinepidlist_t structure whose resources are to be released.

Flags [Reserved for future expansion.] Set this value to 0.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain this function. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

SEE ALSO

aspidsinash_server(3x)

cc(1), ld(1)

array_services(7), array_sessions(7)

NAME

asfreeoptinfo – Releases command line options information structure

SYNOPSIS

```
#include <arraysvcs.h>
void asfreeoptinfo(asoptinfo_t *OptInfo, uint32_t Flags);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The asfreeoptinfo function releases the resources used by the specified asoptinfo_t structure. The asparseopts(3x) function typically generates these structures.

The formal parameters are as follows:

OptInfo Specifies a pointer to the asoptinfo_t structure whose resources are to be released.

Flags Specifies the resources to be released. The *Flags* value can be one of the following:

ASFLF_CLOSESRV	Closes the server token in the *token* member if it is currently valid.
0	Releases only the storage used by the asoptinfo_t structure.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain this function. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

SEE ALSO

asparseopts(3x)

cc(1), ld(1)

array_services(7), array_sessions(7)

NAME

asfreepidlist – Releases process identification enumeration structures

SYNOPSIS

```
#include <arraysvcs.h>
void asfreepidlist(aspidlist_t *PIDList, uint32_t Flags);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The asfreepidlist function releases the resources used by the specified aspidlist_t structure. The aspidsinash_local(3x) and aspidsinash(3x) functions typically generate these structures.

The formal parameters are as follows:

PIDList Specifies a pointer to the aspidlist_t structure whose resources are to be released.

Flags [Reserved for future expansion.] Set this value to 0.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain this function. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

SEE ALSO

aspidsinash(3x),

cc(1), ld(1)

array_services(7), array_sessions(7)

NAME

asgetattr – Searches an attribute list for a particular name

SYNOPSIS

```
#include <arraysvcs.h>

const char *asgetattr(const char *attrname, const char **attrs,
int numattrs)
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The asgetattr function searches through a list of strings for a particular attribute name and returns a corresponding value, similar to the way getenv(3C) searches through the environment for a particular variable.

The formal parameters are as follows:

attrname Specifies the attribute to be found. Attributes are assumed to be of the format *NAME*=*VALUE*, so this amounts to searching the attributes for the first one that starts with *attrname* followed either by a null or the character =. If *NAME* is not found, asgetattr returns a null pointer. If *VALUE* is present, asgetattr returns a pointer to *VALUE*.

attrs Specifies the list of strings. This value is typically returned by a function such as aslistarrays(3x) or aslistmachines(3x).

numattrs Specifies the number of strings in the list.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain this function. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

RETURN VALUES

If no attribute with the specified *NAME* is found, asgetattr returns a null pointer. If *NAME* is found but has no corresponding *VALUE*, then asgetattr returns a pointer to a null character. Otherwise, asgetattr returns a pointer to the *VALUE* associated with *NAME*.

SEE ALSO

aslistarrays(3x), aslistmachines(3x), setenv(3C)

cc(1), ld(1)

array_services(7), array_sessions(7)

NAME

asgetdfltarray – Gets information about the default array

SYNOPSIS

```
#include <arraysvcs.h>
asarray_t* asgetdfltarray(asserver_t Server);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The asgetdfltarray function returns a description of the default array that the Array Services daemon uses for commands and other operations if no other array has been specified. The description is in the form of an asarray_t structure, which is defined in the arraysvcs.h file. The libarray library uses the malloc(3C) function to allocate storage for this structure. To release the storage space, use the asfreearray(3x) function.

The formal parameter is as follows:

Server Specifies an optional array server token, which can be used to direct the request to a specific Array Services daemon. If you specify a null pointer, the default Array Services daemon processes the request. For information on how the default Array Services daemon is selected, see the array(1) man page. For more details on creating an array server token, see the asopenserver(3x) man page.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain this function. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

RETURN VALUES

If successful, asgetdfltarray returns a pointer to an asarray_t structure. If unsuccessful. asgetdfltarray returns a null pointer and sets aserrorcode(3x) accordingly.

SEE ALSO

aserrorcode(3x), asfreearray(3x), aslistarrays(3x), asopenserver(3x) malloc(3C)

array(1), cc(1), ld(1)

array_services(7), array_sessions(7)

arrayd(8)

NAME

asin, asinf, asinl, acos, acosf, acosl, atan, atanf, atanl, atan2, atan2f, atan2l – Determines arcsine, arccosine, or arctangent of a value

SYNOPSIS

```
#include <math.h>
double asin (double x);
float asinf (float x);
long double asinl (long double x);
double acos (double x);
float acosf (float x);
long double acosl (long double x);
double atan (double x);
float atanf (float x);
long double atanl (long double x);
double atan2 (double x, double y);fR
float atan2f (float x, float y);
long double atan2l (long double x, long double y);
```

IMPLEMENTATION

All Cray Research systems (asin, acos, atan, atan2 only)
Cray PVP systems (asinl, acosl, atanl, atan2l only)
Cray MPP systems (asinf, acosf, atanf, atan2f only)

STANDARDS

ISO/ANSI (asin, acos, atan, atan2 only)
CRI extension (all others)

DESCRIPTION

The asin, asinf, and asinl functions return the arcsine of *x* in radians. A domain error occurs for arguments not in the range [–1,+1].

The acos, acosf, and acosl functions return the arccosine of *x* in radians. A domain error occurs for arguments not in the range [–1,+1].

The atan, atanf, and atanl functions return the arctangent of *x* in radians. A domain error occurs if both arguments are 0.

The atan2, atan2f, and atan2l functions return the arctangent of *x/y*.

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

When code containing calls to any of 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. 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, ASIN(3M)).

RETURN VALUES

The return values for the acos, acosf, and acosl functions are in the range [0,π] radians. The return values for the asin, asinf, asinl, atan, atanf, and atanl functions are in the range [−π/2,+p/2] radians. The return values for the atan2, atan2f, and atan2l functions are in the range [−π,+π] radians. The signs of both arguments are used to determine the quadrant of the return value.

When a program is compiled with -hstdc or -hmatherror=errno on Cray MPP systems and CRAY T90 systems with IEEE arithmetic, the following functions return NaN and set errno to EDOM when called with the specified parameters: acos(+/- *infinity*), acosl(+/- *infinity*), asin(+/- *infinity*), asinl(+/- *infinity*), acos(*NaN*), acosl(*NaN*), asin(*NaN*), asinl(*NaN*), atan(*NaN*), atanl(*NaN*), atan2(*NaN*, *x*), atan2(*y*, *NaN*), atan21(*NaN*, *x*), and atan21(*y*, *NaN*).

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
acos(*x*), where *x* is not in the range [-1,1]	0	NaN
acosl(*x*), where *x* is not in the range [-1,1]	0	NaN
acosf(*x*), where *x* is not in the range [-1,1]	0	NaN
asin(.0+0.0*1.0*i*), where *x* is not in the range [-1,1]	0	NaN
asinl(*x*), where *x* is not in the range [-1,1]	0	NaN
asinf(*x*), where *x* is not in the range [-1,1]	0	NaN
atan2(0.0, 0.0)	0	NaN
atan2f(0.0, 0.0)	0	NaN

Error	CRI_IEEE_LIB=0	CRI_IEEE_LIB=1
atan2l(0.0, 0.0)	0	NaN

SEE ALSO

errno.h(3C)

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

NAME

askillash_array, askillash_local, askillash_server – Sends a signal to an array session

SYNOPSIS

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

int askillash_array(asserver_t Server, const char *ArrayName,
ash_t ASH, int Sig);

int askillash_local(ash_t ASH, int Sig);

int askillash_server(asserver_t Server, ash_t ASH, int Sig);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The askillash_array, askillash_local, and askillash_server functions all send a signal to each of the processes that belong to the array session specified by the array session handle value *ASH* at the moment the function is executed.

The formal parameters are as follows:

Server Specifies an optional array server token for the askillash_array and askillash_server functions. This token can be used to direct the request to a specific Array Services daemon. If you specify a null pointer, the request is processed by the default Array Services daemon. For information on selecting the default Array Services daemon, see the array(1) man page. For information on creating an array server token, see the asopenserver(3x) man page.

ArrayName Specifies the array name.

ASH Specifies the array session handle.

Sig Specifies the signal to be sent. The signal is either one from the list given in signal(2) or 0. If Sig is 0 (the null signal), error checking is performed but no signals are actually sent. This can be used to check the validity of *ASH*.

The real or effective user ID of the sending process must match the real, saved, or effective user ID of the receiving processes, unless the effective user ID of the sending process is that of the superuser.

The askillash_array function sends a signal to the members of the specified array session on each of the machines in the array specified by *ArrayName*, or the default array if *ArrayName* is a null pointer. The Array Services daemon specified by the server token *Server* coordinates the operation.

The askillash_local function only sends a signal to the members of the specified array session that are running on the same machine as the one that executes askillash_local. Unlike askillash_array and askillash_server, this function does not require the Array Services daemon.

The askillash_server function only sends a signal to the members of the specified array session that are running on the machine specified with the server token Server.

All three functions will fail if one or more of the following are true:

- *Sig* is not a valid signal number.
- *Sig* is SIGKILL and the specified array session contains process 1.
- The user ID of the sending process is not that of the superuser, and its real or effective user ID does not match the real, saved, or effective user ID of the receiving processes.
- The Array Services daemon is not currently active (not applicable for askillash_local).

If one or more of these conditions apply only to a subset of the processes in an array session, it is undefined whether or not these functions will complete for some or all of processes that are **not** affected.

These functions are **not** atomic with respect to process creation. As a result, it is possible that a new process could join the array session after the signaling operation has started but before it has completed. Consequently, the process would never receive the signal itself.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain this function. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

RETURN VALUES

If successful, the askillash_array, askillash_local, and askillash_server functions return a value of 0. If unsuccessful, they return a value of –1 and set aserrorcode(3x) accordingly.

SEE ALSO

aserrorcode(3x), askillpid_server(3x), asopenserver(3x)

array(1), cc(1), ld(1)

kill(2)

array_services(7), array_sessions(7)

arrayd(8)

NAME

`askillpid_server` – Sends a signal to a remote process

SYNOPSIS

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

int askillpid_server(asserver_t Server, pid_t PID, int Sig);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The `askillpid_server` function sends a signal to the process specified by the value *PID*. Depending on the server token *Server*, the process specified by *PID* does not necessarily have to reside on the same machine as the one executing `askillpid_server`.

The real or effective user ID of the sending process must match the real, saved, or effective user ID of the receiving process, unless the effective user ID of the sending process is that of the superuser.

The formal parameters are as follows:

Server Specifies an optional array server token, which can be used to direct the request to a specific machine. If you specify a null pointer, the default Array Services daemon processes the request. For information on selecting the default Array Services daemon, see the `array`(1) man page. For information on creating an array server token, see the `asopenserver`(3x) man page.

PID Specifies the process identification number.

Sig Specifies the signal to be sent. *Sig* is either one from the list given in `signal`(2) or 0. If *Sig* is 0 (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *PID*.

The `askillpid_server` function will fail if one or more of the following are true:

- *Sig* is not a valid signal number.
- *Sig* is `SIGKILL` and *PID* is process 1.
- The user ID of the sending process is not that of the superuser, and its real or effective user ID does not match the real, saved, or effective user ID of the receiving process.
- The Array Services daemon (`arrayd`) is not currently active on the machine specified by *Server*.

NOTES

The IRIX `libarray.so` and the UNICOS `libarray.a` libraries contain this function. You can load the `libarray.so` or `libarray.a` library by using the `-larray` option with `cc`(1) or `ld`(1).

RETURN VALUES

If successful, `askillpid_server` returns a value of 0. If unsuccessful, `askillpid_server` returns a value of –1 and sets `aserrorcode(3x) accordingly.`

SEE ALSO

`aserrorcode`(3x), `askillash_server`(3x), `asopenserver`(3x)

`array`(1), `cc`(1), `ld`(1)

`kill`(2)

`array_services`(7), `array_sessions`(7)

`arrayd`(8)

NAME

aslistarrays – Enumerates known arrays

SYNOPSIS

```
#include <arraysvcs.h>
asarraylist_t *aslistarrays(asserver_t Server);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The aslistarrays function returns a list of all arrays that are known to the specified Array Services daemon. The machine invoking this function may or may not be a member of one or more of those arrays.

The formal parameter is as follows:

Server Specifies an optional array server token, which can be used to direct the request to a specific Array Services daemon. If you specify a null pointer, the default Array Services daemon processes the request. For information on how the default Array Services daemon is selected, see the array(1) man page. For information on creating an array server token, see the asopenserver(3x) man page.

Each array is described by an asarray_t structure, and the entire list is contained in an asarraylist_t structure. Both of these are defined in the arraysvcs.h file. The libarray library uses the malloc(3C) function to allocate storage for these structures. To release the storage space, use the asfreearraylist(3x) function.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain this function. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

RETURN VALUES

If successful, aslistarrays returns a pointer to an asarraylist_t structure. If unsuccessful, aslistarrays returns a null pointer and sets aserrorcode(3x) accordingly.

SEE ALSO

aserrorcode(3x), asfreearraylist(3x), aslistmachines(3x), asopenserver(3x), malloc(3C)

array(1), cc(1), ld(1)

array_services(7), array_sessions(7)

arrayd(8)

NAME

aslistashs, aslistashs_array, aslistashs_local, aslistashs_server – Enumerates array session handles

SYNOPSIS

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

asashlist_t *aslistashs(asserver_t Server, const char *ArrayName,
int Destination, uint32_t Flags);

asashlist_t *aslistashs_array(asserver_t Server, const char *ArrayName);

asashlist_t *aslistashs_local(void);

asashlist_t *aslistashs_server(asserver_t Server);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The aslistashs function returns a list of array session handles that are currently active on the local machine, some other machine, or some array.

The formal parameters are as follows:

Server Specifies an optional array server token, which can be used to direct the request to a specific Array Services daemon. If you specify a null pointer, the default Array Services daemon processes the request if necessary. For information on how the default Array Services daemon is selected, see the array(1) man page For information on creating an array server token, see the asopenserver(3x) man page.

ArrayName Specifies the array name.

Destination Specifies the target of aslistashs. *Destination* may have one of the following values:

- ASDST_ARRAY — Retrieves the active array session handles on all machines in the array specified by *ArrayName*, or the default array if *ArrayName* is a null pointer.
- ASDST_LOCAL — Retrieves the active array session handles on the local machine only.
- ASDST_SERVER — Retrieves the active array session handles on the machine specified by *Server* only.

If *Destination* is not ASDST_ARRAY, the *ArrayName* value should be a null pointer.

Flags Controls some of the details about the array session handles that are returned. The *Flags* value is constructed from a logical OR of zero or more of the following flags. (If you do not specify a flag, set the value to 0.) The flags are as follows:

ASLAF_NOLOCAL	Does not include local array session handles.
ASLAF_NODUPS	Causes duplicate array session handles to be removed from the list, with some additional cost in execution time. Ordinarily, an array session handle may appear more than once in the returned list.

The list of array session handles is returned in an asashlist_t structure, which is defined in the arraysvcs.h file. The libarray library uses the malloc(3C) function to allocate storage for this structure. To release the storage space, use the asfreeashlist(3x) function.

The aslistashs_array, aslistashs_local and aslistashs_server functions are convenience functions that are equivalent to variations of aslistash:

- aslistashs_array(*Server*, *ArrayName*) is equivalent to:

```
aslistashs(Server, ArrayName, ASDST_ARRAY,
                (ASLAF_NOLOCAL | ASLAF_NODUPS))
```

- aslistashs_local() is equivalent to:

```
aslistashs(NULL, NULL, ASDST_LOCAL, ASLAF_NODUPS)
```

- aslistashs_server(*Server*) is equivalent to:

```
aslistashs(Server, NULL, ASDST_SERVER,
                (ASLAF_NOLOCAL | ASLAF_NODUPS))
```

Because array sessions are transient, this information cannot be completely accurate; it may omit some new array sessions and/or include array sessions that have already terminated.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain this function. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

RETURN VALUES

If successful, the aslistashs, aslistashs_array, aslistashs_local, and aslistashs_server functions return a pointer to an asashlist_t structure. If unsuccessful, they return a null pointer and set aserrorcode(3x) accordingly.

SEE ALSO

asashisglobal(3x), aserrorcode(3x), asfreeashlist(3x), asopenserver(3x) malloc(3C)
array(1), cc(1), ld(1)
array_services(7), array_sessions(7)
arrayd(8)

NAME

aslistmachines – Enumerates machines in an array

SYNOPSIS

```
#include <arraysvcs.h>
asmachinelist_t *aslistmachines(asserver_t Server, const char *Name);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The aslistmachines function returns a list of the machines that are members of the array specified by *Name*. If *Name* is a null pointer, a list of the machines that are members of the default array is returned.

The formal parameters are as follows:

Server Specifies an optional array server token, which can be used to direct the request to a specific Array Services daemon. If you specify a null pointer, the request is processed by the default Array Services daemon. For information on selecting the default Array Services daemon, see the array(1) man page. For information on creating an array server token, see the asopenserver(3x) man page.

Name Specifies the name of the array for which a list of member machines is returned.

An asmachine_t structure describes each machine, and an asmachinelist_t structure contains the entire list. The arraysvcs.h file defines these structures. The libarray library uses the malloc(3C) function to allocate storage for these structures. To release the storage space, use the asfreemachinelist(3x) function.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain this function. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

RETURN VALUES

If successful, aslistmachines returns a pointer to an asmachinelist_t structure. If unsuccessful, aslistmachines returns a null pointer and sets aserrorcode(3x) accordingly.

SEE ALSO

aserrorcode(3x), asfreemachinelist(3x), aslistarrays(3x), asopenserver(3x), malloc(3C)

array(1), cc(1), ld(1)

array_services(7), array_sessions(7)

arrayd(8)

NAME

asmakeerror – Generates an Array Services error code

SYNOPSIS

```
#include <arraysvcs.h>
aserror_t asmakeerror(int Errno, int What, int Why, int extra);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The asmakeerror function combines the various fields of an Array Services error code into a single value. The global variable *aserrorcode* contains these values.

The formal parameters are as follows:

Errno Specifies the error number.

What Specifies the type of error.

Why Specifies why this is an error.

Extra Specifies other information.

The arraysvcs.h file describes the specific values that are typically stored in these fields. No validation is done on the values of the individual fields or of the resulting error code.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain this function. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

RETURN VALUES

The asmakeerror function always returns a value of type aserror_t that is composed of the specified fields.

SEE ALSO

aserrorcode(3x)

cc(1), ld(1)

array_services(7), array_sessions(7)

arrayd(8)

NAME

asopenserver, ascloseserver – Creates or destroys an array server token

SYNOPSIS

```
#include <arraysvcs.h>

asserver_t asopenserver(const char *ServerName, int PortNumber);
void ascloseserver(asserver_t Server);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The asopenserver function creates an array server token. You can use this token with other libarray functions to direct Array Services requests to a specific Array Services daemon.

The formal parameters are as follows:

ServerName Specifies the host name of the machine to which Array Services requests made with this token should be directed. If you specify a null pointer, the default Array Services host processes the request.

PortNumber Specifies the network port number of the Array Services daemon on the specified machine. If you specify -1, the default port number is used.

For information on determining the default Array Services host and port number, see the array(1) man page.

You should use the ascloseserver function to destroy the array server token specified by *Server* when it is no longer needed. This releases the resources that it is using.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain this function. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

RETURN VALUES

If successful, asopenserver returns a nonzero array server token. If unsuccessful, asopenserver returns a null pointer and sets aserrorcode(3x) accordingly.

SEE ALSO

aserrorcode(3x), assetserveropt(3x)

cc(1), ld(1)

array_services(7), array_sessions(7)

arrayd(8)

NAME

asopenserver_from_optinfo – Creates an array server token

SYNOPSIS

```
#include <arraysvcs.h>
asserver_t asopenserver_from_optinfo(const asoptinfo_t *Info);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The asopenserver_from_optinfo function creates and modifies an array server token using parameters taken from an asoptinfo_t structure. You can use the resulting array server token with other libarray functions to direct Array Services requests to a specific Array Services daemon. For further details, see asopenserver(3x) and assetserveropt(3x).

The formal parameter is as follows:

Info Points to an asoptinfo_t structure that contains all of the relevant information needed to create the server token and optionally set various options pertaining to it. Typically, you will generate asoptinfo_t structures from a list of command line arguments by using the asparseopts(3x) function; however, you can also generate asoptinfo_t structures manually. asopenserver_from_optinfo uses only members that have been marked as valid in the asoptinfo_t structure. For more information about asoptinfo_t structures, see the asparseopts(3x) man page.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain this function. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

RETURN VALUES

If successful, asopenserver_from_optinfo returns a nonzero array server token. If unsuccessful, asopenserver_from_optinfo returns a null pointer and sets aserrorcode(3x) accordingly.

SEE ALSO

aserrorcode(3x), asopenserver(3x), asparseopts(3x), assetserveropt(3x)

cc(1), ld(1)

array_services(7), array_sessions(7)

arrayd(8)

NAME

asparseopts – Parses standard Array Services command line options

SYNOPSIS

```
#include <arraysvcs.h>
asoptinfo_t *asparseopts(int Argc, char **Argv, int Select, int Control);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The asparseopts function parses standard Array Services command line options from a list of strings, typically the list of arguments to an Array Services client program. The results are returned in the form of an asoptinfo_t structure, which contains parsed, validated values for the options specified in the argument list, and a list of the arguments that were not recognized as one of the selected Array Services options. The libarray library uses the malloc(3C) function to allocate storage for this structure. To release the storage space, use the asfreeoptinfo(3x) function.

The formal parameters are as follows:

Argc Specifies an argument count in the form typically provided to the function main of an ordinary C program.

Argv Specifies an argument list in the form typically provided to the function main of an ordinary C program.

Select Specifies which of the standard array service command line options are to be included in this operation. It is constructed from the logical OR of one or more of the following flags, which are defined in the arraysvcs.h file:

ASOIV_ARRAY Parses the -array option, which takes the name of an array as a subargument. -a is a synonym for the -array option.

ASOIV_ASH Parses the -ash option, which takes an array session handle as a subargument. The array session handle may be specified in decimal, octal (if preceded by 0) or hexadecimal (if preceded by 0x). -h and -arsess are both synonyms for the -ash option.

ASOIV_CONNECTTO Parses the -connectto option, which takes a connection timeout value as a subargument. The value must be specified in decimal only. -C is a synonym for the -connectto option.

ASOIV_FORWARD Parses the -forward and -direct options, which specify the Array Services forwarding mode:

- The `-forward` option indicates that Array Services commands should be forwarded to their ultimate destination through the server on the local machine.
- The `-direct` option indicates that Array Services commands should be sent directly to the remote server.

`-F` is a synonym for the `-forward` option and `-D` is a synonym for the `-direct` option.

`ASOIV_LCLKEY` Parses the `-localkey` option, which takes the authentication key for the local machine as a subargument. `-Kl` is a synonym for the `-localkey` option.

`ASOIV_LOCAL` Parses the `-local` option, which indicates that an Array Services function should take place only on the local server, as opposed to being broadcast to all of the servers in an array (for example). `-l` is a synonym for the `-local` option.

`ASOIV_PID` Parses the `-pid` option, which takes a process identification number (PID) as a subargument. The PID should be specified in decimal only and must be positive. `-i` and `-process` are both synonyms for the `-pid` option.

`ASOIV_PORTNUM` Parses the `-portnum` option, which takes a port number as a subargument. The port number should be specified in decimal only and must be in the range 1 through 65535. `-p` is a synonym for the `-portnum` option.

`ASOIV_REMKEY` Parses the `-remotekey` option, which takes the authentication key for a remote machine as a subargument. `-Kr` is a synonym for the `-remotekey` option.

`ASOIV_SERVER` Parses the `-server` option, which takes the hostname of an array daemon as a subargument. `-s` is a synonym for the `-server` option.

`ASOIV_TIMEOUT` Parses the `-timeout` option, which takes a timeout value as a subargument. The value must be specified in decimal only. `-t` is a synonym for the `-timeout` option.

`ASOIV_TOKEN` Creates a server token using the parsed options, by using `asopenserver_from_optinfo`(3x), assuming that no invalid arguments were encountered (in other words, the `invalid` member of the returned `asoptinfo_t` structure is 0).

ASOIV_VERBOSE — Parse the `-v` option, which is used to set a verbosity level. The default verbose level is 0, and each occurrence of `-v` increases the level by 1. If an option begins with `v` and is followed by any number of other non-whitespace characters (for example, `-vvv`), then the verbose level is increased by the number of characters following the hyphen (three in the case of `-vvv`).

`Control` Specifies flags that modify the parsing behavior. This value is constructed from the logical OR of zero or more of the following flags, which are defined in the `arraysvcs.h` file. (If you do not specify a flag, set the value to 0 so that no modification takes place.) The flags are as follows:

ASOIC_LOGERRS — Specifies that syntax errors and other abnormal conditions should be reported to the normal Array Services error logging destination, which is typically standard error. You must specify this flag to generate error messages. However, if you do not specify this flag, the `invalid` member of the returned `asoptinfo_t` structure can still be checked to determine if any errors were detected.

ASOIC_NODUPS — Calls out duplicate occurrences of an option as errors and marks the option as invalid in the returned `asoptinfo_t`. Ordinarily, if an option is specified more than once, the last occurrence of the option in the argument list quietly overrides previous occurrences of the option.

ASOIC_OPTSONLY — Stops parsing as soon as an argument that does not begin with a `-` character is encountered (not including subarguments to valid options). The `non-option` argument and all arguments following it are returned as unrecognized arguments, even if some of the subsequent arguments would otherwise have been valid Array Services options.

ASOIC_SELONLY — Stops parsing as soon as an argument that is not a selected option or the subargument of a selected option is encountered.

If the argument list is successfully parsed, a pointer to an `asoptinfo_t` structure (also defined in the `arraysvcs.h` file) is returned. An `asoptinfo_t` structure has the following format:

```
typedef struct asoptinfo {
        int         argc;
        char        **argv;
        int         valid;
        int         invalid;
        int         options;
        asserver_t  token;
        char        *server;
        char        *array;
        askey_t     localkey;
        askey_t     remotekey;
        ash_t       ash;
        pid_t       pid;
        int         portnum;
        int         timeout;
        int         connectto;
        int         verbose;
} asoptinfo_t;
```

The members are as follows:

argc Specifies the count of arguments that were not recognized as selected Array Services options or their corresponding subarguments.

argv Specifies the list of arguments that were not recognized as selected Array Services options or their corresponding subarguments.

valid Specifies a bitmap used to specify which options were successfully parsed and are present in the `asoptinfo_t` structure. The same flags used to specify the *Select* argument to `asparseopts` are used to indicate which options are present.

invalid Specifies a bitmap of options that were selected and specified in the argument list, but had values that were invalid in some way. If the `ASOIC_LOGERRS` control flag was specified, then an error message explaining the nature of the problem should already have been generated. This member also uses the same flags as *valid* and *Select*.

options Specifies a bitmap of flags indicating the state of the various binary options.

A flag in `options` should only be examined if it is also marked as valid in *valid*. For example, the state of the `ASOIO_FORWARD` flag in `options` is only meaningful if the `ASOIV_FORWARD` flag is set in *valid*. If the appropriate flag in *valid* is **not** set, then the option should be considered unspecified and a default setting should be used instead. The flags that may be set are as follows:

`ASOIO_FORWARD` Sets command forwarding. If not set, a direct connection is desired.

`ASOIO_LOCAL` Restricts the command to the local server. If not set, the command is considered eligible for broadcast to all servers in an array.

token Specifies a server token. This member is not a value directly parsed from the argument list, but instead a server token created using the values that were successfully parsed from the argument list. It is only created if the ASOIV_TOKEN flag was set in *Select*. If it is successfully created, the ASOIV_TOKEN flag is set in the *valid* member of the asoptinfo_t structure. Otherwise, ASOIV_TOKEN is set in the *invalid* member and aserrorcode(3x) is set accordingly.

The remaining members of the asoptinfo_t structure contain the values of the selected Array Services options. If a selected option was specified in the argument list, then its flag in *valid* is set and the corresponding member of asoptinfo_t structure contains the parsed value of that option. If a selected option was *not* specified in the argument list, then its flag in *valid* is not set and the corresponding member of asoptinfo_t structure contains a default value (generally a null pointer, 0 or –1, as appropriate). If a selected option had an invalid value, its flag is set in *invalid* and the contents of the corresponding member of asoptinfo_t structure are unpredictable. The remaining members are as follows:

server	Specifies the server name.
array	Specifies the array name.
localkey	Specifies the local key.
remotekey	Specifies the remote key.
ash	Specifies the array session handle.
pid	Specifies the process identification number.
portnum	Specifies the port number.
timeout	Specifies the timeout value.
connectto	Specifies the connection timeout value.
verbose	Specifies the verbose level.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain this function. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

RETURN VALUES

If successful, asparseopts returns a pointer to an asoptinfo_t structure.

If asparseopts is successful and the ASOIV_TOKEN flag of *Select* was specified but a server token could not be created, asparseopts returns the pointer to the asoptinfo_t structure as usual, but sets the ASOIV_TOKEN flag of the *invalid* member and sets aserrorcode so that it contains the error returned by asopenserver_from_optinfo(3x).

If a severe error occurs, asparseopts returns a null pointer and sets aserrorcode accordingly.

SEE ALSO

aserrorcode(3x), asfreeoptinfo(3x), asopenserver_from_optinfo(3x), malloc(3C)

cc(1), ld(1)

array_services(7), array_sessions(7)

arrayd(8)

NAME

asperror – Prints an Array Services error message

SYNOPSIS

```
#include <arraysvcs.h>
void asperror(const char *Format, .../* args */);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The asperror produces a message on the standard error output (file descriptor 2) that describes the last error encountered during a call to certain Array Services functions.

The error is determined from the external variable aserrorcode, which is set by many Array Services functions when errors occur.

The formal parameter is as follows:

Format Specifies a format string that is treated as a format string similar to an IRIX printf(3S) or UNICOS printf(3C) string and is printed first, followed by a colon and a blank, then the message and a newline. (However, if *Format* is a null pointer or points to a null string, the colon is not printed.) Arguments needed to satisfy any conversion specifications in *Format* should follow *Format* in the function invocation.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain this function. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

SEE ALSO

aserrorcode(3x), asstrerror(3x), IRIX printf(3S), UNICOS printf(3C)

cc(1), ld(1)

array_services(7), array_sessions(7)

NAME

aspidsinash, aspidsinash_array, aspidsinash_local, aspidsinash_server – Enumerates processes in an array session

SYNOPSIS

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

aspidlist_t *aspidsinash(ash_t ASH);

asarraypidlist_t *aspidsinash_array(asserver_t Server,
const char *ArrayName, ash_t ASH);

aspidlist_t *aspidsinash_local(ash_t ASH);

asmachinepidlist_t *aspidsinash_server(asserver_t Server, ash_t ASH);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The aspidsinash, aspidsinash_array, aspidsinash_local, and aspidsinash_server functions return lists of process identification (PID) numbers that belong to the array session specified by the array session handle *ASH*. Each function returns its list in a data structure that is defined in the arraysvcs.h file. The libarray library uses the malloc(3C) function to allocate storage for these structures. When the space is no longer needed, release it with the appropriate function noted below.

The formal parameters are as follows:

ASH Specifies the array session handle.

Server Specifies an optional array server token, which can be used to direct the request to a specific Array Services daemon. If you specify a null pointer, the request is processed by the default Array Services daemon if necessary. For information on how the default Array Services daemon is selected, see the array(1) man page. For information on creating an array server token, see the asopenserver(3x) man page.

ArrayName Specifies the array name.

The functions return the following information:

`aspidsinash_local` Returns only those processes in the array session that are running on the local machine. `aspidsinash_local` returns the list of PIDs in an `aspidlist_t` structure, which you can free by using `asfreepidlist`(3x). `aspidsinash` is the same as `aspidsinash_local`, and is retained mainly for backward compatibility. Unlike the remaining functions, `aspidsinash_local` and `aspidsinash` do not require the Array Services daemon to be running in order to complete successfully.

`aspidsinash_server` Returns the list of processes in the specified array session that are running on the machine specified by *Server*. `aspidsinash_server` returns the list in a `asmachinepidlist_t` structure, which you can free by using `asfreemachinepidlist`(3x).

`aspidsinash_array` Returns a list of processes in the specified array session for all of the machines in the array specified by `ArrayName`. `aspidsinash_array` returns the data in the form of an `asarraypidlist_t` structure, which you can free by using `asfreearraypidlist`(3x). The `asarraypidlist_t` structure in turn contains pointers to one or more `asmachinepidlist_t` structures, one for each machine in the array. Each `asmachinepidlist_t` structure contains the name of the particular machine and a list of the processes that (in the specified array session) that are running on the machine.

NOTES

Because processes and array sessions are transient, this information cannot be completely accurate; it may omit some new processes and/or include processes that have already terminated.

The IRIX `libarray.so` and the UNICOS `libarray.a` libraries contain these functions. You can load the `libarray.so` or `libarray.a` library by using the `-larray` option with `cc`(1) or `ld`(1).

The Array Services daemon, `arrayd`(8), must be running on all affected machines for the functions `aspidsinash_array` and `aspidsinash_server` to work properly.

RETURN VALUES

If successful, `aspidsinash` returns a pointer to an `aspidlist_t` structure. If unsuccessful, `aspidsinash` returns a null pointer and sets `aserrorcode`(3x) accordingly.

SEE ALSO

asashisglobal(3x), aserrorcode(3x), asfreearraypidlist(3x), asfreemachinepidlist(3x), asfreepidlist(3x), aslistashs_server(3x), malloc(3C)

cc(1), ld(1)

array_services(7), array_sessions(7)

arrayd(8)

NAME

asrcmd, asrcmdv – Executes a command on a remote machine

SYNOPSIS

```
#include <arraysvcs.h>

int asrcmd(asserver_t Server, char *User, char *CmdLine, int *fd2p);
int asrcmdv(asserver_t Server, char *User, char **CmdV, int *fd2p);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The asrcmd and asrcmdv functions execute a command on a remote machine. They are similar in some respects to IRIX rcmd(3N) and UNICOS rcmd(3C) except that the connection and user authentication is provided by Array Services, so the user does not need root privileges. Both asrcmd and asrcmdv pass the command to the remote user's default shell for execution using the standard shell command line option -c. For example, if the requested command is ls -l and the remote user's shell is /bin/tcsh, then the following command would be invoked on the remote machine:

```
/bin/tcsh -c "ls -l"
```

The only difference between asrcmd and asrcmdv is in the way that the remote command is specified.

The formal parameters are as follows:

Server Specifies an array server token created with asopenserver(3x) that specifies the remote machine that is to execute the command. If you specify a value of a null pointer, the command is executed on the same machine as the one running the default Array Services daemon, although this is not generally very useful. For information on how the default Array Services daemon is selected, see the array(1) man page.

User Specifies the login name of the user on the remote machine that should execute the command. Specifying a null pointer executes the command using the same user login name as the one executing asrcmd or asrcmdv. Authorization for the local user to execute commands as the user specified by the *User* value on the remote machine is determined with IRIX ruserok(3N), the same mechanism used by rsh(1) and IRIX rcmd(3N) and UNICOS rcmd(3C) that involves checking for the user in /etc/hosts.equiv and/or ~/.rhosts.

CmdLine Specifies a single string containing the entire command to be executed, such as it might be typed on the command line.

CmdV Specifies an array of string pointers (similar to that used with `argv`) that contains the list of arguments that make up the command to be executed. The array should be terminated with a null pointer. The list of arguments is concatenated into a single string (with a single space between each) before it is passed to the remote user's default shell for execution. It may therefore be necessary to include appropriate shell quote characters if individual arguments contain embedded space or tab characters.

If the remote command is successfully initiated, a socket in the internet domain of type `SOCK_STREAM` is returned to the caller and given to the remote command as `stdin` and `stdout`. If `fd2p` is nonzero, then an auxiliary channel to a control process is set up, and a descriptor for it is placed in `*fd2p`. The control process returns output from the command's standard error, and also accepts bytes on this channel as being IRIX or UNICOS signal numbers. These signal numbers are to be forwarded to the process group of the command. If `fd2p` is 0, then the standard error of the remote command is made the same as the standard output and no provision is made for sending arbitrary signals to the remote process.

NOTES

The IRIX `libarray.so` and the UNICOS `libarray.a` libraries contain these functions. You can load the `libarray.so` or `libarray.a` library by using the `-larray` option with `cc`(1) or `ld`(1).

RETURN VALUES

If successful, `asrcmd` and `asrcmdv` return a socket descriptor attached to the remote command's standard input and standard output. If the remote command cannot be started, `asrcmd` and `asrcmdv` return a value of –1 and set `aserrorcode` accordingly.

SEE ALSO

`ascommand`(3x), `aserrorcode`(3x), `asopenserver`(3x), IRIX `rcmd`(3N), UNICOS `rcmd`(3C), IRIX `ruserok`(3N)

`array`(1), `cc`(1), `ld`(1) `rsh`(1)

`array_services`(7), `array_sessions`(7)

`arrayd`(8)

NAME

`assert` – Verifies program assertion

SYNOPSIS

```
#include <assert.h>
void assert (int expression);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The `assert` macro is useful in debugging programs. If you want to check if a certain condition is true at a certain point in the program, you can do so by stating that condition as the argument (*expression*) to the `assert` macro at that point. When the macro is executed, if that condition is false (0), `assert` prints the following on the standard error file and aborts:

`Assertion failed:` *expression*`, file` *xyz*`, line` *nnn*

In the error message, *xyz* is the name of the source file and *nnn* is the source line number of the `assert` statement.

When the first false assertion is encountered in the executing program, the program aborts after the printed message. With this facility, it is not possible to get more than one failed assertion message in one run of the program.

The `assert` macro can be disabled by defining the macro `NDEBUG` prior to the `#include <assert.h>`. In this case, the `assert` macro expands to `((void) 0))` and the argument passed to assert will not be evaluated. On the other hand, if the `assert` facility is enabled by the absence of `NDEBUG`, the `assert` macro expands to code to evaluate the argument and test the assertion. Therefore, whether the argument is evaluated depends on whether `NDEBUG` is defined; when using `assert`, statements following the call to `assert` should not depend on side effects of evaluation of the argument.

By default, `NDEBUG` is not defined, so all `assert` macros in the compiled program are enabled. To globally disable the `assert` macro, you can include option `-DNDEBUG` on the `cc` command line. Alternatively, you can include a `#define NDEBUG` in your source code prior to the `#include <assert.h>`.

If you want to selectively enable and disable the `assert` facility in parts of the program, include two lines in your source code each time you want to toggle the facility:

To disable:

```
#define NDEBUG
#include <assert.h>
```

To enable:

```
#undef NDEBUG
#include <assert.h>
```

`assert` is implemented only as a macro. If `#undef` is used to remove the macro definition from `assert` and obtain access to the underlying function, the behavior is undefined.

RETURN VALUES

The `assert` macro does not return a value.

SEE ALSO

`abort`(3C)

NAME

`assert.h` – Library header for diagnostic functions

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

TYPES

None

MACROS

The header `assert.h` defines the `assert` macro and refers to the `NDEBUG` macro, which is not defined by `assert.h`.

If `NDEBUG` is defined as a macro name at the point in the C source file where `assert.h` is included, the `assert` macro is defined as follows (that is, expands to a `void` expression that does nothing):

```
#define assert(ignore) ((void) 0)
```

The `assert` header is one case where multiple inclusions of a header can, by design, give different results than a single inclusion. See the description of the `assert` function.

If `#undef` is used to remove the macro definition from the `assert` macro and obtain access to the underlying function, the behavior is undefined.

FUNCTION DECLARATIONS

None

NAME

assetserveropt, asgetserveropt, asdfltserveropt – Sets or retrieves server options

SYNOPSIS

```
#include <arraysvcs.h>

int assetserveropt(asserver_t Server, int OptName,
const void *OptVal, int OptLen);

int asgetserveropt(asserver_t Server, int OptName,
void *OptVal, int OptLen);

int asdfltserveropt(int OptName, void *OptVal, int OptLen);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The asgetserveropt and assetserveropt functions manipulate options associated with the server token *Server*. The asdfltserveropt function retrieves the standard default value for those options when a new server token is created using asopenserver(3x).

The formal parameters are as follows:

Server Specifies the server name.

OptName Specifies the option to be manipulated. The *OptName* value may be one of the following (defined in the arraysvcs.h file):

AS_SO_TIMEOUT — Sets or retrieves the timeout value (in seconds) for a response to a request made to the Array Services daemon associated with the server token. The timeout value is of type int.

AS_SO_CTIMEOUT — Sets or retrieves the timeout value (in seconds) for establishing an initial connection with the Array Services daemon associated with the server token. The timeout value is of type int.

AS_SO_FORWARD — Sets or retrieves the state of the forwarding flag associated with the server token. If the flag is nonzero, then any requests made with the token are forwarded to the server associated with the token via the Array Services daemon at the default port on the local machine. If the flag is 0, requests are sent directly to the server associated with the token. The default setting of this flag is 0 unless the environment variable ARRAYD_FORWARD has a value beginning with the letter Y (as in "yes", in either uppercase or lowercase) at the time the token was created. The value of the flag is of type int.

AS_SO_LOCALKEY	Sets or retrieves the authentication key that is used for any messages sent to the Array Services daemon associated with the server token. The default value of this key is obtained from the environment variable ARRAYD_LOCALKEY, if it exists, or otherwise is set to 0. The key is of type askey_t.
AS_SO_REMOTEKEY	Sets or retrieves the authentication key that is used for any messages received from the Array Services daemon associated with the server token. The default value of this key is obtained from the environment variable ARRAYD_REMOTEKEY, if it exists, or otherwise is set to 0. The key is of type askey_t.
AS_SO_PORTNUM	Retrieves the port number of the default Array Services daemon. This value is obtained from the environment variable ARRAYD_PORT, if it exists, otherwise the port number associated with the service sgi-arrayd is used. This value is only valid with asdfltserveropt.
AS_SO_HOSTNAME	Retrieves the hostname of the default Array Services daemon. This value is obtained from the environment variable ARRAYD, if it exists, otherwise localhost is used. This value is only valid with asdfltserveropt.

OptVal Specifies an option value for assetserveropt. For asgetserveropt and asdfltserveropt, identifies the buffer in which the value for the requested option is to be returned.

OptLen Specifies the length of an option for assetserveropt. For asgetserveropt and asdfltserveropt, identifies the length of the buffer in which the value for the requested option is to be returned. For those functions, *OptLen* is a value-result parameter, initially containing the size in bytes of the buffer pointed to by *OptVal*, and modified on return to indicate the actual size of the value returned.

RETURN VALUES

If successful, the assetserveropt, asgetserveropt, and asdfltserveropt functions return a value of 0. If unsuccessful, these functions return a value of –1 and set aserrorcode(3x) accordingly.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain these functions. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

SEE ALSO

asclose server(3x), aserror code(3x), asopen server(3x)

cc(1), ld(1)

array_services(7), array_sessions(7)

arrayd(8)

NAME

asstrerror – Gets an Array Services error message string

SYNOPSIS

```
#include <arraysvcs.h>
const char *asstrerror(aserror_t Errorcode);
```

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The asstrerror function returns a pointer to a character string that describes the Array Services error code in errorcode. The string is contained in a static buffer and should be copied elsewhere before a subsequent call to either asstrerror or asperror.

The formal parameter is as follows:

Errorcode Specifies the error code to be described.

NOTES

The IRIX libarray.so and the UNICOS libarray.a libraries contain this function. You can load the libarray.so or libarray.a library by using the -larray option with cc(1) or ld(1).

RETURN VALUES

The asstrerror function always returns a valid character string, even if errorcode is an invalid error code.

SEE ALSO

aserrorcode(3x), asperror(3x)

cc(1), ld(1)

array_services(7), array_sessions(7)

NAME

atexit, atabort – Calls specified function on normal/abnormal termination

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (atexit only)
CRI extension (atabort only)

DESCRIPTION

The atexit function registers the function pointed to by *func*, to be called without arguments at normal program termination (for example, when the exit function is called). The atabort function registers the function pointed to by *func*, to be called without arguments at abnormal program termination (for example, when the abort function is called).

The standard requires that at least 32 functions can be registered by atexit. These functions are called in the reverse order of registration; no called function can call exit.

RETURN VALUES

Both atexit and atabort return 0 if the registration succeeds, a nonzero value if it fails.

SEE ALSO

exit(3C) abort(3C)

NAME

BARASGN – Identifies an integer variable to use as a barrier

SYNOPSIS

CALL BARASGN(*name*, *value*)

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

Before an integer variable can be used as an argument to any of the other barrier routines, it must first be identified as a barrier variable by BARASGN.

The following is a list of valid arguments for this routine.

Argument	Description
name	Integer variable to be used as a barrier. The library stores an identifier into this variable. Do not modify the variable after the call to BARASGN, unless a call to BARREL(3F) first releases the variable.
value	The integer number of tasks, between 1 and 31 inclusive, that must call BARSYNC(3F) with *name* before the barrier is opened and the waiting tasks are allowed to proceed.

The initial state of the barrier is closed. A barrier remains closed until its count is met (that is, until the BARSYNC(3F) routine has been called with this variable by the appropriate number of tasks). At this point, all waiting tasks are allowed to execute, and the barrier is once again closed.

SEE ALSO

BARREL(3F), BARSYNC(3F)

NAME

BARREL – Releases the identifier assigned to a barrier

SYNOPSIS

CALL BARREL(*name*)

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

BARREL releases the identifier assigned to a barrier. If a task is waiting for passage through the barrier, an error results. This subroutine is useful primarily in detecting erroneous uses of a barrier outside the region the program has planned for it. The barrier variable can be reused following another call to BARASGN(3F).

Argument	Description
name	Integer variable used as a barrier.

SEE ALSO

BARASGN(3F)

NAME

BARSYNC – Registers the arrival of a task at a barrier and suspends task execution until all other tasks arrive at the barrier

SYNOPSIS

CALL BARSYNC(*name*)

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

BARSYNC registers the arrival of a task at a barrier. This causes the barrier's count to be decremented by 1. If the current count is greater than 0, the task waits. If the current count is 0, the task is permitted to proceed through the barrier, all tasks waiting at the barrier are permitted to resume execution, and the barrier is closed, with the current count reset to the initial value set with the BARASGN(3F) call.

Argument	**Description**
name	Integer variable used at a barrier.

SEE ALSO

BARASGN(3F)

NAME

j0, j1, jn, y0, y1, yn – Returns Bessel functions

SYNOPSIS

```
#include <math.h>
double j0 (double x);
double j1 (double x);
double jn (int n, double x);
double y0 (double x);
double y1 (double x);
double yn (int n, double x);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The j0, j1, and jn functions return Bessel functions of *x* of the first kind of orders 0, 1, and *n* respectively.

The y0, y1, and yn functions return Bessel functions of *x* of the second kind of orders 0, 1, and *n* respectively. The value of *x* must be positive.

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

RETURN VALUES

Upon successful completion, these functions return the relevant Bessel value of *x* of the first or second kind. Nonpositive arguments cause y0, y1, and yn to return the value -HUGE_VAL and to set errno to EDOM.

Arguments too large in magnitude cause j0, j1, y0, and y1 to return 0 and to set errno to ERANGE.

On Cray MPP systems and CRAY T90 systems with IEEE arithmetic, j0(*NaN*), j1(*NaN*), jn(*NaN*), y0(*NaN*), y1(*NaN*), and yn(*NaN*) return NaN and errno is set to EDOM.

SEE ALSO

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

NAME

bindresvport – Binds a socket to a privileged IP port

SYNOPSIS

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

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The bindresvport library routine binds a socket descriptor to a privileged IP port, that is, a port number in the range 512 through 1023. This routine returns 0 if it is successful; otherwise, it returns –1, and errno is set to reflect the cause of the error. This routine differs from the rresvport routine (see rcmd(3C)) in that bindresvport works for any IP socket, and rresvport works for TCP only.

errno can take the following values:

EADDRINUSE	All reserved ports between 512 and 1023 are already in use, or the address *sin* is already in use.
EPFNOSUPPORT	The socket address *sin* address family is not AF_INET.
EACCES	Socket address *sin* is protected, and the current user has inadequate permission to access it.
EADDRNOTAVAIL	Socket address *sin* is unavailable from the local machine.
EBADF	Descriptor *sd* is invalid.
EFAULT	The address specified by *sin* is not a valid part of the user address space.
EINVAL	Descriptor *sd* is already bound to an address.
ENOTSOCK	Descriptor *sd* is not a socket.
ENOMEM	Unable to malloc enough memory for an internal table.

Only root or a process with PRIV.SOCKET on a least-privilege system can bind to a privileged port; this call fails for any other users.

The privileged ports present in the `/etc/services` file are not used by `bindresvport`. Programs using this routine do not conflict with servers that have privileged ports assigned in `/etc/services`.

SEE ALSO

`rcmd`(3C)

NAME

bsearch – Performs a binary search of an ordered array

SYNOPSIS

```
#include <stdlib.h>

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The bsearch function searches an ordered array of *nmemb* objects, the initial element of which is pointed to by *base*, for an element that matches the object pointed to by *key*. The size of each element of the array is specified by *size*. The elements of the array must be ordered so that the following is true:

key1 ≤ key2 ≤ . . . ≤ key*n*

The comparison function pointed to by *compar* is called with two arguments that point to the *key* object and to an array object, in that order. The function returns an integer less than, equal to, or greater than 0 if the *key* object is considered, respectively, to be less than, to match, or to be greater than the array object. The array consists of all the elements that compare less than, all the elements that compare equal to, and all the elements that compare greater than the *key* object, in that order.

NOTES

The pointers to the key and the element at the base of the table may be pointers to any type.

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

The value required should be cast into type pointer-to-element.

RETURN VALUES

The bsearch function returns a pointer to a matching element of the array, or a null pointer if no match is found. If two elements compare as equal, which element is matched is unspecified.

SEE ALSO

lsearch(3C) qsort(3C),

NAME

bcmp, bcopy, bzero, ffs – Operates on bits and byte strings

SYNOPSIS

```
#include <string.h>
int bcmp (const void *b1, const void *b2, size_t length);
void bcopy (const void *b1, void *b2, size_t length);
void bzero (void *b, size_t length);
int ffs (int i);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The bcmp, bcopy, and bzero functions operate on variable-length byte arrays. They do not check for null bytes as the functions described in string(3C) do.

The bcmp function compares byte array *b1* against byte array *b2*, returning 0 if they are identical; otherwise, it returns a nonzero value. Both byte arrays are assumed to be *length* bytes long.

The bcopy function copies *length* bytes from byte array *b1* to byte array *b2*.

The bzero function places *length* bytes of 0's in byte array *b*.

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

NOTES

The bcopy function takes parameters backwards in relation to the memcpy function described in memory(3C).

SEE ALSO

memory(3C), string(3C)

NAME

BUFDUMP – Writes an unformatted dump of the multitasking history trace buffer

SYNOPSIS

```
CALL BUFDUMP(empty, file)
```

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

BUFDUMP writes an unformatted dump of the contents of the multitasking history trace buffer to a specified file. The mtdump(1) command can later use this file to provide formatted reports of its contents or to let you examine the file. Actions are reported in chronological order. A special entry is added if the buffer has overflowed and entries are lost.

The following is a list of valid arguments for this routine:

Argument	Description
empty	On entry, an integer flag that is 0 if the buffer pointers are to be left unchanged; the flag is nonzero if the buffer is to be emptied after its contents are dumped.
file	Integer variable, expression, or constant containing the name of the file to which an unformatted dump of the multitasking history trace buffer is to be written. The name is case-sensitive, and it must be in ASCII, left-justified, and terminated by a zero byte. If you specify *file* as 0, the file passed to BUFTUNE(3F) is used; if no file was specified through BUFTUNE(3F), the request is ignored.

CAUTIONS

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

SEE ALSO

BUFTUNE(3F)

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

NAME

BUFPRINT – Writes formatted dump of multitasking history trace buffer to a specified file

SYNOPSIS

CALL BUFPRINT(*empty* [,*file*])

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

BUFPRINT writes a formatted dump of the contents of the multitasking history trace buffer to a specified file. Actions are reported in chronological order.

The following is a list of valid arguments for this routine:

Argument	Description
empty	On entry, an integer flag that is 0 if the buffer pointers are to be left unchanged or nonzero if the buffer is to be emptied after its contents are printed.
file	Integer variable, expression, or constant containing the name of the file to which a formatted dump is to be written. The name is case-sensitive, and it must be in ASCII, left-justified, and terminated by a zero byte. If no name is specified, stdout is used.

CAUTIONS

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

EXAMPLES

Example 1: The following example of BUFPRINT leaves the buffer unchanged after its output to stdout:

```
IEMPTY = 0
CALL BUFPRINT(IEMPTY)
```

Example 2: The following example of BUFPRINT zeroes out the buffer after its contents are written to stdout:

```
IEMPTY = 1
CALL BUFPRINT(IEMPTY)
```

SEE ALSO

BUFDUMP(3F)

NAME

BUFTUNE – Tunes parameters controlling multitasking history trace buffer

SYNOPSIS

CALL BUFTUNE(*keyword, value* [,*string*])

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

BUFTUNE tunes paramaters that control the multitasking history trace buffer. The following is a list of valid arguments for this routine:

Argument	Description
keyword	An integer variable containing an ASCII string, left-justified, blank-filled.
value	Either an integer or an ASCII string (left-justified, blank-filled), depending on the keyword.
string	A 24-character string (left-justified, blank-filled) used only with the keyword INFO.

You must specify a keyword, which must be in uppercase. Valid keywords, and their associated functions and meanings, are as follows:

Keyword	Description
DN	The value of the DN keyword is the file that you specify to receive a dump of the multitasking history trace buffer. DN itself directs this dump of the buffer to the file. If BUFTUNE is called without the DN keyword, the multitasking history trace buffer is not dumped to any file. The file name should be zero-filled (for example, 'ABC'L). Case is also important; 'ABC'L and 'abc'L are two distinct files.
FLUSH	Minimum integer number of unused entries in the multitasking history trace buffer. When the number of unused entries falls below this level, the buffer is flushed automatically; that is, it is written to the file specified by the DN option. If DN is specified, the default FLUSH value is 40.
ACTIONS	The value of ACTIONS is a 128-element integer array with a flag for each action that can be recorded in the multitasking history trace buffer. If the array element corresponding to a particular action is nonzero, that action is recorded; if the array element is 0, the action is ignored. The array indexes (action codes) corresponding to each action follow.

Code	Action
0	Starts task.
1	Completes task.
2	TSKWAIT, no wait.
3	Begins wait for task.
4	Runs after wait for task.

5	Tests task.
6	Assigns lock.
7	Releases lock.
8	Sets lock.
9	Begins wait to set lock.
10	Runs after wait for lock.
11	Clears lock.
12	Tests lock.
13	Assigns event.
14	Releases event.
15	Posts event.
16	Clears event.
17	EVWAIT, no wait.
18	Begins wait for event.
19	Runs after wait for event.
20	Tests event.
21	Attaches to logical CPU.
22	Detaches from logical CPU.
23,24	Requests a logical CPU. (These actions require two action codes, the second containing internal information.)
25	Acquires a logical CPU.
26,27	Deletes a logical CPU. (These actions require two action codes, the second containing internal information. (Cray PVP systems))
28,29	Suspends a logical CPU. (These actions require two action codes, the second containing internal information. (Cray PVP systems))
30,31	Activates a logical CPU. (These actions require two codes, the second containing internal information. (Cray PVP systems))
32	Begins spin-wait for a logical CPU.
33	Assigns barrier.
34	Releases barrier.
35	Calls BARSYNC(3F), no wait.
36	Begins wait at barrier.
37	Runs after wait for barrier.
38-63	Reserved for future use.
64-127	Reserved for user access (see BUFUSER(3F)).

INFO The value for this keyword is the integer user action code (64 through 127). The *string* argument is a 24-character information string, unique to each action, which you enter; it is printed for each user action code that is dumped.

BUFUSER(3F) lets you add entries to the multitasking history trace buffer. When the multitasking history trace buffer is dumped using BUFPRINT(3F) or mtdump(1) on Cray PVP systems, this 24-character information string is dumped along with each action. This information must be available early in the program so that the strings can be written to the dump file for processing by mtdump(1).

The INFO keyword does not turn these actions on to be recorded. They are normally on by default, but if you have previously turned them off, you may reactivate them by using the ACTIONS or USERS keyword in a BUFTUNE call.

TASKS If *value*='ON'H, actions numbered 1 through 6 are recorded; if *value*='OFF'H, those actions are ignored.

LOCKS If *value*='ON'H, actions numbered 7 through 13 are recorded; if *value*='OFF'H, those actions are ignored.

EVENTS If *value*='ON'H, actions numbered 14 through 21 are recorded; if *value*='OFF'H, those actions are ignored.

CPUS If *value*='ON'H, actions numbered 22 through 33 are recorded; if *value*='OFF'H, those actions are ignored.

BARRIERS If *value*='ON'H, actions 34 through 38 are recorded; if *value*='OFF'H, those actions are ignored.

USERS If *value*='ON'H, actions numbered 65 through 128 are recorded; if *value*='OFF'H, those actions are ignored.

FIOLK On Cray PVP systems, if *value*='ON'H, actions affecting the Fortran I/O lock are recorded; if *value*='OFF'H they are ignored. Library routines that handle Fortran reads and writes use this lock.

BUFTUNE can be called any number of times. If it is not called, or before it is called for the first time, default parameter values are used.

Before BUFTUNE is called, all actions involving tasks, locks, events, logical CPUs, barriers, and users are recorded, except for actions involving the Fortran I/O lock, which are ignored. A call to BUFTUNE with the TASKS, LOCKS, EVENTS, CPUS, BARRIERS, or USERS keyword affects only the actions associated with that keyword. The ACTIONS keyword overrides what has been requested through TASKS, LOCKS, EVENTS, CPUS, BARRIERS, or USERS.

CAUTIONS

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

EXAMPLES

The following BUFTUNE examples show two different ways to dump only task actions to file mtdumpfile:

```
*     Turn on task actions, turn everything else off
      INTEGER ACTION(128)
      DATA ACTION/6*1,122*0/
      CALL BUFTUNE('DN'L, 'mtdumpfile'L)
      CALL BUFTUNE('ACTIONS'L,ACTION)
```

or

```
*     Turn on task actions, turn everything else off
      CALL BUFTUNE('DN'L, 'mtdumpfile'L)
      CALL BUFTUNE('TASKS'L, 'ON'L)
      CALL BUFTUNE('LOCKS'L, 'OFF'L)
      CALL BUFTUNE('EVENTS'L, 'OFF'L)
      CALL BUFTUNE('CPUS'L, 'OFF'L)
      CALL BUFTUNE('BARRIERS'L, 'OFF'L)
      CALL BUFTUNE('USERS'L, 'OFF'L)
```

NAME

BUFUSER – Adds entries to the multitasking history trace buffer

SYNOPSIS

CALL BUFUSER(*action*, *data*)

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

BUFUSER lets you add entries to the multitasking history trace buffer. The following is a list of valid arguments for this routine.

Argument **Description**

action On entry, code for the type of action (see action codes in mtdump(1)). This value is compared against the bit of the same number in the mask in global variable G@BUFMSK, set up by BUFTUNE(3F). If the mask bit is set, an entry is added to the buffer. This value becomes the third word of the buffer entry.

A numerical code determines the action to be recorded in the buffer. Action codes 65 through 128 are reserved for this. The codes and their associated actions follow:

Code	Action
0 – 63	You cannot add entries with these action codes; if you attempt to do so, a warning is printed to stdout.
64 – 127	This action code is compared to the action codes specified in BUFTUNE(3F), either explicitly by the user or by default. If the action code appears in the BUFTUNE call, or if it is on by default, the corresponding entry is added to the multitasking history trace buffer. If the action code does not appear in the BUFTUNE call, this action/entry is ignored.

If a string is provided (see BUFTUNE), it is dumped into the action field of the output for this entry. If no string is provided, the (decimal) action code is dumped into the action field. In either case, *data* is written in octal (and ASCII, if it is a legal character) to the action-dependent data field of the output.

data Values added to the multitasking history trace buffer in addition to the internal task identifier and the current time. These actions-dependent data codes can be user-defined task values, a logical CPU number, a lock or event address, or the task identifier of the waited-upon task. The only restriction on these values is that they should be a single word. If an entry is added to the buffer, this value becomes the fourth word of the entry.

These entries are added unconditionally.

CAUTIONS

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

SEE ALSO

BUFTUNE(3F)

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

NAME

`htonl`, `htons`, `ntohl`, `ntohs` – Converts values between host and network byte order

SYNOPSIS

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

unsigned long htonl (unsigned long hostlong);

unsigned short htons (unsigned short hostshort);

unsigned long ntohl (unsigned short netlong);

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

These macros resolve differences between hosts that read the bytes in a word in an order other than network byte order (bytes ordered from left to right). Deviations from network byte order are referred to as *host byte order* because the ordering is host-dependent. The Cray Research system reads words in network byte order, so it does not need to resolve byte-order differences; however, in order to maximize the transportability of source code, these macros are still defined (as no-ops).

The macros that convert values between network byte order and host byte order are defined as null macros in the include file `/usr/include/netinet/in.h`. These macros are most often used in conjunction with Internet addresses and ports, as returned by `gethostent` (see `gethost`(3C)) and `getservent` (see `getserv`(3C)).

NOTES

There is no function definition for these names on Cray systems. `htonl`, `htons`, `tohl`, and `ntohs` are macros. If the macro definition is suppressed in order to access an actual function, the behavior is undefined.

SEE ALSO

`gethost`(3C), `getserv`(3C)

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

NAME

catgetmsg – Reads a message from a message catalog

SYNOPSIS

```
#include <nl_types.h>
char *catgetmsg (nl_catd catd, int set_num, int msg_num, char (**buf, int buflen);
```

IMPLEMENTATION

UNICOS systems

IRIX systems

STANDARDS

CRI extension

DESCRIPTION

The catgetmsg function returns the requested message string. The message string is placed in the user-supplied buffer (pointed to by *buf*) and terminated with a null byte. If the message is longer than *buflen* bytes, it is truncated with a null byte.

The *catd* argument is a catalog descriptor returned from an earlier call to catopen(3C); it identifies the message catalog that contains the message identified by the message set (*set_num*) and the message number (*msg_num*).

The *set_num* and *msg_num* arguments are defined as integer values for maximum portability. However, it is recommended that programmers use symbolic names for message and set numbers wherever possible, rather than having integer values hard-coded into their source programs. The NL_MSGSET macro in the nl_types.h file must be passed as the *set_num* argument.

NOTES

You can use the catgetmsg and catgets(3C) functions to retrieve messages from a message catalog. On Cray Research systems, catgetmsg is optimized for programs that retrieve only a few messages. catgets(3C) is optimized for programs that retrieve many messages.

Specifically, catgetmsg minimizes memory usage at the expense of more frequent disk accesses. The catgets(3C) function minimizes disk accesses at the expense of more memory usage. If it is important to your application to minimize usage of one of these resources, use the corresponding function.

RETURN VALUES

If successful, `catgetmsg` returns a pointer to the message string in *buf*.

If `catgetmsg` is unsuccessful because the message catalog identified by *catd* is not currently available, or the requested message is not in the message catalog, a pointer to a null ("") string is returned.

SEE ALSO

`catgets`(3C), `catmsgfmt`(3C), `catopen`(3C) describe message system library functions

`caterr`(1), `catxt`(1), `explain`(1), `gencat`(1), `whichcat`(1) describe message system user commands in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`nl_types`(5) describes the file that defines message system variables for use in programs in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

Cray Message System Programmer's Guide, Cray Research publication SG–2121, contains details about all aspects of the message system

NAME

catgets – Gets message from a message catalog

SYNOPSIS

```
#include <nl_types.h>
char *catgets (nl_catd catd, int set_num, int msg_num, const char *s);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The catgets function returns a pointer to the requested message string. The string is terminated by a null byte. The message text is contained in an internal buffer and should not be altered or freed (by using free(3C)). It should be used or copied before any subsequent calls to catgets, catgetmsg(3C), or catclose(3C).

The *catd* argument is a catalog descriptor returned from an earlier call to catopen(3C); it identifies the message catalog containing the message identified by the message set (*set_num*) and the message number (*msg_num*).

The *set_num* and *msg_num* arguments are defined as integer values for maximum portability. However, it is recommended that programmers use symbolic names for message and set numbers wherever possible, rather than having integer values hard-coded into their source programs. The NL_MSGSET macro in the nl_types.h file must be passed as the *set_num* argument.

The *s* argument points to a default message string that will be returned by catgets if the identified message catalog is not currently available or if any other error is encountered during message retrieval.

NOTES

You can use the catgets and catgetmsg(3C) functions to retrieve messages from a message catalog. On Cray Research systems, catgetmsg(3C) is optimized for programs that retrieve only a few messages. catgets is optimized for programs that retrieve many messages.

Specifically, catgetmsg(3C) minimizes memory usage at the expense of more frequent disk accesses. The catgets function minimizes disk accesses at the expense of more memory usage. If it is important to your application to minimize usage of one of these resources, use the corresponding function.

RETURN VALUES

If successful, `catgets` returns a pointer to the null-terminated message string in an internal buffer.

If `catgets` is unsuccessful, a pointer to *s* is returned.

SEE ALSO

`catclose`(3C), `catgetmsg`(3C), `catmsgfmt`(3C), `catopen`(3C) describe message system library functions

`caterr`(1), `catxt`(1), `explain`(1), `gencat`(1), `whichcat`(1) describe message system user commands in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`nl_types`(5) describes the file that defines message system variables for use in programs in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

Cray Message System Programmer's Guide, Cray Research publication SG–2121, contains details about all aspects of the message system

NAME

catmsgfmt – Formats an error message

SYNOPSIS

```
#include <nl_types.h>

char *catmsgfmt (const char *cmdname, const char *groupcode, int msgnum, const
char *severity, const char *msgtext, char *buf, int buflen [, const char *position [,
const char *debug]]);
```

IMPLEMENTATION

UNICOS systems

IRIX systems

STANDARDS

CRI extension

DESCRIPTION

The catmsgfmt function produces a formatted message that consists of the command name (*cmdname*), group code (*groupcode*), message number (*msgnum*), severity level (*severity*), message text (*msgtext*), and optional position (*position*) and debugging (*debug*) information. The formatted message is placed in the user-supplied buffer, which is pointed to by *buf*, and terminated with a null byte. If the formatted message is longer than *buflen* bytes, it is truncated to *buflen* bytes with a null byte.

The *cmdname*, *groupcode*, *severity*, *msgtext*, and optional *position* and *debug* arguments are null-terminated strings. The command name identifies the command or function issuing the error message. Typically, the group code is the same value as that specified as the *name* parameter on the catopen(3C) function. Typically, the message number is the same value as that specified on the catgetmsg(3C) or catgets(3C) function.

The *position* and *debug* arguments are optional. Their contents are inserted in the error message only if provided and only if included in the MSG_FORMAT environment variable. Specifying a null value for either (or both) parameters is equivalent to not specifying either (or both) parameters.

NOTES

The MSG_FORMAT environment variable controls the formatting of the message. If the MSG_FORMAT environment variable is not defined, a default format is used. See the explain(1) man page for a description of message formats and the MSG_FORMAT environment variable.

MSG_FORMAT can include an optional time stamp for the message. The format of this time stamp is equivalent to that produced by the cftime(3C) function and can be overridden by the CFTIME environment variable. For a description of time-stamp formats, see the strftime(3C) man page.

RETURN VALUES

If successful, catmsgfmt returns a pointer to the user-supplied buffer. If unsuccessful, it returns a null pointer.

SEE ALSO

catgetmsg(3C), catgets(3C), catopen(3C) describe message system library functions
strftime(3C) describes time-stamp formatting

caterr(1), catxt(1), explain(1), gencat(1), whichcat(1) describe message system user commands in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

nl_types(5) describes the file that defines message system variables for use in programs in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

Cray Message System Programmer's Guide, Cray Research publication SG–2121, contains details about all aspects of the message system

NAME

catopen, catclose – Opens or closes a message catalog

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The catopen function opens a message catalog and returns a catalog descriptor. *name* specifies the group name of the message catalog to be opened; it is a pointer to a null-terminated string. If *name* contains a slash (/), it specifies a path name for the message catalog; otherwise, the NLSPATH environment variable is used, with *name* substituted for %N. NLSPATH is described in the following paragraphs. The message catalog is opened with the FD_CLOEXEC flag set.

If the value of the *oflag* argument is 0, the LANG environment variable is used to locate the catalog without regard to the LC_MESSAGES category. If the *oflag* argument is NL_CAT_LOCALE, the LC_MESSAGES category is used to locate the message catalog. (The LC_MESSAGES category is part of the locale environment. See the locale(1) and setlocale(3C) man pages for information about reading and setting the locale environment.)

The catclose function closes the message catalog identified by *catd* and releases all memory allocated for use by that catalog file.

The catopen function uses the NLSPATH environment variable and either the LANG environment variable or the LC_MESSAGES category to locate the correct message catalog. The LANG environment variable or LC_MESSAGES category identifies the user's requirements for native language, local customs, and coded character set. These components are specified by a string of the following form:

language[*_territory*[.*codeset*]]

The string En is the designation for the English language. Other language designations (if any) are defined and supported locally.

The value of the LANG environment variable or the LC_MESSAGES category is part of the message system default value of NLSPATH, the message system search path environment variable. Message system functions substitute fields denoted by % characters in the definition of NLSPATH to determine the catalog search path.

The following are the valid fields defined for NLSPATH:

- %N The value of the *name* argument passed to catopen.
- %L The value of the LANG environment variable or the LC_MESSAGES category.
- %l The language component of the LANG environment variable or the LC_MESSAGES category. This element determines the language in which the message is displayed.
- %t The territory component of the LANG environment variable or the LC_MESSAGES category.
- %c The codeset component of the LANG environment variable or the LC_MESSAGES category.

Path name templates defined in NLSPATH are separated by colons (:). A leading colon or two adjacent colons (::) is equivalent to specifying %N.

If NLSPATH is not defined by the user, it is assumed to be defined as follows:

```
/usr/lib/nls/%l/%N.cat:/lib/nls/%l/%N.cat:/usr/lib/nls/En/%N.cat:
/lib/nls/En/%N.cat
```

NOTES

Using catopen could cause another file descriptor to be allocated by the calling process. Applications should take care not to close this file descriptor by mistake.

RETURN VALUES

If successful, catopen returns a message catalog descriptor for use on subsequent calls to catgetmsg(3C), catgets(3C), and catclose. If unsuccessful, catopen returns –1.

If successful, catclose returns 0. If unsuccessful, catclose returns –1.

SEE ALSO

`catgetmsg`(3C), `catgets`(3C), `catmsgfmt`(3C) describe message system library functions

`setlocale`(3C) describes setting the locale environment from a program

`caterr`(1), `catxt`(1), `explain`(1), `gencat`(1), `whichcat`(1) describe message system user commands in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`locale`(1) describes reading the locale environment in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`nl_types`(5) describes the file that defines message system variables for use in programs in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

Cray Message System Programmer's Guide, Cray Research publication SG–2121, contains details about all aspects of the message system

NAME

cfgetospeed, cfsetospeed, cfgetispeed, cfsetispeed – Gets or sets terminal input or output baud rates

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX

DESCRIPTION

The following interfaces get and set the values of the input and output baud rates in the termios structure. The effects on the terminal device do not become effective until function tcsetattr is successfully called.

The input and output baud rates are stored in the termios structure. The values shown in the following list are supported. The name symbols in this list are defined in header <termios.h>.

Name	Description
B0	Hang up
B50	50 baud
B75	75 baud
B110	110 baud
B134	134.5 baud
B150	150 baud
B200	200 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud
B2400	2400 baud
B4800	4800 baud
B9600	9600 baud

Name	Description
B19200	19,200 baud
B38400	38,400 baud

The type speed_t is defined in header <termios.h> and is an unsigned integral type.

The termios_p argument is a pointer to a termios structure.

Function cfgetospeed returns the output baud rate stored in the termios structure pointed to by termios_p.

Function cfsetospeed sets the output baud rate stored in the termios structure pointed to by termios_p to *speed*. The zero baud rate, B0, is used to terminate the connection. If B0 is specified, the modem control lines are no longer asserted. Normally, this disconnects the line.

Function cfgetispeed returns the input baud rate stored in the termios structure.

Function cfsetispeed sets the input baud rate stored in the termios structure to *speed*. If the input baud rate is set to 0, the input baud rate will be specified by the value of the output baud rate.

RETURN VALUES

Functions cfsetispeed and cfsetospeed both return a value of 0 if successful; otherwise, they return –1 to indicate an error.

Attempts to set unsupported baud rates are ignored, and no errors are returned by cfsetispeed, cfsetospeed, or tcsetattr in these cases. Such attempts include both changes to baud rates not supported by the hardware, and changes setting the input and output baud rates to different values, if the hardware does not support this.

SEE ALSO

terminal(3C), tcgetattr(3C)

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

NAME

character – Introduction to character-handling functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The character-handling functions provide various means for testing characters for specific attributes or for translating one character to another.

Unless otherwise noted, all of these functions have an argument of type `int`, the value of which must be representable as an `unsigned char` (that is, less than or equal to `UCHAR_MAX`, defined in `limits.h` to be 255 for Cray Research systems) or equal to `EOF`. If the argument has any other value, the behavior of the function is undefined.

All of these functions are implemented as both inline macros and library functions. (If `#undef` is used to remove the macro definition and obtain access to the underlying function, however, the behavior is undefined.) These functions are implemented as macros for speed and as functions so that the address of the function can be taken. Under normal circumstances (`#undef` not used), the results returned are the same for either the macro version or the function version.

In all locales, the value of each character after the 0 digit character in the set of decimal digits is one greater than the value of the previous character; and the value of each character after "a" in the set of a through f is one greater than the value of the previous character; and the value of each character after A in the set of A through F is one greater than the value of the previous character. This is so algorithms that convert octal, decimal, and hexadecimal characters to numeric values can work efficiently. For all other characters, the collating sequence and the attributes can be changed when changing to a different locale.

The behavior of most `ctype` functions is dependent upon the current locale. The behaviors described here are for the standard ASCII character set and the C locale. Other locales are possible; if any other locale is selected by using the `setlocale` function, refer to documentation for that locale for description of any changes.

ASSOCIATED HEADERS

<`ctype.h`>	File defining character classification and conversion functions and macros
<`wchar.h`>	File defining wide character functions

ASSOCIATED FUNCTIONS

Testing Functions

Function	Description
`isalnum`	Tests for alphanumeric characters (see `ctype`(3C)).
`isalpha`	Tests for alpha characters (see `ctype`(3C)).
`isascii`	Tests for ASCII character (see `ctype`(3C)).
`iscntrl`	Tests for control characters (see `ctype`(3C)).
`isdigit`	Tests for decimal-digit character (see `ctype`(3C)).
`isenglish`	Tests whether *wc* is a wide character representing a character of class `english` in the program's current locale (see `wctype`(3C)).
`isgraph`	Tests for any printing character but space (see `ctype`(3C)).
`isideogram`	Tests whether *wc* is a wide character representing a character of class `ideogram` in the program's current locale (see `wctype`(3C)).
`islower`	Tests for lowercase alpha character (see `ctype`(3C)).
`isnumber`	Tests whether *wc* is a wide character representing a character of class `number` in the program's current locale (see `wctype`(3C)).
`isphonogram`	Tests whether *wc* is a wide character representing a character of class `phonogram` in the program's current locale (see `wctype`(3C)).
`isprint`	Tests for printing character (see `ctype`(3C)).
`ispunct`	Tests for punctuation character (see `ctype`(3C)).
`isspace`	Tests for white-space character (see `ctype`(3C)).
`isspecial`	Tests whether *wc* is a wide character representing a character of class `special` in the program's current locale (see `wctype`(3C)).
`isupper`	Tests for uppercase alpha character (see `ctype`(3C)).
`iswalnum`	Tests whether *wc* is a wide character representing a character of class `alpha` or `digit` in the program's current locale (see `wctype`(3C)).
`iswalpha`	Tests whether *wc* is a wide character representing a character of class `alpha` in the program's current locale (see `wctype`(3C)).
`iswcntrl`	Tests whether *wc* is a wide character representing a character of class `cntrl` in the program's current locale (see `wctype`(3C)).
`iswctype`	Determines whether the wide character *wc* has the character class *charclass*, returning true or false (see `wctype`(3C)).
`iswdigit`	Tests whether *wc* is a wide character representing a character of class `digit` in the program's current locale (see `wctype`(3C)).
`iswgraph`	Tests whether *wc* is a wide character representing a character of class `graph` in the program's current locale (see `wctype`(3C)).
`iswlower`	Tests whether *wc* is a wide character representing a character of class `lower` in the program's current locale (see `wctype`(3C)).
`iswprint`	Tests whether *wc* is a wide character representing a character of class `print` in the program's current locale (see `wctype`(3C)).
`iswpunct`	Tests whether *wc* is a wide character representing a character of class `punct` in the program's current locale (see `wctype`(3C)).

`iswspace`	Tests whether *wc* is a wide character representing a character of class `space` in the program's current locale (see `wctype`(3C)).
`iswupper`	Tests whether *wc* is a wide character representing a character of class `upper` in the program's current locale (see `wctype`(3C)).
`iswxdigit`	Tests whether *wc* is a wide character representing a character of class `xdigit` in the program's current locale (see `wctype`(3C)).
`isxdigit`	Tests for hexadecimal-digit character (see `ctype`(3C)).
`wctype`	Converts character class names to an argument suitable for `iswctype`(3C) (see `wctype`(3C)).

Translating Functions

Function	Description
`toascii`	Translates characters (see `conv`(3C)).
`tolower`	Translates characters to lowercase (see `conv`(3C)).
`_tolower`	Translates characters to lowercase (see `conv`(3C)).
`toupper`	Translates characters to uppercase (see `conv`(3C)).
`_toupper`	Translates characters to uppercase (see `conv`(3C)).
`towupper`	Translates wide characters to upper case (see `wconv`(3C).
`towlower`	Translates wide characters to lower case (see `wconv`(3C).

Other Functions

Function	Description
`wcwidth`	Returns number of column positions of a wide–character code.

SEE ALSO

`conv`(3C), `ctype`(3C), `limits.h`(3C), `locale`(3C) (the introduction to locale information functions), `wconv`(3C), `wctype`(3C)

NAME

cimag, creal, conj – Manipulates parts of complex values

SYNOPSIS

```
#include <complex.h>
double cimag (double complex x);
double creal (double complex x);
double complex conj (double complex x);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The cimag function computes the imaginary part of the double complex number *x*.

The creal function computes the real part of the double complex number *x*.

The conj function computes the conjugate of the double complex number *x* by negating the imaginary part of *x*.

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

RETURN VALUES

The cimag function returns the imaginary part of *x*.

The creal function returns the real part of *x*.

The conj function returns the conjugate of *x*.

NAME

clock – Reports CPU time used

SYNOPSIS

```
#include <time.h>
clock_t clock (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The clock function returns the implementation's best approximation of the amount of processor time (in microseconds) used since the first call to clock. Under UNICOS, the time reported is the sum of the user and system times of the calling process and its terminated child processes for which it has executed a wait(2), pclose(3C), or system(3C) call.

To determine the time in seconds, the value returned by the clock function should be divided by the value of the macro CLOCKS_PER_SEC, defined in <time.h>. If the processor time used is not available, or its value cannot be represented, the function returns the value (clock_t)-1.

NOTES

The value returned by the clock function is not consistent. This is because it includes system time, which, in a multiprogramming environment, is not consistent. Even in a monoprogramming situation, disk I/O can cause inconsistency.

EXAMPLES

```
#include <stdio.h>
#include <time.h>
#define SIZE 4096
main()
{
   float a[SIZE], b[SIZE], c[SIZE];
   clock_t time1, time2;
   int i;

   for (i = 0; i < SIZE; i++)
      a[i]=b[SIZE-1-i]=i;

   time1=clock();
   for (i = 0; i < SIZE; i++)
      c[i]=a[i]+b[i];
   time2=clock();

   printf("This loop takes %d/%d seconds\n",time2-time1,CLOCKS_PER_SEC);
}
```

SEE ALSO

`pclose` (see `popen`(3C)), `system`(3C)

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

`SECOND`(3F) in the

NAME

common_def – Introduction to common definition headers

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The common definition headers provide type definitions (typedefs) and macros that are used often by many programs and expand to implementation-specific values.

When the typedef or macro is directly associated with a set of functions that have a common purpose, it is usually defined in the header associated with that set of functions. For example, the definition

```
typedef long int clock_t;
```

is found in header <time.h>.

There are, however, some definitions that are used with more than one set of functions; for that reason, the common definition headers are provided.

ASSOCIATED HEADERS

<stddef.h>
<sys/types.h> (described on man page sys_types.h.)
<unistd.h>

ASSOCIATED FUNCTIONS

None

NAME

`complex.h` – Library header for complex math functions

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

TYPES

None

MACROS

The macros defined in header `complex.h` are as follows:

Macro	Standards	Description
`complex`	CRI extension	Defines the complex type keyword.
`CMPLXF`	CRI extension	Composes a `float complex` value from two `float` arguments.
`CMPLX`	CRI extension	Composes a `double complex` value from two `double` arguments.
`CMPLXL`	CRI extension	Composes a `long double complex` value from two `long double` arguments.

FUNCTIONS

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

`csin`(3C)	`ccos`(3C)	`cexp`(3C)	`clog`(3C)	`cpow`(3C)
`csqrt`(3C)	`cabs`(3C)	`cimag`(3C)	`conj`(3C)	`creal`(3C)

NOTES

The `complex.h` header must be included in every source file where the `complex` data type is used. The `complex` macro must be used to specify `complex` type.

EXAMPLES

When executed, the following example prints `z1 = <1.50,0.20>`:

```
#include <stdio.h>
#include <complex.h>
main()
{
      double complex z1;

      z1 = CMPLX(1.5,.2);
      printf("z1 = <%.2f,%.2f>0, creal(z1), cimag(z1));
}
```

SEE ALSO

cabs(3C), ccos(3C), cexp(3C), cimag(3C), clog(3C), conj(3C), cpow(3C), creal(3C), csin(3C), csqrt(3C), math.h(3C)

NAME

confstr – Gets configurable string values

SYNOPSIS

```
#include <unistd.h>
size_t confstr(int name, char *buf, size_t len);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX

DESCRIPTION

The confstr function gets configuration-defined string values.

The system variable to be queried is the *name* argument. This argument can be _CS_PATH, which returns a value for the PATH environment variable that finds all standard utilities. _CS_PATH is defined in the header file unistd.h.

If *len* is greater than zero and *name* has a configuration-defined value, constr copies that value into the *len*-byte buffer pointed to by *buf*. If the string to be returned exceeds *len* bytes, including the final null, confstr truncates the string to *len*– 1 bytes and null-terminates the result. The application can detect that the string was truncated by comparing the returned value with *len*.

If *len* is zero, confstr returns the integer value defined below, but no string. (This is true whether or not *buf* is null.)

RETURN VALUES

If *name* has no configuration-defined value, confstr returns zero and leaves errno unchanged.

If *name* has a configuration-defined value, confstr returns the buffer size of the entire configuration-defined value. If this return value exceeds *lens*, the *buf* return string has been truncated.

NAME

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

SYNOPSIS

```
#include <ctype.h>
int toupper (int c);
int tolower (int c);
int _toupper (int c);
int  _tolower  (int c);
int toascii (int c);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (toupper and tolower only)
XPG4 (_toupper, _tolower, toascii only)

DESCRIPTION

The toupper and tolower functions have as domains a type int, the range of which is representable as an unsigned char (that is, ≤ UCHAR_MAX, defined in limits.h to be 255 for Cray Research systems) or equal to EOF. If the argument of toupper represents a lowercase letter, and there exists a corresponding uppercase letter in the program's locale, the result is the corresponding uppercase letter. If the argument of tolower represents an uppercase letter, and there exists a corresponding lowercase letter in the program's locale, the result is the corresponding lowercase letter. All other arguments in the domain are returned unchanged.

The _toupper and _tolower functions accomplish the same thing as toupper and tolower, but have a restricted domain and are faster. The _toupper function requires a lowercase letter as its argument; its result is the corresponding uppercase letter. The _tolower function requires an uppercase letter as its argument; its result is the corresponding lowercase letter. Arguments outside the domain cause undefined results.

The toascii function yields its argument with all bits turned off that are not part of a standard 7-bit ASCII character.

NOTES

The behavior of functions `toupper` and `tolower` may be affected by the current locale.

The behavior of functions `_toupper`, `_tolower`, and `toascii` are **not** affected by the current locale.

SEE ALSO

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

NAME

`copysign`, `copysignf`, `copysignl` – Assigns the sign of its second argument to the value of its first argument

SYNOPSIS

CRAY T90 systems with IEEE floating-point hardware:

```
#include <fp.h>

double copysign (double *x, double y);
float copysignf (float *x, float y);
long double copysignl (long double *x, long double y);
```

Cray MPP systems:

```
#include <fp.h>

double copysign (double *x, double y);
```

IMPLEMENTATION

Cray MPP systems (implemented as a macro)
CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

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

DESCRIPTION

The `copysign` function and macro and the `copysignf` and `copysignl` functions produce values with the magnitude of *x* and the sign of *y*. If *x* is a NaN, they produce a NaN with the sign of *y*.

RETURN VALUES

Returns the value of *x* with the sign of *y*.

SEE ALSO

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

NAME

_cptofcd, _fcdtocp, _fcdlen, _btol, _ltob – Passes character strings and logical values between Standard C and Fortran

SYNOPSIS

```
#include <fortran.h>
_fcd _cptofcd (char *ccp, unsigned len);
char *_fcdtocp (_fcd fcd);
unsigned _fcdlen (_fcd fcd);
long _ltob (long *log);
long _btol (long bool);
int _isfcd (void *ptr);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

All of these functions communicate between functions written in Standard C and functions written in Fortran that pass character strings and logical values as arguments.

Type _fcd is defined in the header file fortran.h and matches the format of a Fortran character descriptor. An object with type _fcd can be passed to a Fortran subprogram whose corresponding formal parameter has type CHARACTER.

Function _cptofcd creates a Fortran character descriptor from the C character pointer *ccp* and byte length *len*. The resulting descriptor points to the same string as *ccp* and is compatible with Fortran type CHARACTER.

Function _fcdtocp extracts a C character pointer from the Fortran character descriptor *fcd*.

Function _fcdlen extracts the byte length from the Fortran character descriptor *fcd*.

Function _btol converts a 0 to a Fortran logical .FALSE. Function _btol converts a nonzero long int to a Fortran logical .TRUE.

Function _ltob converts a Fortran logical .FALSE. to a 0. Function _ltob converts a Fortran logical .TRUE. to a 1.

Function `_isfcd` determines whether a generic pointer is a Fortran character descriptor. If the pointer is not a Fortran character descriptor, it returns 0; otherwise it returns nonzero.

NOTES

At present, type `_fcd` matches the format of a Fortran character descriptor. This format might change in the future, however, which would cause the underlying C type `_fcd` to change also.

The use of `_fcd` in a cast is not guaranteed to work; the underlying type might be a structure type.

RETURN VALUES

Function `_cptofcd` returns a Fortran character descriptor from the C character pointer *ccp* and byte length *len*.

Function `_fcdtocp` returns a C character pointer that points to the same character string as the Fortran character descriptor *fcd*.

Function `_fcdlen` returns the byte length of the character string to which the Fortran character descriptor *fcd* points.

Function `_btol` returns the C `long int` Boolean value of a Fortran `LOGICAL` argument.

Function `_ltob` returns the Fortran `LOGICAL` value of a C `long int` Boolean argument.

Function `_isfcd` returns the C `int` Boolean value if the generic pointer is a Fortran character descriptor.

EXAMPLES

The following example shows a C function calling a Fortran subprogram, and the associated Fortran subprogram:

```
/*                      C program (main.c):                        */

#include <stdio.h>
#include <string.h>
#include <fortran.h>

fortran double CFTFCTN (_fcd, int *);

double REAL1 = 1.6;
double REAL2;  /* Initialized in CFTFCTN */

main( )
{
     int clogical, cftlogical, cstringlen;
     double rtnval;
     char *cstring = "C character string";
     _fcd cftstring;

/* Convert cstring and clogical to their Fortran equivalents */
     cftstring = _cptofcd(cstring, strlen(cstring));
     clogical = 1;
     cftlogical = _btol(clogical);

/* Print values of variables before call to Fortran function */
     printf(" In main: REAL1 = %g; REAL2 = %g\n",
           REAL1, REAL2);
     printf(" Calling CFTFCTN with arguments:\n");
     printf(" string = \"%s\"; logical = %d\n\n", cstring, clogical);

     rtnval = CFTFCTN(cftstring, &cftlogical);

/* Convert cftstring and cftlogical to their C equivalents */
     cstring = _fcdtocp(cftstring);
     cstringlen = _fcdlen(cftstring);
     clogical = _ltob(&cftlogical);

/* Print values of variables after call to Fortran function */
     printf(" Back in main: CFTFCTN returned %g\n", rtnval);
     printf(" and changed the two arguments:\n");
     printf(" string = \"%.*s\"; logical = %d\n",
     cstringlen, cstring, clogical);
}
```

The following Fortran subprogram is associated with the preceding C function:

```
C     Fortran subprogram (cftfctn.f):

      FUNCTION CFTFCTN(STR, LOG)

      REAL CFTFCTN
      CHARACTER*(*) STR
      LOGICAL LOG

      COMMON /REAL1/REAL1
      COMMON /REAL2/REAL2
      REAL REAL1, REAL2
      DATA REAL2/2.4/         ! REAL1 initialized in MAIN

C  Print current state of variables
      PRINT*, '    IN CFTFCTN:  REAL1 = ', REAL1,
     1                         ';REAL2 = ', REAL2
      PRINT*, '    ARGUMENTS:    STR = "', STR, '"; LOG = ', LOG

C  Change the values for STR(ing) and LOG(ical)
      STR = 'New Fortran String'
      LOG = .FALSE.

      CFTFCTN = 123.4
      PRINT*, '    Returning from CFTFCTN with ', CFTFCTN
      PRINT*
      RETURN
      END
```

The following example shows a Fortran subprogram calling a C function and the associated C function:

```
C   Fortran program (main.f):

      PROGRAM MAIN

      REAL CFCTN
      COMMON /REAL1/REAL1
      COMMON /REAL2/REAL2
      REAL REAL1, REAL2
      DATA REAL1/1.6/      ! REAL2 initialized in cfctn

      LOGICAL LOG
      CHARACTER*24 STR
      REAL RTNVAL

C  Initialize variables STR(ing) and LOG(ical)
      STR = 'Fortran Character String'
      LOG = .TRUE.

C  Print values of variables before call to C function
      PRINT*, ' IN MAIN: REAL1 = ', REAL1,
     1                '; REAL2 = ', REAL2
      PRINT*, ' CALLING CFCTN WITH ARGUMENTS: '
      PRINT*, ' STR = "', STR, '"; LOG = ', LOG
      PRINT*

      RTNVAL = CFCTN(STR, LOG)

C  Print values of variables after call to C function
      PRINT*, ' Back in MAIN:  CFCTN returned ', RTNVAL
      PRINT*, ' and changed the two arguments: '
      PRINT*, ' STR = "', STR, '"; LOG = ', LOG
      END
```

The following is the associated C function:

```
/*                  C function (cfctn.c):                        */
#include <fortran.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

float REAL1;        /* Initialized in MAIN */
float REAL2 = 2.4;

float CFCTN(_fcd str, int *log)

{
       int slen;
       int clog;
       float returnval;
       char *cstring;
       char newstr[25];

/* Convert str and log passed from Fortran MAIN into C equivalents */
       slen = _fcdlen(str);
       cstring = malloc(slen+1);
       strncpy(cstring, _fcdtocp(str), slen);
       cstring[slen] = '\0';
       clog = _ltob(log);

/* Print the current state of the variables */
       printf("     In CFCTN:  REAL1 = %.1f; REAL2 = %.1f\n",
                  REAL1, REAL2);
       printf("     Arguments:  str = \"%s\"; log = %d\n",
       cstring, clog);

/* Change the values for str and log */
       strncpy(_fcdtocp(str), "C Character String     ", 24);
       *log = 0;

       returnval = 123.4;
       printf("     Returning from CFCTN with %.1f\n\n", returnval);
       return(returnval);
}
```

SEE ALSO

Cray Standard C Reference Manual, Cray Research publication SR–2074, for complete examples of interlanguage communication

NAME

cpused – Gets task CPU time in RTC ticks

SYNOPSIS

```
#include <time.h>
long cpused (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

On Cray PVP systems, the cpused function returns the user CPU time used by the calling task in real-time clock (RTC) ticks. On Cray MPP systems, the cpused function returns the user CPU time used by the calling process in real-time clock (RTC) ticks.

The accuracy of cpused is not affected by system interrupts.

This function is equivalent to the TSECND(3F) function (except it returns the time in RTC ticks rather than seconds); it returns the elapsed CPU time of the calling task or process.

See SECOND(3F) for information about gathering CPU time for all tasks or processes.

For the CRAY T90 and CRAY C90 series, CPU times returned by cpused include wait-semaphore time. For all other systems, CPU times returned by cpused do not include wait-semaphore time.

On Cray PVP systems, use of cpused while running Flowtrace can cause incorrect Flowtrace statistics to be generated.

EXAMPLES

In the following example, cpused collects data before and after a section of code. Subtracting the first value from the second yields the CPU time spent within the code.

```
#include <stdio.h>
#include <sys/types.h>
#include <time.h>

main()
{
     time_t before, after, utime;

     before = cpused();


     /* Section of code here is where user execution time is to be measured. */

     after = cpused();

     utime = after - before;

     printf("\nCPU time used in user space = %ld clock ticks\n", utime);
}
```

The output appears as follows:

```
CPU time used in user space = 211 clock ticks
```

FORTRAN EXTENSIONS

The `ICPUSED` entry point is the Fortran-callable equivalent of `cpused`.

SEE ALSO

`rtclock`(3C)

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

`SECOND`(3F), `TSECND`(3F) in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165

NAME

crypt, encrypt, setkey – Generates DES encryption

SYNOPSIS

```
#include <crypt.h>
char *crypt (const char *key, const char *salt);
void encrypt (char block[64], int edflag);
void setkey (const char *key);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

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

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

The `setkey` and `encrypt` entries provide rather primitive access to the actual DES algorithm. The argument to `setkey` is a character array of length 64, containing only the characters with numerical values 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored; this gives a 56-bit key that is set into the machine. This is the key that is used with the previously mentioned algorithm to encrypt or decrypt string *block* with the `encrypt` function.

The *block* argument to the `encrypt` entry is a character array of length 64, containing only the characters with numerical values 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the DES algorithm using the key set by `setkey`. If *edflag* is `0`, the argument is encrypted; if *edflag* is nonzero, the argument is decrypted, or, if the implementation does not support this functionality, `errno` is set to `ENOSYS`.

NOTES

Inclusion of the Data Encryption Standard (DES) encryption code requires a special license for sites outside the United States and Canada. If these encryption functions are not available on your system, check with your system administrator or site analyst.

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

SEE ALSO

`getpass`(3C), `libudb`(3C)

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

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

NAME

ctermid – Generates file name for terminal

SYNOPSIS

```
#include <stdio.h>
char *ctermid (char *s);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX

DESCRIPTION

The ctermid function generates the path name of the controlling terminal for the current process and stores it in a string.

If *s* is a null pointer, the string is stored in an internal static area, the contents of which are overwritten at the next call to ctermid, and the address of which is returned. Otherwise, *s* is assumed to point to a character array of at least L_ctermid elements; the path name is placed in this array, and the value of *s* is returned. The constant L_ctermid is defined in the header file stdio.h.

NOTES

The difference between ctermid and ttyname(3C) is that ttyname(3C) must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while ctermid returns a string (/dev/tty) that refers to the terminal, if the terminal name is used as a file name. Thus, ttyname(3C) is useful only if the process already has at least one file open to a terminal.

SEE ALSO

ttyname(3C)

NAME

ctime, ctime_r, localtime, localtime_r, gmtime, gmtime_r, asctime, asctime_r, timezone, daylight, tzname, tzset – Converts from and to various forms of time

SYNOPSIS

```
#include <time.h>
char *ctime (const time_t *timer);
char *ctime_r (const time_t *timer, char *buf);
struct tm *localtime (const time_t *timer);
struct tm *localtime_r (const time_t *timer, struct tm *result);
struct tm *gmtime (const time_t *timer);
struct tm *gmtime_r (const time_t *timer, struct tm *result);
char *asctime (const struct tm *timeptr);
char *asctime_r (const struct tm *timeptr, char *buf);
extern long timezone;
extern int daylight;
extern char *tzname[2];
void tzset (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (ctime, localtime, gmtime, and asctime only)
POSIX (tzname and tzset only)
XPG4 (timezone and daylight only)
PThreads (ctime_r, localtime_r, gmtime_r, and asctime only)

DESCRIPTION

The ctime function converts the calendar time pointed to by *timer* to local time in the form of a string. It is equivalent to the following:

```
asctime(localtime(timer))
```

The `localtime` function converts the calendar time pointed to by *timer* into a broken-down time, expressed as local time. This means that `localtime` adds or subtracts seconds from the calendar time if the locale has defined adjustments for time zone or daylight saving time.

The `gmtime` function converts the calendar time pointed to by *timer* into a broken-down time, expressed as Coordinated Universal Time (CTU).

The functions whose names end with `_r` provide equivalent functionality but with an interface that is safe for multitasked applications. Instead of using internal static buffers, they require the caller to pass in either a buffer of at least 26 bytes (`ctime_r` and `asctime_r`) or a pointer to a structure of type `struct tm` (`localtime_r` and `gmtime_r`) into which the result will be placed.

If you are compiling in extended mode (the default), the objects `timezone`, `daylight`, and `tzname`, and function `tzset` are defined in header `time.h`. If you want to use `timezone`, `daylight`, `tzname`, or `tzset` in a program compiled in strict conformance mode, you must explicitly declare them in your program.

The `asctime` function converts the broken-down time in the structure pointed to by *timeptr* into a string in the form

```
Sun Sep 16 01:03:52 1973\n\0
```

using the equivalent of the following algorithm:

```
char *asctime(const struct tm *timeptr)
{
        static const char wday_name[7][3] = {
                "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
        };
        static const char mon_name[12][3] = {
                "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
        };
        static char result[26];

        sprintf(result, "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n",
                wday_name[timeptr->tm_wday],
                mon_name[timeptr->tm_mon],
                timeptr->tm_mday, timeptr->tm_hour,
                timeptr->tm_min, timeptr->tm_sec,
                1900 + timeptr->tm_year);
        return result;
}
```

The TZ environment variable specifies time zone information. The value of TZ has the following form:

std offset dst offset,*rule*

The expanded form is as follows:

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

std, dst Time zone, standard (*std*) or summer (*dst*). *std* is required; omission of *dst* indicates summer time is not used in this locale. Three or more characters, upper or lowercase, except a leading colon (:), comma, minus (-), plus (+) or ASCII NUL.

offset Difference in hours between local time and Greenich mean time (GMT), in the form *hh*[:*mm*[:*ss*]]. Required following *std*. Following *dst*, defaults to 1 hour. A number preceded by minus (-) indicates a zone east of the prime meridian. The hour (*ff*) should be in the range 0 through 24 and, if specified, the minutes and seconds should be in the range 0 through 59.

rule Indicates when to change to and from summer time, in the following form:

date/*time*,*date*/*time*

In the above format for *rule*, the first *date* specifies when to change from standard to summer time; the second specifies when to change back. Each *time* specifies what time on that date the change occurs. The *date* specification, with a J prefix, indicates the day of the year with a value of 1 through 365; February 29 cannot be specified. A number with no letter prefix is a similar number with range 0 through 365, allowing specification of February 29. Alternatively, *date* can be in the form M*m*.*n*.*d*, indicating the *d*th day of week *n* of month *m*, with ranges $0 \le d \le 6$, $0 \le n \le 5$, and $0 \le m \le 12$. For $n = 5$, the last *d* day of month *m* is used. The *time* value has the same format as *offset*; no leading + or − sign is allowed for *time*.

Setting TZ changes the value of the external variables timezone and daylight. In addition, the time-zone names contained in the external variable

```
char *tzname[2] = { "CST", "CDT" };
```

are set by the function tzset from the environment variable TZ. Function tzset is called by localtime; you may also call tzset explicitly.

The TZ environment variable may also affect functions ctime, asctime, and mktime. If you want these functions to behave in a strictly ANSI conforming way, that is, not to have any effect on timezone, daylight, and tzname, or be affected by their values, you must not have the TZ environment variable present in your environment.

RETURN VALUES

The ctime function returns the pointer returned by the asctime(3C) function, with that broken-down time as argument.

The `localtime` function returns a pointer to that object.

The `gmtime` function returns a pointer to that object, or a null pointer if UTC is not available.

The `asctime` function returns a pointer to the string.

NOTES

CTU is the number of seconds since 00:00:00 GMT Jan 01, 1970. This has been the traditional starting point in UNIX systems and is maintained in this implementation for compatibility, though it is not required by the ANSI Standard. A negative value of calendar time represents time prior to 1970. Any value of calendar time that can be represented by a `long int` is legal, but some values may not have historical significance or may not be convertible to meaningful ASCII representation.

Although the `gmtime` function is defined in terms of calendar time and UTC, it uses any `time_t` value and converts it to a proper `tm` structure. `gmtime` makes no adjustments in its calculations for locale specific variations such as time zone or daylight saving time.

The `asctime` function checks each member of the structure for valid range. If any member is out of range, `asctime` puts asterisks in that part of the output string. Although this is not required by the ANSI standard, it lets you see explicitly where bad values are being passed.

The `asctime`, `ctime`, `gmtime`, and `localtime` functions return values are one of two static objects, a broken-down time structure and an array of `char`. Execution of any of these functions may overwrite the information returned in either of these objects by any of the other functions.

SEE ALSO

`getenv`(3C), `locale`(3C)

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

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

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

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

NAME

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

SYNOPSIS

```
#include <ctype.h>
int isalnum (int c);
int isalpha (int c);
int isascii (int c);
int iscntrl (int c);
int isdigit (int c);
int isgraph (int c);
int islower (int c);
int isprint (int c);
int ispunct (int c);
int isspace (int c);
int isupper (int c);
int isxdigit (int c);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

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

DESCRIPTION

All functions except isascii are defined only for values that are representable as an unsigned char (that is, less than or equal to UCHAR_MAX, defined in limits.h to be 255 for Cray Research systems) or equal to EOF. isascii is defined for all integer values.

The isalnum function tests for any character for which isalpha or isdigit is true.

The isalpha function tests for any character for which isupper or islower is true, or any character that is one of a locally defined set of characters for which none of iscntrl, isdigit, ispunct, or isspace is true. In the C locale, isalpha returns true only for the characters for which isupper or islower is true.

The `isascii` function tests for any ASCII character code less than 0200. The `isascii` macro is defined on all integer values.

The `iscntrl` function tests for any control character. In the C locale, control characters are characters whose values are from `0` (NUL) through `0x1F`, and the character `0x7F` (DEL).

The `isdigit` function tests for any decimal-digit character, 0 through 9, inclusive.

The `isgraph` function tests for any printing character except space (‘ ’).

The `islower` function tests for any character that is a lowercase letter or is one of a locally defined set of characters for which none of `iscntrl`, `isdigit`, `ispunct`, or `isspace` is true.

The `isprint` function tests for any printing character including space (‘ ’). In the C locale, printing characters are characters whose values are from `0x20` (space) through `0x7E` (tilde).

The `ispunct` function tests for any printing character that is neither space (‘ ’) nor a character for which `isalnum` is true.

The `isspace` function tests for any character that is a standard white-space character or is one of a locally defined set of characters for which `isalnum` is false.

In the C locale, the white-space characters are the following:

Space (‘ ’)
Form feed (`\f`)
New-line (`\n`)
Carriage return (`\r`)
Horizontal tab (`\t`)
Vertical tab (`\v`)

The `isupper` function tests for any character that is an uppercase letter or is one of a locally defined set of characters for which none of `iscntrl`, `isdigit`, `ispunct`, or `isspace` is true.

The `isxdigit` function tests for any hexadecimal-digit character, as follows:

```
0  1  2  3  4  5  6  7  8  9
a  b  c  d  e  f
A  B  C  D  E  F
```

NOTES

If the argument to these functions is not in the domain of the function, the result is undefined.

The behavior of functions `isalnum`, `isalpha`, `isascii`, `iscntrl`, `isgraph`, `islower`, `isprint`, `ispunct`, `isspace`, and `isupper` may be affected by the current locale; functions `isdigit` and `isxdigit` are *not* affected by the current locale.

RETURN VALUES

All of these functions return nonzero if, and only if, the value of the argument conforms to that in the description of the function.

SEE ALSO

`locale.h`(3C)

NAME

ctype.h – Library header for character-handling functions

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (except isascii, toascii, _tolower, _toupper)
XPG4 (isascii, toascii, _tolower, _toupper only)

TYPES

None

MACROS

Macros declared in header <ctype.h> are as follows:

```
isalnum       isalpha       isascii       iscntrl       isdigit
isgraph       islower       isprint       ispunct       isspace
isupper       isxdigit      toascii       tolower       _tolower
toupper       _toupper
```

FUNCTION DECLARATIONS

Functions declared in header <ctype.h> are as follows:

```
isalnum       isalpha       isascii       iscntrl       isdigit
isgraph       islower       isprint       ispunct       isspace
isupper       isxdigit      toascii       tolower       _tolower
toupper       _toupper
```

SEE ALSO

locale.h(3C)

NAME

cuserid – Gets character login name of the user

SYNOPSIS

```
#include <stdio.h>
char *cuserid (char *s);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The cuserid function generates a character-string representation of the login name of the owner of the current process. If *s* is a null pointer, this representation is generated in an internal static area, the address of which is returned. Otherwise, *s* is assumed to point to an array of at least L_cuserid characters; the representation is left in this array. The constant L_cuserid is defined in the header file stdio.h.

NOTES

Under certain circumstances, cuserid() may call getudbuid(). Mixing cuserid and getudb*xxx* calls may have unexpected side effects.

RETURN VALUES

If the specified login name or user identification cannot be found, cuserid returns a null pointer; if *s* is not a null pointer, a null character (\0) is placed at s[0].

SEE ALSO

getlogin(3C), getpwent(3C)

NAME

daemon – Run an application in the background

SYNOPSIS

daemon(*int nochdir, int noclose*);

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The daemon function is for programs wishing to detach themselves from the controlling terminal and run in the background as system daemons.

Unless the argument *nochdir* is non-zero, daemon(3C) will change the current working directory to the root directory /.

Unless the argument *noclose* is non-zero, daemon(3C) will redirect standard input, standard output, and standard error to /dev/null.

ERRORS

The function daemon(3C) may fail and set *errno* for any of the errors specified for the library functions fork(2) and setsid(2).

SEE ALSO

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

HISTORY

The daemon(3C) function first appeared in 4.4BSD.

NAME

`date_time` – Introduction to date and time functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The date and time functions provide various means for manipulating date and time and for converting date and time to various forms.

The two basic forms for the date and time are calendar time and broken-down time. *Calendar time* is a single value representing a date and time. In the Cray Research, Inc. implementation, it is a signed long integer with the number of seconds since January 1, 1970, Coordinated Universal Time (CTU). The value of the calendar time may be zero or negative for time on this date.

Broken-down time is a structure of values representing a date and time. It is equivalent to the calendar time except that the values of the members of the structure separately specify the year, month, day, and so on. The structure is described under the header `time.h`.

A variation of broken-down time is *local time*, which is the date and time adjusted for the difference between the time under local customs and the universal coordinated time. These local customs include the time zone and daylight saving time.

The algorithms used for converting times from one form to the other follow the rules for the Gregorian calendar, even though this is not historically correct for times before the Gregorian calendar was adopted or for locales that do not follow the Gregorian calendar.

ASSOCIATED HEADERS

`time.h`

ASSOCIATED FUNCTIONS

Time Manipulation Functions

Function	Description
`clock`(3C)	Reports CPU time used
`cpused`(3C)	Gets CPU time in real-time clock (RTC) ticks
`difftime`(3C)	Finds difference between two calendar times
`mktime`(3C)	Converts local time to calendar time
`rtclock`(3C)	Gets current RTC reading
`time`(3C)	Determines the local calendar time

Time Conversion Functions

Function	Description
ascftime	Formats time information in a character string (see strftime(3C))
asctime	Converts broken-down time to string (see ctime(3C))
asctime_r	Converts broken-down time to string (see ctime(3C))
cftime	Formats time information in a character string (see strftime(3C))
ctime(3C)	Converts calendar time to local time
ctime_r(3C)	Converts calendar time to local time
gmtime	Converts calendar time to broken-down time (see ctime(3C))
gmtime_r	Converts calendar time to broken-down time (see ctime(3C))
localtime	Converts calendar time to broken-down time (see ctime(3C))
localtime_r	Converts calendar time to broken-down time (see ctime(3C))
strftime(3C)	Formats time information from broken-down time to a character string
strptime(3C)	Formats time information from a character string to broken-down time

SEE ALSO

locale.h(3C)

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

NAME

dbmclose, dbminit, fetch, store, delete, firstkey, nextkey – Provides database subfunctions

SYNOPSIS

```
#include <rpcsvc/dbm.h>
int dbminit (char *file);
datum fetch (datum key);
int store (datum key, datum content);
int delete (datum key);
datum firstkey (void);
datum nextkey (datum key);
int dbmclose (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

Functions dbminit, fetch, store, delete, firstkey, and nextkey maintain *key*/*content* pairs in a database. These functions handle very large databases, up to a billion blocks, and access a keyed item in one or two file system accesses.

The *key* and *content* arguments are described by the following datum type definition:

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

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

The database is stored in two files. One file is a directory containing a bit map and has .dir as its suffix. The second file contains all data and has .pag as its suffix. Before a database can be accessed, it must be opened by dbminit. At the time of this call, the files *file*.dir and *file*.pag must exist. (You can create an empty database by making zero-length .dir and .pag files.)

Once the database is open, the `fetch` function accesses data stored under a key, and the `store` function places data under a key. The `delete` function removes a key and its associated contents. You can make a linear pass through all keys in a database, in an (apparently) random order, by using `firstkey` and `nextkey`. The `firstkey` function returns the first key in the database. Beginning with any key, `nextkey` returns the next key in the database. The following code traverses the database:

```
for (key = firstkey(); key.dptr != NULL; key = nextkey(key))
```

You must close a database before opening a new one. To close a database, call `dbmclose`.

NOTES

The `.pag` file contains holes; therefore its apparent size is about four times its actual content. Older UNIX systems might create real file blocks for these holes when `touch`(1) is executed. The `.pag` files cannot be copied by normal means (`cp`(1), `cat`(1), `tar`(1), `ar`(1)) without filling in the holes.

The *dptr* pointers returned by these subfunctions point into static storage that is changed by subsequent calls.

The sum of the sizes of a *key*/*content* pair must not exceed the internal block size (currently 1024 bytes). Moreover, all *key*/*content* pairs that hash together must fit on a single block. The `store` function returns an error if a disk block fills with inseparable data.

The `delete` function does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by `firstkey` and `nextkey` depends on a hashing function.

There are no interlocks and no reliable cache flushing; thus, concurrent updating and reading is risky.

RETURN VALUES

A zero return indicates that there are no errors. An integer with a negative value (such as –1) indicates an error. A type `datum` return indicates an error with a null (0) *dptr*.

SEE ALSO

`ar`(1), `cat`(1), `cp`(1), `tar`(1), `touch`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

`difftime` – Finds difference between two calendar times

SYNOPSIS

```
#include <time.h>
double difftime (time_t time1, time_t time0);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The `difftime` function computes the difference between two calendar times, *time1* and *time0*.

RETURN VALUES

The `difftime` function returns the difference expressed in seconds as a value of type `double`.

NAME

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

SYNOPSIS

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

DIR *opendir (const char *file);

struct dirent *readdir (DIR *dirp);

int readdir_r (DIR *dirp, struct dirent *entry, struct dirent **result);

long telldir (DIR *dirp);

void seekdir (DIR *dirp, long loc);

void rewinddir (DIR *dirp);

int closedir (DIR *dirp);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX (except telldir and seekdir)
XPG4 (telldir and seekdir only)
PThreads (readdir_r only)

DESCRIPTION

The opendir function opens the directory named by *file* and associates a directory stream, *dirp*, with it. The opendir function returns a pointer to be used to identify the directory stream *dirp* in subsequent operations. A null pointer is returned if *file* cannot be accessed or is not a directory, or if it cannot obtain enough memory (using malloc(3C)) or a buffer for the directory entries. A successful call to any of the exec(2) functions will close any directory streams that are open in the calling process.

The readdir function returns a pointer to the next active directory entry *dirp*. No inactive entries are returned. It returns null upon reaching the end of the directory or upon detecting an invalid location in the directory.

The readdir_r function provides functionality equivalent to the readdir function but with an interface that is safe for multitasked applications. Storage for the directory entry, *dirent*, is provided by the caller using the *entry* argument, which must be of at least the following value:

```
sizeof(struct dirent) + fnamemax
```

In this expression, *fnamemax* is the maximum size of a file name, which can be determined using the `pathconf` or `fpathconf` interface.

The `readdir_r` function initializes the structure referenced by `entry` to the correct values, and stores a pointer to this structure at the location referenced by *result*. On successful return, **result* should point to `entry`. At end-of-directory, this pointer is NULL. The `readdir_r` function returns zero on success, or nonzero if an error occurs.

The `telldir` function returns the current location associated with the directory stream *dirp*.

The `seekdir` function sets the position of the next `readdir` operation on the directory stream *dirp*. The new position reverts to the one associated with *dirp* when the `telldir` operation from which *loc* was obtained is performed. Values returned by `telldir` are good only if the directory has not changed due to compaction or expansion.

The `rewinddir` function resets the position of the named directory stream, *dirp*, to the beginning of the directory.

The `closedir` function closes the directory stream *dirp* and frees the `DIR` structure.

On error, `opendir` puts one of the following values into `errno`, defined in header `errno.h`:

Error Code	Description
`ENOTDIR`	A component of *file* is not a directory.
`EACCES`	A component of *file* denies search permission.
`EMFILE`	The maximum number of file descriptors are currently open.
`EFAULT`	Argument *file* points outside the allocated address space.

On error, `readdir` and `readdir_r` put one of the following values into `errno`, defined in header `errno.h`:

Error Code	Description
`ENOENT`	The current file pointer for the directory is not located at a valid entry.
`EBADF`	The file descriptor determined by the `DIR` stream is no longer valid. This results if the DIR stream has been closed.

EXAMPLES

The following example shows source code that searches a directory for the entry `name`:

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

#define FOUND 1
#define NOT_FOUND 0

findname(char *name)
{
        DIR *dirp;
        struct dirent *dp;

        dirp = opendir(".");
        while ((dp = readdir(dirp)) != NULL) {
                if (strcmp(dp->d_name, name) == 0) {
                        (void) closedir(dirp);
                         return FOUND;
                }
        }
        (void) closedir(dirp);
        return NOT_FOUND;
}
```

SEE ALSO

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

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

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

NAME

`div`, `ldiv`, `lldiv` – Computes integer or long integer quotient and remainder

SYNOPSIS

```
#include <stdlib.h>
div_t div (int numer, int denom);
ldiv_t ldiv (long int numer, long int denom);
lldiv_t lldiv (long long int numer, long long int denom);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The `div`, `ldiv`, and `lldiv` function computes the quotient and remainder of the division of the numerator *numer* by the denominator *denom*. If the division is inexact, the resulting quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be represented, the behavior is undefined; otherwise, `quot * denom + rem` equals *numer*.

RETURN VALUES

The `div` function returns a structure of type `div_t`, which is defined in the header file `stdlib.h`, comprising both the quotient and the remainder.

The `ldiv` function is similar to the `div` function, except that the argument and the members of the returned structure (which has type `ldiv_t`) all have type `long int`.

The `lldiv` function is similar to the `div` function, except that the argument and the members of the returned structure (which has type `lldiv_t`) all have type `long long int`.

NAME

dmf_offline, dmf_hashandle, dmf_vendor – Determines migrated status

SYNOPSIS

```
int dmf_offline(struct stat *st);
int dmf_hashandle(struct stat *st);
int dmf_vendor(int portno);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The dmf_offline, dmf_hashandle, and dmf_vendor functions provide information about migrated files on UNICOS file systems. They apply to the different migration systems supported by UNICOS. The migration systems supported are the Cray Data Migration Facility (DMF) and FILESERV.

The functions are as follows:

- dmf_offline determines whether a migrated file is offline or not. A file is considered to be offline if it has a valid copy offline and no copy of the data online.
- dmf_hashandle determines whether a file has a migration handle or not.
- dmf_vendor determines which data migration vendor, if any, owns the port specified in *portno*.

RETURN VALUES

The dmf_offline function returns 1 if the file is offline, 0 if it is not.

The dmf_hashandle function returns 1 if the file has a handle and the vendor is DMF, 2 if the file has a handle and the vendor is FILESERV, 0 if it does not have a handle, and –1 if the type of handle or vendor cannot be determined.

The dmf_vendor function returns 1 if the vendor is DMF, 2 if the vendor is FILESERV, and 0 if the vendor cannot be determined.

SEE ALSO

Cray Data Migration Facility (DMF) Administrator's Guide, Cray Research publication SG–2135

NAME

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

SYNOPSIS

```
#include <stdlib.h>
double drand48 (void);
double erand48 (unsigned short xsubi[3]);
long lrand48 (void);
long nrand48 (unsigned short xsubi[3]);
long mrand48 (void);
long jrand48 (unsigned short xsubi[3]);
void srand48 (long seedval);
unsigned short (**seed48 (unsigned short seed16v[3]);
void lcong48 (unsigned short param[7]);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

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

Functions drand48 and erand48 return nonnegative, floating-point values uniformly distributed over the interval (0.0,1.0).

Functions lrand48 and nrand48 return nonnegative long integers uniformly distributed over the interval $(0, 2^{31})$.

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

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

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

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

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

$$a = 5\text{DEECE66D}_{16} = 273673163155_8$$
$$c = \text{B}_{16} = 13_8.$$

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

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

The initialization function `srand48` sets the high-order 32 bits of X_i to the 32 bits contained in its argument. The low-order 16 bits of X_i are set to the arbitrary value $330\text{E}_{16}(31416_8)$.

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

The initialization function `lcong48` lets you specify the initial X_i, the multiplier value *a*, and the addend value *c*. Argument array elements `param[0-2]` specify X_i, elements `param[3-5]` specify the multiplier *a*, and element `param[6]` specifies the 16-bit addend *c*. After `lcong48` has been called, a subsequent call to either `srand48` or `seed48` restores the "standard" multiplier and addend values, *a* and *c*, specified previously.

SEE ALSO

rand(3C)

NAME

dup2 – Duplicates an open file descriptor

SYNOPSIS

```
#include <unistd.h>
int dup2 (int oldfd, int newfd);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX

DESCRIPTION

Function dup2 duplicates an open file descriptor, *oldfd*, onto a new file descriptor, *newfd*, using the fcntl(2) system call. Argument *newfd* must be a nonnegative integer less than NOFILE. The dup2 function causes *newfd* to refer to the same file as *oldfd*. If *newfd* already refers to an open file, that file is first closed, as if the close(2) system call had been performed.

The dup2 function is roughly equivalent to the following (checking is performed that is not shown here):

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

dup2(oldfd, newfd)
int oldfd, newfd;
{
    if (oldfd != newfd)
          (void) close(newfd);
    return(fcntl(oldfd, F_DUPFD, newfd));
}
```

RETURN VALUES

On successful completion, dup2 returns the file descriptor, a nonnegative integer. If dup2 does not complete successfully, it returns –1 and sets errno to indicate the error, as follows:

Error	Description
EBADF	The *oldfd* argument is not a valid open file descriptor.
EMFILE	NOFILE number of files are currently open.

SEE ALSO

`errno.h`(3C)

`close`(2), `creat`(2), `dup2`(3C), `exec`(2), `fcntl`(2), `open`(2), `pipe`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

ecvt, fcvt, gcvt – Converts a floating-point number to a string

SYNOPSIS

```
#include <stdlib.h>
char *ecvt (double value, int ndigit, int *decpt, int *sign);
char *fcvt (double value, int ndigit, int *decpt, int *sign);
char  *gcvt  (double value, int ndigit, char *buf);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

AT&T extension

DESCRIPTION

The ecvt function converts *value* to a null-terminated string of *ndigit* digits and returns a pointer to the string. The high-order digit is nonzero, unless *value* is 0. The low-order digit is rounded. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt* (*negative* means to the left of the returned digits). The decimal point is not included in the returned string. If the sign of the result is negative, the word pointed to by *sign* is nonzero; otherwise, it is 0.

The fcvt function is identical to ecvt, except that the correct digit has been rounded for printf %f (Fortran F-format) output of the number of digits specified by *ndigit*.

The gcvt function converts *value* to a null-terminated string in the array pointed to by *buf* and returns *buf*. It attempts to produce *ndigit* significant digits in Fortran F-format if possible (otherwise in E-format), ready for printing. A minus sign (if there is one) or a decimal point is included as part of the returned string. Trailing 0's are suppressed.

For machines with IEEE arithmetic, all of these functions return NaN for "not-a-number" and Inf for "infinity."

NOTES

The values returned by ecvt and fcvt point to a single static data array that is overwritten by each call.

SEE ALSO

printf(3C)

NAME

erf, erfc – Returns error function and complementary error function

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The erf function returns the error function of *x*, defined as $\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$. The erf function returns the error function of *x*, defined as 2 divided by the square root of *pi* times the integral from 0 to *x* of *e* raised to the power of (-*t*) squared times *dt*.

The erfc function, which returns 1.0 - erf(*x*), is provided because of the extreme loss of relative accuracy if erf(*x*) is called for a large *x* and the result is subtracted from 1.0 (for example, when *x* = 5, 12 places are lost).

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

RETURN VALUES

On Cray MPP systems and CRAY T90 systems with IEEE arithmetic, erf(*NaN*) and erfc(*NaN*) returns NaN and errno is set to EDOM.

SEE ALSO

exp(3C)

NAME

errno.h – Library header for reporting error conditions

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

MACROS

The following macros are defined in errno.h:

Macro	Description
errno	An identifier that expands to a modifiable lvalue that has type int, the value of which is set to a positive error number by several library functions and UNICOS system calls. ISO/ANSI.

The following macros, which are defined in sys/errno.h, expand into integral constant expressions with distinct nonzero values suitable for use in #if preprocessing directives. These are the standard defined macros; there are many other macros defined in sys/errno.h. See intro(2) for a list of the UNICOS errors. Unless noted, all macros conform with the POSIX standard.

Macro	Description
E2BIG	Argument list too big
EACCES	Permission denied
EAGAIN	Resource unavailable, try again
EBADF	Bad file number
EBUSY	Device or resource busy
ECHILD	No child processes
EDEADLK	Resource deadlock would occur
EDOM	Argument out of domain of function. ISO/ANSI standard.
EEXIST	File exists
EFAULT	Bad address
EFBIG	File too large
EIDRM	Identifier removed. X/Open standard.
EILSEQ	Illegal byte sequence. X/Open standard.
EINTR	Interrupted function

EINVAL	Invalid argument
EIO	I/O error
EISDIR	Is a directory
EMFILE	Too many open files
EMLINK	Too many links
ENAMETOOLONG	Filename too long
ENFILE	File table overflow
ENODEV	No such device
ENOENT	No such file or directory
ENOEXEC	Executable file format error
ENOLCK	No locks available
ENOMEM	Not enough space
ENOMSG	No message of desired type. X/Open standard.
ENOSPC	No space left on device
ENOSYS	Functionality not supported
ENOTDIR	Not a directory
ENOTEMPTY	Directory not empty
ENOTTY	Inappropriate I/O control operation
ENXIO	No such device or address
EPERM	Operation not permitted
EPIPE	Broken pipe
ERANGE	Result not representable in return type. ISO/ANSI standard.
EROFS	Read-only file system
ESPIPE	Invalid seek
ESRCH	No such process
ETXTBSY	Text file busy. X/Open standard.
EXDEV	Cross-device link

FUNCTION DECLARATIONS

None

NOTES

For multitasking, `errno` must be defined as a per-task variable. This is done by making `errno` a macro that dereferences a per-task pointer returned by the `_Errno` function. This is shown in an excerpt from `errno.h`:

```
#define errno (*_Errno())
extern int errno;
```

To be ISO/ANSI conformant, you must include `errno.h` to use `errno`. However, the POSIX 1003.1 standard allows users to simply declare `extern int errno` in their program without including `errno.h`; while this is allowed, its usage is discouraged, and programs doing this will not work with multitasking.

Actual errors are defined in header `sys/errno.h`, which is included automatically by `errno.h`. See `intro`(2) for a list of the UNICOS errors.

SEE ALSO

`prog_diag`(3C)

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

NAME

EVASGN – Identifies an integer variable to be used as an event

SYNOPSIS

CALL EVASGN(*name* [,*value*])

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

EVASGN identifies an integer variable that the program will use as an event. You must call this routine for an event variable before that variable is used with any of the other event routines. The multitasking library sets the initial state of the event to be cleared. A data statement can initialize the event to the value in the optional argument so that the event can be assigned in a routine. The first call assigns the event, and further calls are ignored.

The following is a list of valid arguments for this routine:

Argument	Description
name	Name of an integer variable to be used as an event. The library stores an identifier into this variable; you should not modify this variable after the call to EVASGN.
value	The initial integer value of the event variable. EVASGN stores an identifier into the variable only if that variable still contains the value. If you do not specify *value*, an identifier is stored unconditionally into the variable.

NOTES

For SPARC systems, the *value* parameter is optional, and EVASGN is not predeclared (not intrinsic). Therefore, if a call is made to it with only the *name* parameter, EVASGN must be declared with an INTERFACE block in the calling module.

EXAMPLES

```
      PROGRAM MULTI
      INTEGER EVSTART,EVDONE
      COMMON /EVENTS/ EVSTART,EVDONE
C     ...
      CALL EVASGN (EVSTART)
      CALL EVASGN (EVDONE)
C     ...
      END
      SUBROUTINE SUB1
      INTEGER EVENT1
      COMMON /EVENT1/ EVENT1
      DATA EVENT1 /-1/
C     ...
      CALL EVASGN (EVENT1,-1)
C     ...
      END
```

NAME

EVCLEAR – Clears an event and returns control to the calling task

SYNOPSIS

CALL EVCLEAR(*event*)

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

EVCLEAR clears an event and returns control to the calling task. When the task is clear, all tasks subsequently performing EVWAIT(3F) calls must wait.

The following is a valid argument for this routine:

Argument	**Description**
event	Name of an integer variable used as an event. After an event is posted by a call to EVPOST(3F), the posted condition remains until EVCLEAR is called. The typical use of EVCLEAR is to call it immediately after the call to EVWAIT to indicate that the posting of the event was detected.

EXAMPLES

```
      PROGRAM MULTI
      INTEGER EVSTART,EVDONE
      COMMON /EVENTS/ EVSTART,EVDONE
C     ...
      CALL EVASGN (EVSTART)
      CALL EVASGN (EVDONE)
C     ...
      CALL EVPOST (EVSTART)
      END

      SUBROUTINE MULTI2
      INTEGER EVSTART,EVDONE
      COMMON /EVENTS/ EVSTART,EVDONE
C     ...
      CALL EVWAIT (EVSTART)
      CALL EVCLEAR (EVSTART)
C     ...
      END
```

SEE ALSO

EVPOST(3F), EVWAIT(3F), multif(3F)

NAME

EVPOST – Posts an event and returns control to the calling task

SYNOPSIS

CALL EVPOST(*event*)

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

EVPOST posts an event and returns control to the calling task. Posting the event allows any other tasks waiting on that event to resume execution, but this is transparent to the task calling EVPOST. Posting a posted event has no effect (posts are not queued) and should be avoided.

The following is a valid argument for this routine:

Argument	**Description**
event	Name of an integer variable used as an event.

EXAMPLES

```
      PROGRAM MULTI
      INTEGER EVSTART,EVDONE
      COMMON /EVENTS/ EVSTART,EVDONE
C     ...
      CALL EVASGN (EVSTART)
      CALL EVASGN (EVDONE)
C     ...
      CALL EVPOST (EVSTART)
      END
```

SEE ALSO

EVCLEAR(3F), EVWAIT(3F), MULTIF(3F)

NAME

EVREL – Releases the identifier assigned to an event

SYNOPSIS

CALL EVREL(*name*)

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

EVREL releases the identifier assigned to an event.

The following is a valid argument for this routine:

Argument	Description
name	Name of an integer variable used as an event.

If tasks are currently waiting for this event to be posted, an error results. This routine is useful primarily in detecting erroneous uses of an event outside the region the program has planned for it. The event variable can be reused following another call to EVASGN(3F).

EXAMPLES

```
      PROGRAM MULTI
      INTEGER EVSTART,EVDONE
      COMMON  /EVENTS/ EVSTART,EVDONE
C     ...
      CALL EVASGN (EVSTART)
      CALL EVASGN (EVDONE)
C     ...
      CALL EVPOST (EVSTART)
C     ...
C     EVSTART WILL NOT BE USED FROM NOW ON
      CALL EVREL (EVSTART)
C     ...
      END
```

SEE ALSO

EVASGN(3F)

NAME

EVTEST – Returns the state of an event

SYNOPSIS

```
LOGICAL  EVTEST
return=EVTEST(event)
```

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

EVTEST returns the logical state of an event.

The following is a list of valid arguments for this routine:

Argument	Description
return	A logical .TRUE. if the event is posted. A logical .FALSE. if the event is not posted. The event variable's state is unaffected by a call to EVTEST.
event	Name of an integer variable used as an event.

NOTES

EVTEST and *return* must be declared as type LOGICAL in the calling module.

SEE ALSO

EVCLEAR(3F), EVPOST(3F), EVWAIT(3F), multif(3F)

NAME

EVWAIT – Delays the calling task until the specified event is posted

SYNOPSIS

```
CALL EVWAIT(event)
```

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

EVWAIT delays the calling task until the specified event is posted by a call to EVPOST(3F). If the event is already posted, the task resumes execution without waiting. EVWAIT does not change the state of the event.

The following is a valid argument for this routine:

Argument	Description
event	Name of an integer variable used as an event

EXAMPLES

```
      SUBROUTINE MULTI2
      INTEGER EVSTART,EVDONE
      COMMON /EVENTS/ EVSTART,EVDONE
C     ...
      CALL EVWAIT (EVSTART)
C     ...
      END
```

SEE ALSO

EVCLEAR(3F), EVPOST(3F), multif(3F)

NAME

`exit` – Terminates a program

SYNOPSIS

```
#include <stdlib.h>
void exit (int status);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The `exit` function causes normal program termination to occur. If more than one call to the `exit` function is executed by a program, the behavior is undefined.

First, all functions registered by the `atexit` function are called, in the reverse order of their registration. Each function is called as many times as it was registered.

Next, all open output streams are flushed, all open streams are closed, and all files created by the `tmpfile` function are removed.

Finally, control is returned to the host environment by using the `_exit`(2) system call.

RETURN VALUES

The `exit` function does not return to its caller.

SEE ALSO

`atexit`(3C)

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

NAME

exp, expf, expl, cexp, log, logf, logl, clog, log10, log10f, log10l – Determines exponential and logarithm values

SYNOPSIS

```
#include <math.h>
#include <complex.h> (functions cexp and clog only)
double exp (double x);
float expf (float x);
long double expl (long double x);
double complex cexp (double complex x);
double log (double x);
float logf (float x);
long double logl (long double x);
double complex clog (double complex x);
double log10 (double x);
float log10f (float x);
long double log10l (long double x);
```

IMPLEMENTATION

All Cray Research systems (exp, cexp, log, clog, log10 only)
Cray MPP systems (expf, logf, log10f only)
Cray PVP systems (expl, logl, log10l only)

STANDARDS

ISO/ANSI (exp, log, log10 only)
CRI extension (all others)

DESCRIPTION

The exp, expf, expl, and cexp functions return the exponential of *x* (*e* raised to the power of *x*). A range error occurs if the magnitude of *x* is too large.

The log, logf, logl, and clog functions return the natural logarithm of *x*. A domain error occurs if the value of *x* is negative. A range error occurs if the value of *x* is 0.

The log10, log10f, and log10l functions return the base 10 logarithm of *x*. A domain error occurs if the value of *x* is negative. A range error occurs if the value of *x* is 0.

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

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

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

RETURN VALUES

The following describes the action taken for certain error conditions when a program is compiled with -hstdc or -hmatherror=errno on Cray MPP systems and CRAY T90 systems with IEEE arithmetic:

- The functions exp(*NaN*), expl(*NaN*), cexp(*NaN*), log(*NaN*), logl(*NaN*), log10(*NaN*), log101(*NaN*), and clog(*NaN*) return NaN and errno is set to EDOM.
- The value returned by the functions in the following table when a domain error occurs can be selected by the environment variable CRI_IEEE_LIBM. The second column describes what is returned when CRI_IEEE_LIBM is not set, or is set to a value other than 1. The third column describes what is returned 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
log(*x*), where *x* is less than zero	-HUGE_VAL	NaN
logl(*x*), where *x* is less than zero	-HUGE_VALL	NaN
logf(*x*), where *x* is less than zero	-HUGE_VALF	NaN
clog(.0+0.0*1.0*i*)	(0.0+0.0*1.0*i*)	(NaN+NaN*1.0*i*)
log10(*x*), where *x* is less than zero	-HUGE_VAL	NaN
log101(*x*), where *x* is less than zero	-HUGE_VALL	NaN
log10f(*x*), where *x* is less than zero	-HUGE_VALF	NaN

SEE ALSO

errno.h(3C)

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

NAME

`fclose`, `fflush` – Closes or flushes a stream

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The `fclose` function causes any buffered data for the specified *stream* to be written out and the stream to be closed. If the associated buffer was allocated automatically, it is deallocated. An `fclose` call is performed automatically for all open files when `exit`(2) is called.

If *stream* is an output stream or update stream in which the most recent operation was not input, the `fflush` function causes any buffered data for the specified *stream* to be written to the associated file; otherwise, the behavior is undefined. The stream remains open. If stream is a null pointer, the `fflush` function performs this flushing action on all streams for which the behavior is defined above.

RETURN VALUES

If an error is detected or if the file was already closed, these functions return `EOF`; otherwise, they return 0.

FORTRAN EXTENSIONS

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

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

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

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

SEE ALSO

`fopen`(3C), `setbuf`(3C), `ungetc`(3C)

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

NAME

feclearexcept, fegetexceptflag, feraiseexcept, fesetexceptflag, fetestexcept – Manages floating-point exception flags

SYNOPSIS

```
#include <fenv.h>

void feclearexcept(int excepts);
void feraiseexcept(int excepts);
void fegetexceptflag(fexcept_t *flagp, int excepts);
void fesetexceptflag(const fexcept_t *flagp, int excepts);
int fetestexcept(int excepts);
```

IMPLEMENTATION

CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

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

DESCRIPTION

These functions provide access to the exception flags. The `int` input argument for the functions represents a subset of floating-point exceptions and can be constructed by bitwise ORs of the exception macros, such as `FE_OVERFLOW | FE_INEXACT`. For argument values other than the exception macros, the behavior of these functions is undefined.

The `feclearexcept` function clears the exceptions represented by its argument. The argument *excepts* represents exceptions as a bitwise OR of exception macros.

The `fegetexceptflag` function stores the representation of the exception flags indicated by the argument *excepts* through the pointer argument *flagp*.

The `feraiseexcept` function raises the exceptions represented by its argument. The effect is similar to that of exceptions raised by arithmetic operations. Hence, enabled traps are taken for exceptions raised by this function. The argument *excepts* represents exceptions as a bitwise OR of exception macros. The function is not restricted to accepting only IEEE-valid coincident expressions for atomic operations, which means the function can be used to raise exceptions accrued over several operations. If *excepts* represents valid coincident exceptions for atomic operations (namely, `FE_OVERFLOW` and `FE_INEXACT` or `FE_UNDERFLOW` and `FE_INEXACT`), `FE_OVERFLOW` and `FE_UNDERFLOW` are raised before `FE_INEXACT`; otherwise, the order in which these exceptions are raised is unspecified. On CRAY T90 and CRAY T3E systems with IEEE floating-point hardware, raising the overflow or underflow exception also causes the inexact exception to be raised.

The `fesetexceptflag` function sets the complete status for those exception flags indicated by the argument *excepts*, according to the representation in the object pointed to by *flagp*. The value of **flagp* must have been set by a previous call to `fegetexceptflag`, or the effect on the indicated exception flags is undefined. This function does not raise exceptions; it only sets the state of the flags.

The `fetestexcept` function determines which of a specified subset of the exception flags are currently set. The *excepts* argument specifies (as a bitwise OR of the exception macros) the exception flags to be queried. This mechanism allows testing several exceptions with just one function call.

RETURN VALUES

The `fetestexcept` function returns the bitwise OR of the exception macros corresponding to the currently set exceptions included in the *excepts* argument.

EXAMPLES

The following example calls `f` if `FE_INVALID` is set and `g` if `FE_OVERFLOW` is set:

```
#include <fenv.h>
int set_excepts;
/*...*/
set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
if (set_excepts & FE_INVALID) f();
if (set_excepts & FE_OVERFLOW) g();
```

SEE ALSO

`fenv.h`(3C) for valid macros to serve as arguments

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

NAME

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

SYNOPSIS

```
#include <fenv.h>

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

IMPLEMENTATION

CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

Cray Research extensions to IEEE Std 754-1985

DESCRIPTION

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

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

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

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

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

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

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

RETURN VALUES

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

SEE ALSO

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

NAME

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

SYNOPSIS

```
#include <fenv.h>

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

IMPLEMENTATION

CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

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

DESCRIPTION

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

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

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

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

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

RETURN VALUES

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

EXAMPLES

The following example hides spurious underflow exceptions:

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

SEE ALSO

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

NAME

`fegetround`, `fesetround` – Manage the rounding direction modes

SYNOPSIS

```
#include <fenv.h>

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

IMPLEMENTATION

CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

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

DESCRIPTION

The `fegetround` function gets the current rounding direction.

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

EXAMPLES

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

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

RETURN VALUES

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

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

SEE ALSO

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

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

NAME

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

IMPLEMENTATION

CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

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

DESCRIPTION

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

The following conventions are followed by the Cray Research implementation:

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

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

TYPES

The `fenv.h` header file defines the following types:

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

MACROS

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

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

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

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

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

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

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

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

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

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

FUNCTIONS

The following functions are described on separate man pages:

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

SEE ALSO

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

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

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

NAME

ferror, feof, clearerr – Returns indication of stream status

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

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

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

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

NOTES

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

RETURN VALUES

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

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

SEE ALSO

fopen(3C)

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

NAME

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

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

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

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

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

RETURN VALUES

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

SEE ALSO

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

NAME

file – Introduction to file system and directory functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

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

ASSOCIATED HEADERS

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

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

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

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

ASSOCIATED FUNCTIONS

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

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

SEE ALSO

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

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

NAME

fileno – Returns integer file descriptor associated with stream

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX

DESCRIPTION

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

NOTES

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

FORTRAN EXTENSION

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

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

SEE ALSO

fopen(3C)

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

NAME

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

TYPES

None

MACROS

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

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

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

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

The macros and definitions are shown in the following table:

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

Macro	Definition

`FLT_MANT_DIG`
`DBL_MANT_DIG`
`LDBL_MANT_DIG`
	`float`, `double`, or `long double` value that is the number of base-`FLT_RADIX` digits in the floating-point significand, *p*.

`FLT_DIG`
`DBL_DIG`
`LDBL_DIG`
	`float`, `double`, or `long double` value that is the number of decimal digits, *q*, such that any floating-point number with *q* decimal digits can be rounded into a floating-point number with *p* radix *b* digits and back again without change to the *q* decimal digits.
	$\lfloor (p-1) \times \log_{10} b \rfloor + \begin{cases} 1 & \text{if } b \text{ is a power of 10} \\ 0 & \text{otherwise} \end{cases}$

`FLT_MIN_EXP`
`DBL_MIN_EXP`
`LDBL_MIN_EXP`
	`float`, `double`, or `long double` value that is the minimum negative integer such that `FLT_RADIX` raised to that power minus 1 is a normalized floating-point number, $e_{\min}$

`FLT_MIN_10_EXP`
`DBL_MIN_10_EXP`
`LDBL_MIN_10_EXP`
	`float`, `double`, or `long double` value that is the minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers. $\lceil \log_{10} b^{e_{\min}-1} \rceil$

`FLT_MAX_EXP`
`DBL_MAX_EXP`
`LDBL_MAX_EXP`
	`float`, `double`, or `long double` value that is the maximum integer such that `FLT_RADIX` raised to that power minus 1 is a representable finite floating-point number, $e_{\max}$

`FLT_MAX_10_EXP`
`DBL_MAX_10_EXP`
`LDBL_MAX_10_EXP`
	`float`, `double`, or `long double` value that is the maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers. $\lfloor \log_{10}((1 - b^{-p}) \times b^{e_{\max}}) \rfloor$

`FLT_MAX`
`DBL_MAX`
`LDBL_MAX`
	`float`, `double`, or `long double` value that is the maximum representable finite floating-point number. $(1 - b^{-p}) \times b^{e_{\max}}$

`FLT_EPSILON`
`DBL_EPSILON`
`LDBL_EPSILON`
	`float`, `double`, or `long double` value that is the difference between 1.0 and the least value greater than 1.0 that is representable in the given floating point type, b^{1-p}.

`FLT_MIN`
`DBL_MIN`
`LDBL_MIN`
	`float`, `double`, or `long double` value that is the minimum normalized positive floating-point number. $b^{e_{\min}-1}$

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

Macro	CRI format values	IEEE format values (Cray MPP/CRAY T90)
FLT_ROUNDS	0	1
FLT_RADIX	2	2
FLT_MANT_DIG	47	24 / 53
DBL_MANT_DIG	47	53
LDBL_MANT_DIG	94	53 / 113
FLT_DIG	13	6 / 15
DBL_DIG	13	15
LDBL_DIG	27	15 / 33
FLT_MIN_EXP	– 8189	– 125 / – 1021
DBL_MIN_EXP	– 8189	– 1021
LDBL_MIN_EXP	– 8189	– 1021 / – 16381
FLT_MIN_10_EXP	– 2465	– 37 / – 307
DBL_MIN_10_EXP	– 2465	– 307
LDBL_MIN_10_EXP	– 2465	– 307 / – 4931
FLT_MAX_EXP	8190	128 / 1024
DBL_MAX_EXP	8190	1024
LDBL_MAX_EXP	8190	1024 / 16384
FLT_MAX_10_EXP	2465	38 / 308
DBL_MAX_10_EXP	2465	308
LDBL_MAX_10_EXP	2465	308 / 4932
FLT_MAX	2.726870339048517e2465	3.4028234663852886e+38F / 1.7976931348623158e308
DBL_MAX	2.726870339048517e2465	1.7976931348623158e+308
LDBL_MAX	2.726870339048517e2465L	1.7976931348623158e308 / 1.189731495357231765085759326628007016E+4932L
FLT_EPSILON	7.10542735760100e– 15	1.1920928955078125e– 7F / 2.2204460492503131e– 16
DBL_EPSILON	7.10542735760100e– 15	2.2204460492503131e– 16
LDBL_EPSILON	2.52435489670723777731753140 89e– 29L	2.2204460492503131e– 16 / 1.92592994438723585305597794258492 7319E-34L
FLT_MIN	1/2.726870339048517e2465	1.1754943508222875e– 38F / 2.2250738585072014e– 308

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

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

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

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

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

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

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

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

FUNCTION DECLARATIONS

None

SEE ALSO

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

NAME

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

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

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

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

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

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

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

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

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

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

NOTES

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

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

RETURN VALUES

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

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

FILES

/usr/include/sys/file.h

SEE ALSO

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

NAME

flockfile, ftrylockfile, funlockfile – Locks file stream

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

PThreads

DESCRIPTION

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

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

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

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

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

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

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

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

RETURN VALUES

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

NAME

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

SYNOPSIS

```
#include <math.h>
#include <complex.h>  (for function cabs only)

double floor (double x);

float floorf (float x);

long double floorl (long double x);

double ceil (double x);

float ceilf (float x);

long double ceill (long double x);

double fmod (double x, double y);

float fmodf (float x, float y);

long double fmodl (long double x, long double y);

double fabs (double x);

float fabsf (float x);

long double fabsl (long double x);

double cabs (double complex x);
```

IMPLEMENTATION

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

STANDARDS

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

DESCRIPTION

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

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

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

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

$$sqrtcreal(x)^2+cimag(x)^2$$

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

RETURN VALUES

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

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

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

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

NAME

FLOWMARK – Allows timing of a section of code

SYNOPSIS

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

IMPLEMENTATION

Cray PVP systems

STANDARDS

CRI extension

DESCRIPTION

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

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

EXAMPLES

In Standard C, a mark is used as follows:

```
#include <stdlib.h>

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

FORTRAN EXTENSIONS

In Fortran, the calls would be:

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

SEE ALSO

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

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

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

NAME

fnmatch – Matches file name or path name

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX

DESCRIPTION

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

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

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

NOTES

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

RETURN VALUES

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

SEE ALSO

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

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

NAME

fopen, freopen, fdopen – Opens a stream

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

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

DESCRIPTION

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

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

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

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

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

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

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

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

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

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

When a file is opened for append, you cannot overwrite information already in the file. You can use `fseek`(3C) to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is disregarded. All output is written at the end of the file, and the file pointer is repositioned at the end of the output. If two separate processes open the same file for append, each process may write freely to the file without fear of destroying output being written by the other. The output from the two processes is intermixed in the file.

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

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

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

RETURN VALUES

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

FORTRAN EXTENSIONS

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

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

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

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

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

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

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

SEE ALSO

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

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

NAME

`fortran.h` – Library header for interlanguage communication functions

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

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

Type

The header file `fortran.h` defines the following type:

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

Function Declarations

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

```
_btol       _cptofcd    _fcdtocp    _fcdlen     _isfcd      _ltob
```

SEE ALSO

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

Interlanguage Programming Conventions, Cray Research publication SN–3009

NAME

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

SYNOPSIS

```
#include <fp.h>

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

IMPLEMENTATION

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

STANDARDS

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

DESCRIPTION

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

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

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

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

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

RETURN VALUES

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

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

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

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

`FP_NAN`	The argument is a NaN.
`FP_INFINITE`	The argument is either positive or negative infinity.
`FP_NORMAL`	The argument is a normal floating-point number (neither zero, subnormal, infinite, nor NaN).
`FP_SUBNORMAL`	The argument is a denormalized floating-point number.
`FP_ZERO`	The argument is positive or negative zero.

NOTES

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

SEE ALSO

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

NAME

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

IMPLEMENTATION

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

STANDARDS

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

DESCRIPTION

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

MACROS

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

Macro	Description
`HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL`	On CRAY T90 systems with IEEE arithmetic, expands to positive infinity for their corresponding data types. They expand to `double`, `float`, and `long double` expressions, respectively. `HUGE_VAL` is also defined, with the same value, in the `math.h` header file. On CRAY T3D systems, `HUGE_VAL` expands to a positive expression that is the largest representable `double` value. `HUGE_VALF` and `HUGE_VALL` are not defined.
`INFINITY`	Expands to a floating expression of type `double`, representing positive infinity. The `INFINITY` macro is not suitable for static and aggregate initialization.
`NAN`	Expands to a floating-point expression of type `double`, representing a quiet NaN. The `NAN` macro is not suitable for static and aggregate initialization.
`FP_NAN`, `FP_INFINITE`, `FP_NORMAL`, `FP_SUBNORMAL`, and `FP_ZERO`	Represent the mutually exclusive kinds of floating-point values. They expand to `int` constant expressions with distinct values. They are for number classification. See the `fpclassify`(3C) man page for individual descriptions of these macros.

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

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

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

FUNCTION DECLARATIONS

The following functions are described on their own man pages:

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

SEE ALSO

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

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

NAME

fread, fwrite – Reads or writes input or output

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

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

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

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

RETURN VALUES

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

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

FORTRAN EXTENSIONS

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

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

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

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

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

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

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

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

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

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

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

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

SEE ALSO

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

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

NAME

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

SYNOPSIS

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

IMPLEMENTATION

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

STANDARDS

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

DESCRIPTION

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

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

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

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

RETURN VALUES

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

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

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

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

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

NAME

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

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

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

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

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

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

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

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

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

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

NOTES

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

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

RETURN VALUES

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

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

FORTRAN EXTENSIONS

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

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

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

SEE ALSO

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

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

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

NAME

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

SYNOPSIS

```
#include <unistd.h>
int truncate (const char *path, off_t length);
int ftruncate (int fd, off_t length);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

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

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

RETURN VALUES

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

SEE ALSO

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

NAME

ftw, nftw – Walks a file tree

SYNOPSIS

```
#include <ftw.h>

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

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

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

DESCRIPTION

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

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

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

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

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

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

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

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

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

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

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

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

NOTES

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

Functions ftw/nftw use malloc to allocate dynamic storage during its operation. If they are forcibly terminated (for example, if longjmp is executed by *fn* or an interrupt function) they do not have a chance to free that storage, so it remains allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have *fn* return a nonzero value at its next invocation.

SEE ALSO

malloc(3C)

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

NAME

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

SYNOPSIS

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

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

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

DESCRIPTION

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

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

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

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

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

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

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

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

NOTES

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

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

WARNINGS

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

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

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

RETURN VALUES

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

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

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

SEE ALSO

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

NAME

getconfval, getconfvals, freeconfval – Gets configuration values

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

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

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

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

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

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

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

EXAMPLES

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

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

Example 1:

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

The call in example 1 would result in:

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

Example 2:

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

The call in example 2 would result in:

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

Example 3:

```
#include <stdlib.h>

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

The call in example 3 would result in:

```
        debug    = 0
```

Example 4:

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

The call in example 4 would result in:

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

Example 5:

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

The call in example 5 would result in:

```
val      = NULL
```

Example 6:

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

The call in example 6 would result in:

```
vals     = NULL
```

CAUTIONS

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

NOTES

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

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

RETURN VALUES

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

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

FILES

`/etc/config/confval`	Contains configuration information

SEE ALSO

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

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

NAME

getcwd – Gets path name of current directory

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX

DESCRIPTION

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

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

RETURN VALUES

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

EXAMPLES

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

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

char *cwd;

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

SEE ALSO

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

NAME

getdomainname, setdomainname – Gets or sets name of current domain

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

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

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

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

NOTES

Domain names are limited to 64 characters.

RETURN VALUES

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

MESSAGES

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

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

SEE ALSO

errno.h(3C)

NAME

getdtablesize – Gets file descriptor table size

SYNOPSIS

```
#include <unistd.h>
int getdtablesize (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

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

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

```
#include <unistd.h>

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

SEE ALSO

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

NAME

getenv – Returns the value for the specified environment name

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

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

RETURN VALUES

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

FORTRAN EXTENSIONS

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

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

or

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

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

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

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

or

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

A status is not returned for the GETENV subroutine call.

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

NOTES

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

SEE ALSO

putenv(3C), setenv(3C)

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

NAME

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

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

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

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

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

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

The `setfsent` function opens and rewinds the file.

The `endfsent` function closes the file.

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

NOTES

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

FILES

`/etc/fstab`

RETURN VALUES

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

SEE ALSO

`getmntent`(3C)

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

NAME

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

SYNOPSIS

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

struct group *getgrent (void);

struct group *getgrgid (int gid);

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

struct group *getgrnam (const char *name);

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

void setgrent (void);

void endgrent (void);

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

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

DESCRIPTION

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

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

When first called, getgrent returns a pointer to the first group structure of the first line in the file; thereafter, it returns a pointer to the group structure of the next line in the file. Therefore, successive calls

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

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

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

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

NOTES

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

WARNINGS

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

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

RETURN VALUES

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

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

FILES

`/etc/group`

SEE ALSO

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

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

NAME

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

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

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

DESCRIPTION

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

The following structure describes a network host entry:

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

The members of this structure are as follows:

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

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

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

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

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

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

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

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

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

NOTES

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

RETURN VALUES

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

FILES

`/etc/hosts`

`/etc/hosts.bin`

`/etc/hosts.usenamed`

`/etc/hosts.aliases`

`/usr/include/netdb.h`

`/usr/include/netinet/in.h`

SEE ALSO

`gethostinfo`(3C), `resolver`(3C)

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

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

NAME

gethostinfo – Gets network host and service entry

SYNOPSIS

```
#include <netdb.h>

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

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

The following structure describes a network host and service entry:

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

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

The members of these structures are as follows:

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

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

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

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

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

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

NOTES

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

RETURN VALUES

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

FILES

/etc/hosts

/etc/hosts.bin

/etc/host.usenamed

/etc/services

/usr/include/sys/socket.h

/usr/include/netinet/in.h

/usr/include/netiso/iso.h

/usr/include/netdb.h

SEE ALSO

gethost(3C), resolver(3C)

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

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

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

NAME

getlogin, getlogin_r – Gets login name

SYNOPSIS

```
#include <unistd.h>
char *getlogin (void);
int getlogin_r (char *bufname, size_t bufsize);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX (getlogin)
PThreads (getlogin_r)

DESCRIPTION

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

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

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

NOTES

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

RETURN VALUES

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

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

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

FILES

`/etc/utmp`	File of user information

SEE ALSO

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

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

NAME

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

SYNOPSIS

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

FILE *setmntent (char *fname);

struct mntent *getmntent (FILE *filep);

int endmntent (FILE *filep);

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

struct kmntinfo *getmntinfo (char *fname);

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

void freemntlist (struct tabmntent *tabmnt);

void freemntent (struct mntent *mnt);

struct mntent *dupmntent (struct mntent *mnt);

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

IMPLEMENTATION

Cray PVP systems

STANDARDS

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

DESCRIPTION

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

NOTES

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

RETURN VALUES

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

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

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

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

The `getmntent` function returns the following:

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

The `listmntent` function returns the following:

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

The `getmntinfo` function returns the following:

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

The `findmntentry` function returns the following:

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

The `dupmntent` function returns the following:

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

FILES

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

EXAMPLES

The following program prints the list of mounted file systems:

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

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

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

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

   endmntent(fd);
}
```

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

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

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

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

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

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

   freemntlist(tabmnt);
}
```

The following program finds the first NFS mounted file system:

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

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

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

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

   strcpy(template.mnt_type, MNTTYPE_NFS);

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

   freemntlist(tabmnt);
}
```

SEE ALSO

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

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

NAME

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

SYNOPSIS

```
#include <netdb.h>
int endnetent (void);
struct netent *getnetbyaddr (int net, int type);
struct netent *getnetbyname (char *name);
struct netent *getnetent (void);
int setnetent (int stayopen);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

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

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

The members of this structure are as follows:

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

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

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

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

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

NOTES

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

RETURN VALUES

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

FILES

`/etc/networks`

`/usr/include/netdb.h`

SEE ALSO

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

NAME

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

SYNOPSIS

```
#include <unistd.h>

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

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX

DESCRIPTION

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

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

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

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

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

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

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

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

WARNINGS

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

RETURN VALUES

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

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

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

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

EXAMPLES

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

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

main (int argc, char * argv[])
{
        int c;
        .
        .
        .
        while ((c = getopt (argc, argv, "abf:o:")) != -1){
                switch (c) {
                case 'a':
                        if (bflg)
                                errflg++;
                        else
```

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

FORTRAN EXTENSION

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

The following example shows the call to GETOPTC:

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

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

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

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

The GETVARGC call has the following format:

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

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

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

GETOARGC has the following format:

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

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

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

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

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

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

The GETVARG call has the following format:

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

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

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

The GETOARG call has the following format:

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

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

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

Fortran Examples

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

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

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

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

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

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

          ELSEIF (OPT .EQ. 'b'R) THEN
          PRINT '(a)', ' option -b- present '
```

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

SEE ALSO

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

NAME

`getoptlst` – Gets option argument list

SYNOPSIS

```
#include <stdlib.h>
int getoptlst (char *optarg, char ***optargv);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

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

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

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

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

```
,  element2
```

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

RETURN VALUES

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

EXAMPLES

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

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

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

   while ((c = getopt(argc, argv, "abf:l:o:")) != EOF) {
      switch (c) {
      case 'a':
         if (bflg)
            errflg++;
         else
            aflg++;
         break;
      case 'b':
         if (aflg)
            errflg++;
         else
            bproc( );
         break;
      case 'f':
```

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

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

SEE ALSO

getopt(3C), malloc(3C)

NAME

getpass – Reads a password

SYNOPSIS

```
#include <unistd.h>
char *getpass (const char *prompt);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

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

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

NOTES

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

FILES

/dev/tty

SEE ALSO

getpwent(3C), libudb(3C)

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

NAME

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

SYNOPSIS

```
#include <netdb.h>
int endprotoent (void);
struct protoent *getprotobyname (char *name);
struct protoent *getprotobynumber (int proto);
struct protoent *getprotoent (void);
int setprotoent (int stayopen);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

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

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

The members of this structure are as follows:

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

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

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

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

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

NOTES

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

RETURN VALUES

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

FILES

`/etc/protocols`

`/usr/include/netdb.h`

SEE ALSO

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

NAME

getpw – Gets name from UID

SYNOPSIS

```
#include <stdlib.h>
int getpw (int uid, char *buf);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

AT&T extension

DESCRIPTION

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

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

WARNINGS

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

FILES

/etc/passwd

RETURN VALUES

The getpw function returns nonzero on error.

SEE ALSO

getpwent(3C), libudb(3C)

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

NAME

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

SYNOPSIS

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

struct passwd *getpwent (void);

struct passwd *getpwuid (int uid);

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

struct passwd *getpwnam (const char *name);

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

void setpwent (void);

void endpwent (void);

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

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

DESCRIPTION

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

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

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

The pw_comment and pw_gecos fields point to the same string.

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

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

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

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

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

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

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

NOTES

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

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

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

WARNINGS

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

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

RETURN VALUES

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

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

FILES

`/etc/passwd`

`/etc/udb`

`/etc/udb.public`

SEE ALSO

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

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

NAME

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

SYNOPSIS

```
#include <rpc/netdb.h>
struct rpcent *getrpcent(void)
struct rpcent *getrpcbyname (char *name);
struct rpcent *getrpcbynumber (int number);
int setrpcent (int stayopen);
int endrpcent (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

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

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

A breakdown of this structure is as follows:

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

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

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

The `endrpcent` command closes the file.

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

NOTES

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

RETURN VALUES

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

FILES

`netdb.h`

`/etc/rpc`

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

SEE ALSO

`rpc`(3C)

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

NAME

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

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

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

DESCRIPTION

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

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

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

CAUTIONS

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

RETURN VALUES

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

FORTRAN EXTENSIONS

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

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

or

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

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

SEE ALSO

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

NAME

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

SYNOPSIS

```
#include <netdb.h>
int endservent (void);
struct servent *getservbyname (char *name, char *proto);
struct servent *getservbyport (int port, char *proto);
struct servent *getservent (void);
int setservent (int stayopen);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

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

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

The members of this structure are as follows:

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

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

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

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

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

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

NOTES

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

RETURN VALUES

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

FILES

/etc/services

/usr/include/netdb.h

SEE ALSO

gethostinfo(3C), getprot(3C)

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

NAME

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

SYNOPSIS

```
#include <netdb.h>
int endtosent (void);
struct tosent *gettosbyname (char *name, char *proto);
struct tosent *gettosent (void);
int parsetos (char *name, char *proto);
int settosent (int stayopen);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

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

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

The members of this structure are as follows:

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

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

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

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

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

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

NOTES

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

RETURN VALUES

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

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

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

FILES

`/etc/iptos`

`/usr/include/netdb.h`

SEE ALSO

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

NAME

getusershell, setusershell, endusershell – Gets user shells

SYNOPSIS

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

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

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

NOTES

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

RETURN VALUES

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

FILES

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

NAME

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

SYNOPSIS

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

struct utmp *getutent (void);

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

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

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

void setutent (void);

void endutent (void);

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

AT&T extension

DESCRIPTION

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

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

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

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

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

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

The `endutent` function closes the currently open file.

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

NOTES

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

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

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

RETURN VALUES

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

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

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

FILES

`/etc/utmp`	File of user information

`/etc/wtmp` File of user information

SEE ALSO

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

NAME

getwd – Gets current directory path name

SYNOPSIS

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

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

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

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

CAUTIONS

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

RETURN VALUES

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

SEE ALSO

getcwd(3C)

NAME

glob, globfree – Generates path names matching a pattern

SYNOPSIS

```
#include <glob.h>

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

void globfree (glob_t *pglob);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX

DESCRIPTION

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

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

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

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

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

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

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

Value	Description
`GLOB_APPEND`	Appends path names generated to those from a previous call (or calls) to `glob`. The value of `gl_pathc` will be the total matches found by this call and the previous call(s). The path names are appended to, not merged with the path names returned by the previous call(s). Between calls, the caller must not change the setting of the `GLOB_DOOFFS` flag, nor change the value of `gl_offs` when `GLOB_DOOFFS` is set, nor (obviously) call `globfree` for `pglob`.
`GLOB_DOOFFS`	Causes the `gl_offs` field to specify how many null pointers should be prepended to the beginning of the `gl_pathv` field. That is, `gl_pathv` will point to `gl_offs` null pointers, followed by `gl_pathc` path name pointers, followed by a null pointer.
`GLOB_ERR`	Causes `glob` to return when it encounters a directory that it cannot open or read. Ordinarily, `glob` continues to find matches.
`GLOB_MARK`	Appends a slash to each path name that is a directory matching `pattern`.
`GLOB_NOCHECK`	Causes the following result if *pattern* does not match any path name: `glob` returns a list consisting of only *pattern*, with the total number of path names set to 1, and the number of matched path names set to 0. If `GLOB_QUOTE` is set, its effect is present in the pattern returned.
`GLOB_NOMAGIC`	Has the same effect as `GLOB_NOCHECK` but appends `pattern` only if it contains none of the special characters `*`, `?`, or `[`. `GLOB_NOMAGIC` is needed only to simplify implementation of the historic behavior of `glob` under `csh`(1).
`GLOB_NOSORT`	Disables sorting of path names in ascending ASCII order; this increases the performance of `glob`.
`GLOB_QUOTE`	Enables the backslash (\) character for quoting. Every occurrence of a backslash followed by a character in the pattern is replaced by that character, preventing any special interpretation of the character.

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

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

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

RETURN VALUES

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

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

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

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

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

NOTES

Patterns longer than MAXPATHLEN may cause unchecked errors.

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

EXAMPLES

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

```
GLOB_t g;

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

SEE ALSO

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

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

NAME

herror – Produces host lookup error messages

SYNOPSIS

```
#include <netdb.h>
void herror (char *s);
int h_nerr;
char *h_errlist[];
int h_errno;
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The herror function writes a short error message to stderr, describing the last error encountered during a host lookup. The argument string *s* is printed first (if it is not null). Next, a colon and a blank are printed, followed by the message and a new line. To be of most use, the argument string should include the name of the program (and possibly the name of the subfunction) that encountered the error. The error number is taken from the external variable h_errno, which is set when host lookup errors occur, but not cleared when successful calls are made.

To simplify variant formatting of messages, the vector of message strings, known as the h_errlist table, is provided. h_errno can be used as an index in this table to get the message string without the new line. The number of messages provided for in the table is stored in h_nerr. This field should be checked to ensure that an error code in h_errno has a corresponding message string in the table.

NOTES

The function herror and the objects h_nerr, h_errlist, and h_errno will not work with code that is multitasked.

SEE ALSO

gethost(3C), resolver(3C), stdio.h(3C)

NAME

hsearch, hcreate, hdestroy – Manages hash search tables

SYNOPSIS

```
#include <search.h>
ENTRY *hsearch (ENTRY item, ACTION action);
int hcreate (size_t nel);
void hdestroy (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The hash-table search function hsearch is generalized from Knuth (6.4) Algorithm D. It returns a pointer into a hash table indicating the location at which an entry can be found. *item* is a structure of type ENTRY (defined in header file <search.h>) containing two pointers: *item.key* points to the comparison key, and *item.data* points to any other data to be associated with that key. (Pointers to types other than void should be cast to pointer-to-void.) *action* is a member of an enumeration type ACTION indicating the disposition of the entry if it cannot be found in the table. ENTER indicates that the item should be inserted in the table at an appropriate point. FIND indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a null pointer.

The hcreate function allocates sufficient space for the table and must be called before hsearch is used. The *nel* argument is an estimate of the maximum number of entries that the table will contain. You can use the algorithm to adjust this number upward to obtain certain mathematically favorable circumstances.

The hdestroy function destroys the search table, and it can be followed by another call to hcreate.

NOTES

Only one hash search table may be active at any given time.

The hsearch function uses open addressing with a multiplicative hash function. Its source code, however, has many other options available; you may select options by compiling the hsearch source with the following symbols defined to the preprocessor:

Symbol	Description
DIV	Use the *remainder modulo table size* instead of the multiplicative algorithm as the hash function.

USCR Use a user-supplied comparison function for ascertaining table membership. The function should be named `hcompar` and should behave in a manner similar to `strcmp` (see `string`(3C)).

CHAINED Use a linked list to resolve collisions. If this option is selected, the following other options become available:

- START Places new entries at the beginning of the linked list; default is placement at the end.
- SORTUP Keeps the linked list sorted by key in ascending order.
- SORTDOWN Keeps the linked list sorted by key in descending order.

Additionally, there are preprocessor options for obtaining debugging printout (`-DDEBUG`) and for including a test driver in the calling function (`-DDRIVER`). See the source code for further details.

WARNINGS

Both `hsearch` and `hcreate` use the `malloc`(3C) function to allocate space.

RETURN VALUES

The `hsearch` function returns a null pointer if the action is `FIND` and the item could not be found, or if the action is `ENTER` and the table is full.

If it cannot allocate sufficient space for the table, `hcreate` returns `0`.

EXAMPLES

The following example reads in strings, followed by two numbers, and stores them in a hash table, discarding duplicates. It then reads in strings, finds the matching entry in the hash table, and prints it out.

```
#include <stdio.h>
#include <string.h>
#include <search.h>
struct info {                   /* This is the info stored in the table */
      int age, room;            /* other than the key. */
};
#define NUM_EMPL    5000        /* # of elements in search table */

main( )
{
     char string_space[NUM_EMPL*20];     /* space to store strings */
     struct info info_space[NUM_EMPL];   /* space to store employee info */
     char *str_ptr = string_space;       /* next avail space in string_space */
     struct info *info_ptr = info_space; /* next avail space in info_space */
     ENTRY item, *found_item;
     char name_to_find[30];              /* name to look for in table */
     int i = 0;
```

```
      (void) hcreate(NUM_EMPL);           /* create table */
      while (scanf("%s%d%d", str_ptr, &info_ptr->age,
             &info_ptr->room) != EOF && i++ < NUM_EMPL) {
             item.key = str_ptr;

            /* put info in structure, and structure in item */
             item.data = (void *)info_ptr;
             str_ptr += strlen(str_ptr) + 1;
             info_ptr++;
             (void) hsearch(item, ENTER);
             /* put item into table */
      }
      item.key = name_to_find;            /* access table */
      while (scanf("%s", item.key) != EOF) {
          /* if item is in the table */
          if ((found_item = hsearch(item, FIND)) != NULL) {
              (void)printf("found %s, age = %d, room = %d\n",
                found_item->key,
                ((struct info *)found_item->data)->age,
                ((struct info *)found_item->data)->room);
          } else {
                 (void)printf("no such employee %s\n", name_to_find);
          }
      }
}
```

SEE ALSO

bsearch(3C), lsearch(3C), malloc(3C), string(3C), tsearch(3C)

NAME

`ia_failure` – Processes identification and authentication (I&A) failures

SYNOPSIS

```
#include <ia.h>
#include <udb.h>

int ia_failure(
      ia_failure_t *paramsent,
      ia_success_ret_t *paramret);
```

IMPLEMENTATION

All Cray Research systems except Cray MPP systems running UNICOS MAX

DESCRIPTION

The `ia_failure` routine provides the following functionality:

- Manages the updating of the authentication failure information in the user database (UDB).
- Performs I/A failure auditing.
- Processes delayed logging; this is not done for batch jobs.

paramsent contains a pointer to the structure that contains the input parameters. *paramret* contains a pointer to the structure that contains the output parameters.

RETURN VALUES

If successful, `IA_NORMAL` is returned. Otherwise, an IA exception code is returned. This routine does not return if the exit code supplied in *paramsent* is nonzero.

NOTES

This routine supports two user exits, `ia_uex_failure` (which is called on entry to this routine) and `ia_uex_failaudit` (which is called after normal auditing is performed).

SEE ALSO

`getconfval`(3C), `ia_success`(3C), `ia_user`(3C)

`slgentry`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`exit`(3C), `time`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`confval`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014, for descriptions of `login`-related UNICOS centralized user Identification/Authentication (I/A) options

`udb`(5) for a description of the UNICOS user database file

NAME

ia_mlsuser – Determines the user's mandatory access control (MAC) attributes

SYNOPSIS

```
#include <sys/mac.h>
#include <sys/udb.h>

int ia_mlsuser(
       struct udb *ueptr,
        struct secstat *sbptr,
        struct usrv *usptr,
        mls_t *rlabptr;
        int prntactive);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The ia_mlsuser routine determines the security attributes of the session based on the attributes of the user and the connection. The attributes of the session are not set.

ueptr is a pointer to the user database (UDB) entry of the user. *sbptr* is the pointer to the attributes of the connection. *usptr* is the structure in which the attributes of the session are returned to the caller. *rlabptr* is a required active label of the session. If int prntactive is nonzero, the active label of the session is echoed.

The label range for the session is the intersection of the label range of the user and the label range of the connection. If specified, the required active label must be within the range of the session. If the required active label is null, the active label of the session is set to the default label of the user. The active label is set to the minimum label of the session if the default is not within the range of the session.

NOTES

No auditing is performed by this routine; the caller must perform auditing.

The label on the current process is not changed.

RETURN VALUES

IA_NORMAL is returned for successful completion. Otherwise, IA_MAC is returned.

SEE ALSO

`getconfval`(3C), `mls_create`(3C), `mls_free`(3C), `mls_glb`(3C), `mls_lub`(3C),

`setusrv`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`confval`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014, for descriptions of centralized user identification and authentication options

NAME

ia_success – Processes identification and authentication (I&A) successes

SYNOPSIS

```
#include <ia.h>
#include <udb.h>

int ia_success(
          ia_success_t *paramsent,
          ia_success_ret_t *paramret);
```

IMPLEMENTATION

All Cray Research systems except Cray MPP systems running UNICOS MAX

DESCRIPTION

The ia_success routine provides the following functionality:

- Manages the updating of the authentication success information in the user database (UDB).
- Performs the I/A success auditing.

paramsent contains a pointer to the structure that contains the input parameters. *paramret* contains a pointer to the structure that contains the output parameters.

RETURN VALUES

If successful, IA_NORMAL is returned. Otherwise, an IA exception code is returned.

NOTES

This routine supports two user exits, ia_uex_success (which is called on entry to this routine) and ia_uex_succaudit (which is called at the end of this routine).

SEE ALSO

ia_failure(3C), ia_user(3C)

time(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

slgentry(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

udb(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

ia_user – Performs user identification and authentication (I&A)

SYNOPSIS

```
#include <ace.h>
#include <ia.h>
#include <udb.h>

int ia_user(
         ia_user_t *param,
         ia_user_ret_t *ret);
```

IMPLEMENTATION

All Cray Research systems except Cray MPP systems running UNICOS MAX

DESCRIPTION

The ia_user routine provides a common UNICOS identification and authentication mechanism. The caller specifies what authentication to perform and the order of authentication. The *pswdlist* field in the *param* structure is the list and order of authentication to be performed.

The following types of authentication are supported by this routine:

Type	Description
IA_DIALUP	Dialup authentication; not supported in batch.
IA_SECURID	SecurID passcode identification.
IA_UDB	User database (UDB) password identification.
IA_WAL	Workstation access list (WAL) verification.

These authentications can be requested in any order and combination. However, the order and handling of IA_SECURID and IA_UDB authentication have special rules, which are as follows:

Combination	Description
IA_SECURID then IA_UDB	This combination should be the first choice for authentication. UDB authentication is performed only if the SecurID acm_both flag is set, or the user is not configured for SecurID authentication.
IA_UDB only	This is the authentication choice if SecurID is bypassed.
IA_SECURID only	This is the authentication choice if UDB authentication is bypassed. Neither SecurID or UDB authentication is performed if the user is not configured with SecurID.

IA_UDB then IA_SECURID
: UDB authentication is performed, then SecurID authentication is performed if the user has a SecurID account. In this case, the acm both flag has no meaning. Both authentications are always checked.

IA_SECURID can be specified regardless of whether SecurID is configured at your site. All SecurID authentication is bypassed if your site is not configured with SecurID. No warning is returned.

The caller identifies itself to this routine. The following list describes known callers; special handling is noted for several of the callers:

Caller	Description
IA_DGDAEMON	No special handling.
IA_FTAMD	Sets *uname* to ftp if the IA_GUEST flag is set.
IA_FTPD	Sets *uname* to ftp if the IA_GUEST flag is set.
IA_LOGIN	Supports the login back door. Only allows root to log in from the console when CONSOLE is defined.
IA_NQS	No special handling of root.
IA_REXECD	No special handling.
IA_RSHD	No special handling.
IA_SU	Supports no authentication for authorized users.

The *flags* field in the *param* structure is a bit mask. The following list describes each of the supported flags:

Flag	Description
IA_FFLAG	Indicates authentication can be skipped. This flag implies the same functionality as the login implementation.
IA_GUEST	Indicates anonymous ftp or ftam.
IA_IDENTIFICATION	Indicates that only identification should be performed. The private UDB is returned to authorized callers, while the public UDB is returned to unauthorized users. IA_PUBLIC is returned if the public UDB entry is returned.
IA_INTERACTIVE	Indicates an interactive session and that the user can be prompted for information. If this flag is not set, passwords must be supplied.
IA_PUBLICIDENT	Indicates that identification should be performed. The public UDB entry is returned. Does not require any other flag. If this flag is set, the private UDB is never returned. IA_IDENTIFICATION has no meaning and is not processed.
IA_RFLAG	Indicates that this is a remote I&A and clears IA_FFLAG.

NOTES

To be an authorized user you must have read access to the private UDB.

This routine does not perform auditing or update the UDB based on the status of the I&A request. The caller must perform auditing and ensure that the UDB is updated, which can be done by using the `ia_success`(3C) and `ia_failure`(3C) routines.

This routine supports three user exits, `ia_uex_authrep` (which is called on entry to this routine); `ia_uex_authadd` (which is called after the requested authentication has been performed); and `ia_uex_authend` (which is called at the end of this routine). `ia_uex_authadd` is not called if `IA_IDENTIFICATION` or `IA_PUBLICIDENT` are specified.

RETURN VALUES

The following return values are possible:

Value	Description
`IA_BACKDOOR`	Access allowed through the back door.
`IA_BADAUTH`	Unknown authorization type.
`IA_DIALUPERR`	An error was encountered when processing the dial-up authentication.
`IA_DISABLED`	The account is disabled; disabled flag is set in the UDB.
`IA_GETSYSV`	The `getsysv`(2) system call failed.
`IA_LOCALHOST`	Access from `localhost` not allowed.
`IA_MAXLOGS`	Access denied; maximum failures on account reached.
`IA_NOPASS`	User allowed to bypass authentication.
`IA_NORMAL`	Normal return code.
`IA_PUBLIC`	The user was identified and the public UDB entry was returned. Only returned if `IA_IDENTIFICATION` is set and `IA_PUBLICIDENT` is not set.
`IA_SECURIDERR`	An error was encountered when processing SecurID authentication.
`IA_TRUSTED`	Trusted user not allowed.
`IA_UDBERR`	An error was encountered when processing UDB authentication.
`IA_UDBEXPIRED`	Authentication successful; however the UDB password has expired. Caller must process expired passwords.
`IA_UDBPWDNULL`	The password in the UDB is null, and the user is not configured for SecurID authentication.
`IA_UDBWEEK`	The password expires within the week.
`IA_UNKNOWN`	Identification error; unknown user.

IA_UNKNOWNYP	User known in UDB, but configured for network information services (NIS) and not known to NIS.
IA_WALERR	The workstation access list (WAL) denied access.

EXAMPLES

The following examples show how to use the ia_user routine.

Example 1: This example shows how `ia_user` can be called to identify a user. This example returns the public UDB entry.

```
ia_user_ret_t  uret;    /* Parameters returned from ia_user. */
ia_user_t  usent;       /* Parameters sent to ia_user.*/
struct  udb ue;

/*
 * Set up request structure.
 */
usent.revision = 0;
usent.uname = u_name;   /* May be null for interactive*/
usent.host = NULL;
usent.ttyn = ttyn;
usent.caller = IA_LOGIN;
usent.pswdlist = NULL;
usent.ueptr = &ue;

/*
 * Initialize the return structure.
 */
uret.revision = 0;
uret.pswd = NULL;
uret.normal = 0;

/*
 * Set flag requesting public udb entry.
 */
usent.flags = IA_PUBLICIDENT;

retcode = ia_user(&usent, &uret);

if (retcode == IA_NORMAL) /* User identified, public UDB entry returned.*/
else
                          /* User identified failed, UDB entry NOT returned. */
```

Example 2: This example shows how `ia_user` can be called to identify a user. This example returns either the public or private UDB entry.

```
ia_user_ret_t  uret;    /* Parameters returned from ia_user. */
ia_user_t  usent;       /* Parameters sent to ia_user.*/
struct  udb ue;

/*
 * Set up request structure.
 */
usent.revision = 0;
usent.uname = u_name;   /* May be null for interactive*/
usent.host = NULL;
usent.ttyn = ttyn;
usent.caller = IA_LOGIN;
usent.pswdlist = NULL;
usent.ueptr = &ue;

/*
 * Initialize the return structure.
 */
uret.revision = 0;
uret.pswd = NULL;
uret.normal = 0;

/*
 * Set flag requesting identification only.
 */
usent.flags = IA_IDENTIFICATION;

retcode = ia_user(&usent, &uret);

if (retcode == IA_NORMAL) /* User identified, private UDB entry returned.*/
else if (retcode == IA_PUBLIC)
                          /* User identified, public UDB entry returned.*/
else
                          /* User identified failed, UDB entry NOT returned. */
```

Example 3: This example shows how to call `ia_user` and bypass authentication. In this example, the `ia_uex_authadd` user exit is still called. The difference between this example and example 2 is that disabled account, password expiration, and so on are all processed.

```
ia_user_ret_t  uret;    /* Parameters returned from ia_user. */
ia_user_t  usent;       /* Parameters sent to ia_user.*/
struct  udb ue;

/*
 * Set up request structure.
 */
usent.revision = 0;
usent.uname = u_name;  /* May be null for interactive */
usent.host = utmp.ut_host;
usent.ttyn = ttyn;
usent.caller = IA_SU;
usent.pswdlist = NULL;
usent.ueptr = &ue;

/*
 * Set interactive flag.  This indicates that
 * the user can be prompt for name and password.
 */
usent.flags = IA_INTERACTIVE;

/*
 * Initialize the return structure.
 */
uret.revision = 0;
uret.pswd = NULL;
uret.normal = 0;

retcode = ia_user(&usent, &uret);
```

Example 4: This example shows how `login`(1) can be used to call `ia_user`. In this example, `login` calls `ia_user` to perform SecurID, UDB, DIALUP, and WAL access verification. The verification is performed in this order because of the order of the linked list.

```
ia_user_ret_t  uret;    /* Parameters returned from ia_user. */
ia_user_t  usent;       /* Parameters sent to ia_user.*/
passwd_t pwdacm,        /* Verification elements.     */
         pwddialup,
         pwdudb,
         pwdwal;
struct  udb ue;

/*
 * Set up the verification list.  The order of the list
 * is the order verification will be performed.
 */
pwdacm.atype = IA_SECURID;
pwdacm.pwdp = NULL;
pwdacm.next = &pwdudb;

pwdudb.atype = IA_UDB;
pwdudb.pwdp = NULL;
pwdudb.next = &pwddialup;

pwddialup.atype = IA_DIALUP;
pwddialup.pwdp = NULL;
pwddialup.next = &pwdwal;

pwdwal.atype = IA_WAL;
pwdwal.pwdp = NULL;
pwdwal.next = NULL;

/*
 * Set up request structure.
 */
usent.revision = 0;
usent.uname = u_name;  /* May be null for interactive */
usent.host = utmp.ut_host;
usent.ttyn = ttyn;
usent.caller = IA_LOGIN;
usent.pswdlist = &pwdacm;
usent.ueptr = &ue;

/*
 * Set interactive flag.  This indicates that
 * the user can be prompt for name and password.
 */
usent.flags = IA_INTERACTIVE;

/*
 * Initialize the return structure.
 */
```

```
uret.revision = 0;
uret.pswd = NULL;
uret.normal = 0;

retcode = ia_user(&usent, &uret);
```

SEE ALSO

ia_failure(3C), ia_mlsuser(3C), ia_success(3C) slgentry(2)

time(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

udb(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

iconv, iconv_close, iconv_open – Code conversion function

SYNOPSIS

```
#include <iconv.h>

size_t iconv (iconv_t cd, const char **inbuf, size_t *inbytesleft, char **outbuf,
size_t *outbytesleft);

iconv_t iconv_open (const char *tocode, const char *fromcode);

int iconv_close (iconv_t cd);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The iconv function converts the sequence of characters from one codeset, in the *inbuf* array, into a sequence of corresponding characters in another codeset in the *outbuf* array. Codesets are specified in the iconv_open call that returned the conversion descriptor, *cd*. The *inbuf* argument points to a variable that points to the first character in the input buffer; *inbytesleft* indicates the number of bytes to the end of the buffer to be converted. The *outbuf* argument points to a variable that points to the first available byte in the output buffer, and *outbytesleft* indicates the number of the available bytes to the end of the buffer.

For state-dependent encodings, the conversion descriptor *cd* is placed into its initial shift state by a call for which *inbuf* is a null pointer, or for which *inbuf* points to a null pointer. When iconv is called in this way, and if *outbuf* is not a null pointer or a pointer to a null pointer, and *outbytesleft* points to a positive value, iconv places, into the output buffer, the byte sequence to change the output buffer to its initial shift state. If the output buffer is too small to hold the entire reset sequence, iconv fails and sets errno to [E2BIG]. Subsequent calls with *inbuf* as other than a null pointer or a pointer to a null pointer cause the conversion to occur from the conversion descriptor's current state.

If a sequence of input bytes forms no valid character in the specified codeset, conversion stops after the previous successfully converted character. If the input buffer ends with an incomplete character or shift sequence, conversion stops after the previous successfully converted bytes. If the output buffer is too small to hold the entire converted input, conversion stops just before the input bytes that would cause the output buffer to overflow. The variable to which *inbuf* points is updated to point to the byte following the last byte successfully used in the conversion. The value to which *inbytesleft* points is decremented to reflect the number of still-unconverted bytes in the input buffer. The variable to which *outbuf* points is updated to point to the byte following the last byte of converted output data. The value to which *outbytesleft* points is decremented to reflect the number of bytes still available in the output buffer. For state-dependent encodings, the conversion descriptor is updated to reflect the shift state in effect at the end of the last successfully converted byte sequence.

If `iconv` encounters a character in the input buffer that is legal, but for which an identical character does not exist in the target codeset, `iconv` performs an implementation-defined conversion on this character.

The `iconv_open` function returns a conversion descriptor that describes a conversion from the codeset specified by the string to which the *fromcode* argument points to the codeset specified by the string to which the *tocode* argument points. For state-dependent encodings, the conversion descriptor is in a codeset-dependent initial shift state, ready for immediate use with the `iconv` function.

Settings of *fromcode* and *tocode* and their permitted combinations depend on the implementation. A conversion descriptor remains valid in a process until that process closes it.

The `iconv_close` function deallocates the conversion descriptor *cd* and all other associated resources allocated by the `iconv_open` function. If a file descriptor is used to implement the type `iconv_t`, that file descriptor is closed.

RETURN VALUES

The `iconv` function updates the variables to which the arguments point to reflect the extent of the conversion and returns the number of nonidentical conversions performed. If the entire string in the input buffer is converted, the value to *inbytesleft* points is 0. If the input conversion is stopped due to any of the preceding conditions, the value to which *inbytesleft* points is nonzero and `errno` is set to indicate the condition. If an error occurs, `iconv` returns `(size_t)-1` and sets `errno` to indicate the error.

If successful, the `iconv_open` function returns a conversion descriptor for use on subsequent calls to `iconv`; otherwise, `iconv_open` returns `(iconv_t)-1` and sets `errno` to indicate the error.

If successful, the `iconv_close` function returns 0; otherwise, it returns –1 and sets `errno`.

MESSAGES

The `iconv` function fails if any of the following errors occur:

`[EILSEQ]` Input conversion stopped due to an input byte that does not belong to the input codeset.

`[E2BIG]` Input conversion stopped due to lack of space in the output buffer.

[EINVAL] Input conversion stopped due to an incomplete character or shift sequence at the end of the input buffer.

The `iconv` function may fail if the following occurs:

[EBADF] The *cd* argument is an open conversion descriptor that is not valid.

The `iconv_open` function may fail if any of the following errors occur:

[EMFILE] The *cd* argument is not a valid open conversion descriptor.

[ENFILE] Input conversion stopped due to an input byte that does not belong to the input codeset.

[ENOMEM] Input conversion stopped due to lack of space in the output buffer.

[EINVAL] Input conversion stopped due to an incomplete character or shift sequence at the end of the input buffer.

The `iconv_close` function may fail if the following occurs:

[EBADF] The conversion descriptor is not valid.

SEE ALSO

`locale`(3C), `locale.h`(3C), `localeconv`(3C), `setlocale`(3C)

NAME

uid2nam, gid2nam, acid2nam, nam2uid, nam2gid, nam2acid, gidnamfree, acidnamfree – Maps IDs to names

SYNOPSIS

```
#include <stdlib.h>
char *uid2nam (int uid);
char *gid2nam (int gid*C);
char *acid2nam (int acid);
int nam2uid (char *uname);
int nam2gid (char *gname);
int nam2acid (char *aname);
void gidnamfree (void);
void acidnamfree (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The uid2nam function maps a numerical user ID to a character string; nam2uid maps a character string to a numerical user ID.

The gid2nam function maps a numerical group ID to a character string; nam2gid maps a character string to a numerical group ID.

The acid2nam function maps a numerical account ID to a character string; nam2acid maps a character string to a numerical account ID.

The acidnamfree and gidnamfree functions update the mapping information from the map files.

These functions provide fast mapping of numerical IDs to names, and vice versa, in the UNICOS user-information database. (See newacct(1) and udb(5).)

The acid functions copy the corresponding map files into main memory upon the first call and use either a binary search algorithm (for ID-to-name translations) or a linear search algorithm (for name-to-ID translation); the gid functions call getgrgid and getgrnam (see getgrent(3C)).

If an application depends on the most recent data in the map files and has run for a considerable amount of time, the gidnamfree and acidnamfree functions may be used to force the translation functions to update the memory copy of the map files.

WARNINGS

The nam2uid and uid2nam routines leave the udb file open to assure reasonable performance for multiple calls. If it is important that the program in which the calls are made can be restarted, call endpwent or endudb to close the udb file after the access is complete.

The nam2acid and acid2nam routines leave the account ID file open for the same reason. If it is important that the program in which the calls are made can be restarted, call acidnamfree to close the udb file after the access is complete.

The nam2gid and gid2nam functions close the group file before returning.

RETURN VALUES

If no match is found, acid2nam, uid2nam, and gid2nam return a null pointer. The nam2uid, nam2gid, and nam2acid functions all return –1 if no match is found for the name.

FILES

/etc/acid

/etc/group

/etc/udb.public

SEE ALSO

getpwent(3C), getgrent(3C), libudb(3C)

newacct(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

udbgen(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

acctid(2) in *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

udb(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

ieee_float – Introduction to the IEEE floating-point environment

IMPLEMENTATION

Cray MPP systems
CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

ANSI/IEEE Std 754-1985
X3/TR-17:199x

DESCRIPTION

The man pages in this section describe the header files, types, macros, and functions developed to support the Cray Research implementation of the IEEE floating-point standard in the Cray C and C++ compilers. The corresponding Fortran routines are described in the *Fortran Language Reference Manual, Volume 2*, Cray Research publication SR–3903.

ASSOCIATED HEADERS

<fp.h>
<fenv.h>

ASSOCIATED TYPES

fenv.h Types

Type	Description
fenv_t	Represents the entire floating-point environment
fexcept_t	Represents the floating-point exception flags
fetrap_t	Represents the floating-point trap flags

ASSOCIATED MACROS

fp.h Macros

Macro	Description
HUGE_VAL, HUGE_VALF, HUGE_VALL	Expand to positive infinity
INFINITY	Expands to positive infinity
NAN	Expands to a quiet NaN
FP_NAN, FP_INFINITE, FP_NORMAL, FP_SUBNORMAL, FP_ZERO	Represent the mutually exclusive kinds of floating-point values
DECIMAL_DIG	Represents the digits supported by conversion to internal floating-point formats

`fpclassify`(3C)	Returns the macro (`FP_NAN`, and so on) that identifies its argument
`isfinite`	Determines if its argument value is finite
`isnan`	Determines if its argument value is a NaN
`isnormal`	Determines if its argument value is normal
`signbit`(3C)	Determines if its argument value is negative
`isgreater`(3C)	Determines if its first argument is greater than its second
`isgreaterequal`	Determines if its first argument is greater than or equal to its second
`isless`	Determines if its first argument is less than its second
`islessequal`	Determines if its first argument is less than or equal to its second
`islessgreater`	Determines if its first argument is less than or greater than its second
`isunordered`	Determines if its arguments compare unordered

`fenv.h` Macros

Macro	Description
`FE_INEXACT`	Represents the inexact exception flag
`FE_DIVBYZERO`	Represents the divide-by-zero exception flag
`FE_UNDERFLOW`	Represents the underflow exception flag
`FE_OVERFLOW`	Represents the overflow exception flag
`FE_INVALID`	Represents the invalid exception flag
`FE_EXCEPTINPUT`	Represents the exceptional input exception flag
`FE_ALL_EXCEPT`	Represents the bitwise OR of all exception macros
`FE_TRAP_INVALID`	Represents the invalid operation trap flag
`FE_TRAP_DIVBYZERO`	Represents the divide-by-zero trap flag
`FE_TRAP_OVERFLOW`	Represents the overflow trap flag
`FE_TRAP_UNDERFLOW`	Represents the underflow trap flag
`FE_TRAP_INEXACT`	Represents the inexact trap flag
`FE_ALL_TRAPS`	Represents all of the trap flags
`FE_TONEAREST`	Round toward nearest
`FE_UPWARD`	Round toward positive infinity
`FE_DOWNWARD`	Round toward negative infinity
`FE_TOWARDZERO`	Round toward zero
`FE_DFL_ENV`	Represents the default floating-point environment

ASSOCIATED FUNCTIONS

`fp.h` Functions

Function	Description
`logb`(3C), `logbf`, `logbl`	Return the signed exponent of their arguments
`scalb`(3C), `scalbf`, `scalbl`	Compute x * `FLT_RADIX`^{n} efficiently

`rint`(3C), `rintf`, `rintl`
: Round arguments to an integral value in floating-point format

`rinttol`(3C)
: Rounds a floating-point number to a long integer value

`remainder`(3C), `remainderf`, `remainderl`
: Divide their arguments and return the remainder

`copysign`(3C), `copysignf`, `copysignl`
: Assign the sign of the second argument to the value of the first argument

`nextafter`(3C), `nextafterf`, `nextafterl`
: Return the next value in the direction of the second argument

`fenv.h` Functions

Function	Description
`feclearexcept`(3C)	Clears exception flags
`fegetexceptflag`	Stores the representation of the exception flags
`feraiseexcept`	Raises exceptions
`fesetexceptflag`	Restores the representation of the exception flags
`fetestexcept`	Determines which exception flags are currently set
`fesetround`(3C)	Establishes the rounding direction
`fegetround`	Gets the current rounding direction
`fegetenv`(3C)	Stores the current floating-point environment
`feholdexcept`	Saves the environment, clears exception flags, and disables traps
`fesetenv`	Establishes the floating-point environment
`feupdateenv`	Saves the current exceptions, installs a new environment, and raises the saved exceptions
`fedisabletrap`(3C)	Disables traps
`feenabletrap`	Enables traps
`fegettrapflag`	Stores the representation of the trap flags
`fesettrapflag`	Restores the representation of the trap flags
`fetesttrap`	Determines which traps are currently enabled

SEE ALSO

Migrating to the CRAY T90 Series IEEE Floating Point, Cray Research publication SN–2194

NAME

IHPSTAT – Returns statistics about the heap

SYNOPSIS

value=IHPSTAT(*code*)

IMPLEMENTATION

UNICOS and UNICOS/mk systems

DESCRIPTION

IHPSTAT returns statistics about the heap.

When using the CF90 compiler on UNICOS or UNICOS/mk systems, all arguments must be of default kind unless documented otherwise. On UNICOS and UNICOS/mk, the default kind is KIND=8 for integer, real, complex, and logical arguments.

The following is a list of valid arguments for this routine.

value Requested information.

code Code for the type of information requested, as follows:

Code	Meaning
1	Current heap length
4	Number of allocated blocks
10	Size of the largest free block
11	Amount by which the heap can shrink
12	Amount by which the heap can grow
13	First word address of the heap; on UNICOS/mk systems, byte addresses are returned.
14	Last word address of the heap; on UNICOS/mk systems, byte addresses are returned.
22	Amount by which the shared heap can grow.

All values returned by IHPSTAT are in words.

SEE ALSO

HPALLOC(3F), HPCHECK(3F), HPCLMOVE(3F), HPDEALLC(3F), HPDUMP(3F), HPNEWLEN(3F), HPSHRINK(3F), IHPLEN(3F), IHPVALID(3F)

NAME

index, rindex – Locates characters in string

SYNOPSIS

```
#include <string.h>
char *index (const char *s, int c);
char *rindex (const char *s, int c);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The index function returns a pointer to the first occurrence of character *c* in string *s*, or null if *c* does not occur in the string.

The rindex function returns a pointer to the last occurrence of character *c* in string *s*, or null if *c* does not occur in the string.

These functions operate on null-terminated strings.

NOTES

Functions strchr and strrchr (see string(3C)) are the same as index and rindex, respectively, and they should be used in all new codes. Functions index and rindex are provided only for compatibility with other BSD codes.

SEE ALSO

string(3C)

NAME

inet_addr, inet_lnaof, inet_makeaddr, inet_netof, inet_network, inet_ntoa, inet_subnetof, inet_subnetmaskof – Manipulates Internet address

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_addr (char *cp);

int inet_lnaof (struct in_addr in);

struct in_addr inet_makeaddr (int net, int host);

int inet_netof (struct in_addr in);

unsigned long inet_network (char *cp);

char *inet_ntoa (struct in_addr in);

unsigned long inet_subnetof (struct in_addr in);

unsigned long inet_subnetmaskof (struct in_addr in);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The inet_network and inet_addr functions each interpret character strings representing numbers expressed in the Internet standard "." notation (dot notation), returning numbers suitable for use as Internet network numbers and Internet addresses, respectively.

The inet_netof and inet_lnaof functions break apart Internet host addresses, returning the network number and local network address part, respectively.

The inet_makeaddr function takes an Internet network number and the host number and constructs an Internet address from them.

The inet_ntoa function takes an Internet address and returns an ASCII string representing the address in "." notation.

The `inet_subnetof` and `inet_subnetmaskof` functions return the subnet and subnet mask, respectively, of the Internet address *in*. These functions determine the actual subnet mask by consulting the configured subnet masks of the active network interfaces on the system the first time either function is called. This information is cached, and later calls to either function consult the configured network interfaces again only when a search of the accumulated information fails to match the network portion of *in*, and only for those interfaces whose associated addresses or flags have changed (for example, to detect a newly configured interface).

All Internet addresses are returned in network byte order, except for single port addresses. All network numbers and local address parts are returned as machine-format integer values.

INTERNET ADDRESSES

Values specified using the Internet "`.`" notation take one of the following forms:

a.b.c.d
a.b.c
a.b
a

When a four-part address is specified, each part is interpreted as a byte of data and is assigned, from left to right, to the 4 bytes of an Internet address.

When a three-part address is specified, the last part is interpreted as a 16-bit quantity and is placed in the rightmost 2 bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as *net.net.host*.

When a two-part address is supplied, the last part is interpreted as a 24-bit quantity and is placed in the rightmost 3 bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as *net.host*.

When only a one-part address is specified, the value is stored directly in the network address without any byte rearrangement, in host byte order.

All numbers supplied as parts in a "`.`" notation address may be in decimal, octal, or hexadecimal format, as specified in the C language (that is, a leading 0x or 0X implies hexadecimal, and a leading 0 implies octal; otherwise, the number is interpreted as decimal).

NOTES

The problem of host byte ordering versus network byte ordering is confusing.

A simple way to specify Class C network addresses in a manner similar to that used for specifying Class B and Class A addresses is needed.

The string returned by `inet_ntoa` resides in a static memory area that must be copied if it is to be used.

For inet_subnetof and inet_subnetmaskof, checking the cached information first means that they might use or return an old, incorrect subnet mask if an interface is configured down and configured back up with the same address, but a different subnet mask, between calls to either function. In practice, this should rarely happen.

Relying on active network interfaces for subnet mask information means that inet_subnetof and inet_subnetmaskof are useless without networking facilities (for example, in single-user mode). Similarly, neither function can be of any help for networks that are not directly connected to the system (for example, for networks that are not directly connected, a return value of ~0L means "I don't know if this is a subnet," not "this is definitely not a subnet").

RETURN VALUES

The inet_addr and inet_network functions return a value of ~0L for incorrect requests.

inet_subnetof and inet_subnetmaskof return the value ~0L if the network portion of the address cannot be matched with a configured interface, and 0 for addresses whose network portions are matched with an interface that has no subnet mask. Both functions set errno to EINVAL if the system has more interfaces than they can support.

FILES

/usr/include/arpa/inet.h

/usr/include/netinet/in.h

/usr/include/sys/socket.h

/usr/include/sys/types.h

SEE ALSO

gethost(3C), getnet(3C)

hosts(5), networks(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

initgroups – Initializes group access list

SYNOPSIS

```
#include <grp.h>
int initgroups (char *name, int basegid);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The initgroups function reads through the /etc/group file and uses the setgroups(2) call to set up the group access list for the user specified by *name*. The *basegid* is automatically included in the groups list. Typically, this value is given as the group number from the user database.

NOTES

The initgroups function uses the functions based on getgrent(3C). If the invoking program uses any of these functions, the group structure is overwritten in the call to initgroups.

FILES

/etc/group

RETURN VALUES

If it was not invoked by the super user, initgroups returns –1.

SEE ALSO

getgrent(3C), getpwent(3C)

setgroups(2) in *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

groups(1B), udb(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

inter_lang – Introduction to interlanguage communications functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The interlanguage communications functions provide various means for passing information between functions written in C and functions written in Fortran, Pascal, or Cray Assembly Language (CAL).

These languages use different calling sequences and have different representation for some data types; as explained in this entry, these differences must be understood and the correct conventions must be followed to ensure correct interlanguage communication. The following subsections describe these differences and conventions.

Calling Conventions

The Cray Standard C compiler running in extended mode, and all Cray Research Fortran compilers, generate code that uses the call-by-register convention for math library functions (it is the fastest calling sequence). These functions, called VFUNCTIONS, follow the call-by-register naming convention, which requires that the name be of the form NAME% or %NAME%. The compilers automatically translate the math function names to the VFUNCTION names. For further information about VFUNCTIONS, see the *Cray Standard C Reference Manual*, Cray Research publication SR–2074.

The Cray Standard C compiler running in strict conformance mode generates code that uses the call-by-value math functions. These functions perform argument domain and range checking. The names of these functions do not get translated.

In a Fortran program, if a math library function is declared EXTERNAL or INTRINSIC, the Fortran compilers generate code that uses the call-by-address math functions. The names of these functions do not get translated.

Character Pointers/Character Descriptors

The C language does not explicitly support a character string type, but by convention, C character pointers typically point to character arrays that are terminated with a 0 byte, and several functions in the C library process such strings (for more information, see character(3C)). A C character pointer, like a Fortran character descriptor, contains a character location. Unlike a Fortran character descriptor, a C character pointer does not contain a length. A C character pointer cannot be passed to a Fortran function or subroutine that expects a character argument. The format of the C character pointer is not compatible with the format of a Fortran character descriptor.

A Fortran character variable has a length associated with it that tells the number of characters in a variable. Generally, character strings used in Fortran programs are stored in character variables, rather than in arrays of single characters. A character argument can have an actual argument that is a substring or an array element that does not begin on a word boundary, so that the address of a Fortran character argument has both a word address and a bit offset. A character argument can be declared as `CHARACTER *(*)`, which means that the length of the argument is not known at compile time and must be passed to the subprogram at execution time. A Fortran character descriptor that contains the first character location and the length of a single entity (scalar or array element) is always passed for a character argument.

The interlanguage convention for passing character strings is through the use of Fortran character descriptors. Although this is automatic from Fortran, you, as a C user, must use the functions described in header file `fortran.h` to do the necessary conversions.

C Boolean Data versus Fortran Logical

C users must use functions provided as CRI extensions to pass C Boolean values as Fortran logical values. The interlanguage convention for the representation of logical values is that of Fortran type `LOGICAL`.

Calling C Functions from Fortran Functions

The Fortran language is case-insensitive; therefore, the CRI Fortran compilers map all code into uppercase. This means that functions that have lowercase names cannot be called from Fortran programs.

Calling Fortran Functions from C Functions

The C language is case-sensitive, so you must use the exact case specified in the documentation when coding references to a function.

All of the functions documented in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165, and in the *Scientific Libraries Reference Manual*, Cray Research publication SR–2081, are callable from C programs. If the manual entry for the function does not explicitly provide the C synopsis, the following rules can be used:

- Because the function is not declared in a C header, explicitly declare the function as external and specify the type of the return value.
- When calling the function, pass the address of the arguments by using the address operator (&) for each argument, or by using a pointer to the argument for the argument. Array names are considered to be addresses; therefore, the address operator is not needed when using them.
- The value returned by the function is the value, not the address of the value.
- Specify the `fortran` keyword. The `fortran` keyword is a CRI extension to the C language and is useful when a C program calls a function following the Cray Fortran calling sequence. Specifying the `fortran` keyword causes the C compiler to verify that the arguments used in each call to the function are pass-by-address. For more information on the `fortran` keyword, see the *Cray Standard C Reference Manual*, Cray Research publication SR–2074.

Calling C Functions from CAL Functions

External references from CAL are case-sensitive, so you must use the exact name for functions as specified in the appropriate manual. Cray Research supports two standard calling methods for math library functions: call-by-register and call-by-address. Cray Research supports only the call-by-address method for the scientific library. For more information on the details of calling sequences, see the documentation for the CALL macro for the machine you are using. (See the *UNICOS Macros and Opdefs Reference Manual*, Cray Research publication SR–2403.)

You should use the CALL macros to do function linkage. Avoid direct user calls to functions that use the return-jump instruction.

Scalar functions return the result in registers S1 (and S2 if needed). Vector functions return their result in registers V1 (and V2 if needed). The contents of the vector-length register (VL) upon entry determine the number of elements computed for vector functions.

Calling CAL Functions from C Functions

The following example shows a C program that calls the CAL ALOG function:

```
#include <math.h>
#include <stdio.h>

extern double ALOG(double*);

main()
{
        double x, y;

        x = 1.2345;
        y = ALOG(&x);         /* use call-by-address to pass argument */
        printf("ALOG(%f) = %f\n", x, y);
}
```

The output from the execution of this program is as follows:

```
ALOG(1.234500) = 0.210666
```

It is also possible to call library functions from the scientific library, as shown in the following example:

```
#include <math.h>
#include <stdio.h>
                     n x ix y iy

extern double SDOT(int *, double [ ], int *, double [ ], int*);

main()
{
        double sx[ ] = {1.0,2.0,3.0,4.0,5.0,6.0};
        double sy[ ] = {1.0,2.0,3.0,4.0,5.0,6.0};
```

```
        double answer;
        int  n, incx, incy;

        n     = 6;      /* number of elements in array */
        incx  = 1;      /* increment between elements in words */
        incy  = 1;

        /* Note that arrays are already passed-by-address,
           but other arguments are passed by value */

        answer = SDOT(&n, sx, &incx, sy, &incy);
        printf("Dot product of x and y is %f\n", answer);
}
```

To execute this program (in source file `b.c`), enter the following commands. To get access to the `SDOT` function, you must link to the scientific library.

```
$ cc -lsci b.c
$ a.out
```

The output is as follows:

```
Dot product of x and y is 91.000000
```

Naming Conventions

Most of the Cray Research math library functions adhere to the following naming conventions:

`NAME`	Entry for scalar call-by-address
`%NAME`	Entry for vector call-by-address
`NAME%`	Entry for scalar call-by-register
`%NAME%`	Entry for vector call-by-register

CAL does not support generic function names or automatic data type conversion. For example, no math library `LOG` function exists for the logarithm function. The user must specify either `ALOG`, DLOG, or `CLOG` for real, double-precision, or complex logarithm, respectively, and the argument must be of the correct type (real, double precision, or complex).

Associated Headers

<`fortran.h`>

Associated Functions

Function	Description
`_btol`	Converts a 0 to a Fortran logical `.FALSE.` and a nonzero value to a Fortran logical .TRUE. (see `_ltob`)
`_cptofcd`	Converts a C character pointer to a Fortran character descriptor
`_fcdtocp`	Converts a Fortran character descriptor to a C character pointer (see `_cptofcd`)
`_fcdlen`	Extracts the byte length from the Fortran character descriptor (see `_cptofcd`)

`_ltob`	Converts a Fortran logical `.FALSE.` to a `0` and a Fortran logical .TRUE. to a `1` (see `_cptofcd`)
`_isfcd`	Determines whether a generic pointer is a Fortran character descriptor

SEE ALSO

Cray Standard C Reference Manual, Cray Research publication SR–2074

Application Programmer's Library Reference Manual, Cray Research publication SR–2165

UNICOS Macros and Opdefs Reference Manual, Cray Research publication SR–2403

Interlanguage Programming Conventions, Cray Research publication SN–3009

CF77 Commands and Directives, Cray Research publication SR–3771

NAME

`i_o` – Introduction to input/output functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The input/output functions provide various means for getting data into an executing program and for sending data out of an executing program. Some functions perform input or output on streams, and some perform input or output on files. The following subsections describe these two fundamental concepts.

Streams

Input and output, whether to or from physical devices such as terminals and tape drives, or to or from files supported on structured storage devices, are mapped into logical data *streams*, whose properties are more uniform than their various inputs and outputs. UNICOS supports two forms of mapping, for *text streams* and for *binary streams*. Under the Cray Research operating system UNICOS, text streams and binary streams are implemented identically. This may not be true for other implementations.

A *text stream* is an ordered sequence of characters composed into *lines*, each line consisting of 0 or more characters plus a terminating newline character. Characters may have to be added, altered, or deleted on input and output by library functions to conform to differing conventions for representing text in the host environment. Thus, a one-to-one correspondence may not exist between the characters in a stream and those in the external representation. Data read in from a text stream compares equal to the data that was earlier written out to that stream only if the following conditions are met:

- The data consists only of printable characters, and the control characters consist only of horizontal tabs and newline characters.
- No newline character is immediately preceded by space characters.
- The last character is a newline character.

An implementation defines whether space characters that are written out immediately before a newline character appear when the data is read in. On all Cray Research systems, space characters that are written out immediately before a newline character do appear when read.

A *binary stream* is an ordered sequence of characters that can transparently record internal data. Data read in from a binary stream compares equal to the data that was written earlier out to that stream, under the same implementation. However, such a stream may have an implementation-defined number of null characters appended to the end of the stream on some systems. Under the Cray Research operating system UNICOS, no null characters are appended.

Files

A stream is associated with an external file (or physical device) by *opening* a file or *creating* a new file if no file exists. Creating an existing file causes its former contents to be discarded if necessary. If a file can support positioning requests, a *file position indicator* associated with the stream is positioned at the start (character number zero) of the file; for example, a disk file supports positioning requests, but a terminal does not. If, however, a file that supports positioning is opened with append mode, it is implementation-defined whether the file position indicator is initially positioned at the beginning or the end of the file. On Cray Research systems, the file position indicator is maintained by subsequent reads, writes, and positioning requests ensuring an orderly progression through the file.

Usage

Binary files are not truncated, except as defined in function `fopen`. Whether a write on a text stream causes the associated file to be truncated beyond that point is implementation-defined. On Cray Research systems, the associated file is not truncated.

When a stream is *unbuffered*, characters are intended to appear from the source or at the destination as soon as possible; otherwise, characters may be accumulated and transmitted to or from the host environment as a block. When a stream is *fully buffered*, characters are intended to be transmitted to or from the host environment as a block when a buffer is filled. When a stream is *line-buffered*, characters are intended to be transmitted to or from the host environment as a block when a newline character is encountered. Furthermore, characters are intended to be transmitted as a block to the host environment when a buffer is filled, when input is requested on an unbuffered stream, or when input is requested on a line-buffered stream that requires the transmission of characters from the host environment. Support for these characteristics is implementation-defined and can be affected by use of the `setbuf` and `setvbuf` functions.

To disassociate a file from a controlling stream, *close* the file. Output streams are flushed (any unwritten buffer contents are transmitted to the host environment) before the stream is disassociated from the file. The value of a pointer to a `FILE` object is indeterminate after the associated file is closed (including the standard text streams). Whether a file of 0 length (on which no characters have been written by an output stream) actually exists is implementation-defined. On Cray Research systems, the file exists.

The file may be subsequently reopened, by the same or another program execution, and its contents reclaimed or modified (if it can be repositioned at its start). If the `main` function returns to its original caller, or if the `exit` function is called, all open files are closed and all output streams are flushed before program termination. Other paths to program termination, such as calling the `abort` function, need not close all files properly.

The address of the `FILE` object used to control a stream may be significant; a copy of a `FILE` object may not necessarily serve in place of the original.

At program startup, three text streams are predefined and need not be opened explicitly: *standard input* (for reading conventional input), *standard output* (for writing conventional output), and *standard error* (for writing diagnostic output). When opened, the standard error stream is not fully buffered; the standard input and standard output streams are fully buffered if and only if the stream can be determined not to refer to an interactive device.

Functions that open additional (nontemporary) files require a *file name*, which is a string. The rules for composing valid file names are implementation-defined. Whether the same file can be simultaneously open multiple times is also implementation-defined. For Cray Research systems, file names can consist of letters, numbers, periods, and the underscore symbol and the same file can be open multiple times simultaneously.

Associated Headers

<ffio.h>	File for flexible file I/O (FFIO) functions
<stdio.h>	File for input and output functions

Associated Functions

The I_O(3C) function performs the following associated functions:

Character I/O Functions

Function	Description
fgetc	Gets a character from a stream (see getc)
fgets	Gets a string from a stream (see gets)
fputc	Puts a character on a stream (see putc)
fputs	Puts a string on a stream (see puts)
getc	Gets a character from a stream
getchar	Gets a character from a stream (see getc)
gets	Gets a string from a stream
putc	Puts a character on a stream
putchar	Puts a character on a stream (see putc)
puts	Puts a string on a stream
ungetc	Pushes a character back into the input stream

Direct I/O Functions

Function	Description
fread	Reads input
fwrite	Writes output (see fread)
getw	Gets word from stream (see getc)
putw	Puts a word on a stream (see putc)

File Access Functions

Function	Description
dup2	Duplicates an open file descriptor
fclose	Closes a stream
fdopen	Associates stream with file descriptor (see fopen)
fflush	Flushes a stream (see fclose)
fopen	Opens a stream
freopen	Substitutes named file for stream (see fopen)
getdtablesize	Gets descriptor table size
pclose	Closes a pipe to a process (see popen)
popen	Initiates a pipe to a process

setbuf	Assigns buffering to a stream
setvbuf	Assigns buffering to a stream (see setbuf)

File Error Handling Functions

Function	Description
clearerr	Clears error and EOF indicators (see ferror)
feof	Tests EOF indicator (see ferror)
ferror	Tests error indicator

File Positioning Functions

Function	Description
fgetpos	Stores the value of the file position indicator
fseek	Repositions a file pointer in a stream
fsetpos	Sets file position indicator for stream
ftell	Repositions a file pointer in a stream (see fseek)
rewind	Repositions a file pointer in a stream (see fseek)

Flexible File I/O (FFIO) Functions

Function	Description
ffbksp	Repositions an FFIO file (see ffseek)
ffclose	Closes a file using FFIO (see ffopen)
fffcntl	Performs functions on files opened using FFIO
fflistio	Initiates a list of I/O requests using FFIO
ffopen	Opens a file using FFIO
ffopens	Opens a file using FFIO (see ffopen)
ffpos	Positions files opened using FFIO
ffread	Provides FFIO
ffreada	Provides asynchronous read using FFIO
ffseek	Repositions an FFIO file
ffsetsp	Initiates EOV processing for files opened using FFIO
ffweod	Provides FFIO (see ffread)
ffweof	Provides FFIO (see ffread)
ffwrite	Provides FFIO (see ffread)
ffwritea	Provides asynchronous write using FFIO

For more information about these routines, see the *Application Programmer's I/O Guide*, Cray Research publication SG–2168. Man pages for these routines are found in the *Application Programmer's Library Reference Manual*, Cray Research publication SR–2165.

Formatted I/O Functions

Function	Description
fprintf	Prints formatted output (see printf)
fscanf	Converts formatted input (see scanf)
printf	Prints formatted output

scanf	Converts formatted input
sprintf	Prints formatted output (see printf)
sscanf	Converts formatted input (see scanf)
vfprintf	Prints formatted output of a varargs argument list (see vprintf)
vprintf	Prints formatted output of a varargs argument list
vsprintf	Prints formatted output of a varargs argument list (see vprintf)

Operations on Files

Function	Description
fileno	Returns indication of stream status
ftruncate	Truncates a file to a specified length
mktemp	Makes a unique file name
remove	Removes files
rename	Renames a file
tempnam	Creates a name for a temporary file (see tmpnam)
tmpfile	Creates a temporary binary file
tmpnam	Creates a name for a temporary file

User Information Functions

Function	Description
ctermid	Generates file name for terminal
cuserid	Gets character login name of the user

NAME

ISELFADD, ICRITADD – Allows performance of *ivar* = *ivar*+IVALUE under the protection of a hardware semaphore

SYNOPSIS

jvar = ISELFADD(*ivar*, *ivalue*)

CALL ICRITADD(*ivar*, *ivalue*)

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

ISELFADD is a function, and ICRITADD is a routine.

The following is a list of valid arguments:

Argument	Description
ivar	Integer variable to be incremented by *ivalue*.
ivalue	Amount by which *ivar* should be incremented.

A call to ISELFADD is functionally equivalent to, but considerably faster than, the following code block:

```
CALL LOCKON(lockvar)
jvar = ivar
ivar = ivar + ivalue
CALL LOCKOFF(lockvar)
```

A call to ICRITADD is functionally equivalent to, but considerably faster than, the following code block:

```
CALL LOCKON(lockvar)
ivar = ivar + ivalue
CALL LOCKOFF(lockvar)
```

SEE ALSO

XSELFADD(3F)

NAME

ISELFMUL, ICRITMUL – Allow performance of *ivar* = *ivar**IVALUE under the protection of hardware semaphore

SYNOPSIS

jvar = ISELFMUL(*ivar*, *ivalue*)

CALL ICRITMUL(*ivar*, *ivalue*)

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

ISELFMUL is a function, and ICRITMUL is a routine.

The following is a list of valid arguments:

Argument	Description
ivar	Integer variable to be multiplied by *ivalue*.
ivalue	Amount by which *ivar* should be multiplied.

A call to ISELFMUL is functionally equivalent to, but considerably faster than, the following code block:

```
CALL LOCKON(lockvar)
jvar = ivar
ivar = ivar*ivalue
CALL LOCKOFF(lockvar)
```

A call to ICRITMUL is functionally equivalent to, but considerably faster than, the following code block:

```
CALL LOCKON(lockvar)
ivar = ivar*ivalue
CALL LOCKOFF(lockvar)
```

SEE ALSO

XSELFMUL(3F)

NAME

ISELFSCH – Allows performance of *ivar* = *ivar*+1 under the protection of a hardware semaphore

SYNOPSIS

jvar = ISELFSCH(*ivar*)

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

ISELFSCH allows performance of *ivar* = *ivar*+1 under the protection of a hardware semaphore.

The following is a valid argument for this routine:

Argument	Description
ivar	Integer variable to be incremented

A call to ISELFSCH is equivalent to, but considerably faster than, the following code block:

```
CALL LOCKON(lockvar)
jvar = ivar
ivar = ivar+1
CALL LOCKOFF(lockvar)
```

SEE ALSO

XSELFMUL(3F)

NAME

`isgreater`, `isgreaterequal`, `isless`, `islessequal`, `islessgreater`, `isunordered` – Determines the relationship between two arguments

SYNOPSIS

```
#include <fp.h>

int isgreater (floating-type x, floating-type y);
int isgreaterequal (floating-type x, floating-type y);
int isless (floating-type x, floating-type y);
int islessequal (floating-type x, floating-type y);
int islessgreater (floating-type x, floating-type y);
int isunordered (floating-type x, floating-type y);
```

IMPLEMENTATION

CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

ANSI/IEEE Std 754-1985
X3/TR-17:199x

DESCRIPTION

The relational and equality operators (<, >, >=, <=, ==, and !=) support the usual mathematical relationships between numeric values. For any ordered pair of numeric values, exactly one of the relationships (less, greater, or equal) is true. Relational operators may raise the invalid exception when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the unordered relationship is true. The macros described here are *quiet* (do not raise exceptions) versions of the relational operators that facilitate writing efficient code that accounts for NaNs without raising the invalid exception.

The *floating-type* type in the SYNOPSIS section indicates an argument of any floating type. If the argument is not a floating type, the behavior is undefined.

If any of the macro definitions are suppressed in order to access an actual function, or if a program defines an external identifier with the name of one of the macros, the behavior is undefined.

The `isgreater` macro determines whether its first argument is greater than its second argument.

The `isgreaterequal` macro determines whether its first argument is greater than or equal to its second argument.

The `isless` macro determines whether its first argument is less than its second argument.

The `islessequal` macro determines whether its first argument is less than or equal to its second argument.

The `islessgreater` macro determines whether its first argument is less than or greater than its second argument.

The `isunordered` macro determines whether its arguments are unordered (that is, at least one argument is a NaN).

The IEEE standard enumerates 26 functionally distinct comparison predicates, including combinations of the four comparison results and whether invalid is raised. The following table shows how the Cray Research implementation covers all important cases.

IEEE comparisons

Greater	Less	Equal	Unordered	Raises exception	Cray implementation
		X			`x == y`
X	X		X		`x != y`
X				X	`x > y`
X		X		X	`x >= y`
	X			X	`x < y`
	X	X		X	`x <= y`
			X		`isunordered(x,y)`
X	X			X	N/A
X	X	X		X	N/A
X			X		`! islessequal(x,y)`
X		X	X		`! isless(x,y)`
	X		X		`! isgreaterequal(x,y)`
	X	X	X		`! isgreater(x,y)`
		X	X		`! islessgreater(x,y)`
	X	X	X	X	`! (x > y)`
	X		X	X	`! (x >= y)`
X		X	X	X	`! (x < y)`
X			X	X	`! (x <= y)`
X	X	X			`! isunordered(x,y)`
		X	X	X	N/A
			X	X	N/A
	X	X			`islessequal(x,y)`
	X				`isless(x,y)`
X		X			`isgreaterequal(x,y)`
X					`isgreater(x,y)`
X	X				`islessgreater(x,y)`

RETURN VALUES

The `isgreater` macro returns a nonzero value if its first argument is greater than its second argument.

The `isgreaterequal` macro returns a nonzero value if its first argument is greater than or equal to its second argument.

The `isless` macro returns a nonzero value if its first argument is less than its second argument.

The `islessequal` macro returns a nonzero value if its first argument is less than or equal to its second argument.

The `islessgreater` macro returns a nonzero value if its first argument is less than or greater than its second argument.

The `isunordered` macro returns a nonzero value if its arguments are unordered.

SEE ALSO

Migrating to the CRAY T90 Series IEEE Floating Point, Cray Research publication SN–2194

NAME

isnan – Test for NaN

SYNOPSIS

```
#include <math.h>
int isnan(double x);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The isnan function tests whether *x* is NaN (not a number).

RETURN VALUES

On Cray Research machines supporting IEEE arithmetic, the isnan function returns a nonzero value if *x* is NaN; otherwise, a 0 is returned.

On Cray Research machines with Cray Research floating-point format, isnan always returns a 0.

NOTES

A function-like macro version of isnan is implemented on Cray MPP systems and CRAY T90 systems with IEEE-standard floating-point hardware (see the fpclassify(3C) man page for more information). This IEEE version is defined in the <fp.h> header file.

If you are using a CRAY T90 system with IEEE-standard floating-point hardware, the <fp.h> version of isnan offers the advantage of accepting float and long double arguments as well as double arguments. The Cray MPP systems version accepts only double arguments.

The <math.h> version described on this man page offers XPG4 compatibility, and it is available on all Cray Research systems.

SEE ALSO

fpclassify(3C) on Cray Research IEEE systems for a description of the <fp.h> version of isnan

NAME

iso_addr, iso_ntoa – Manipulates ISO/OSI address

SYNOPSIS

```
#include <sys/types.h>
#include <netiso/iso.h>

struct iso_addr *iso_addr (char *cp);

char *iso_ntoa (struct iso_addr *isoa);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The iso_addr function interprets a character string (*cp*) representing numbers that reference an ISO network service access point (NSAP) address, returning a structure that is a valid ISO address.

The iso_ntoa function does the reverse, taking an ISO address structure (*isoa*) and returning an ASCII string representing the NSAP address.

RETURN VALUES

The iso_addr function returns a pointer to an iso_addr structure. The iso_addr function returns a null pointer for requests that are not in an accepted format of hexadecimal characters or single-quoted ASCII characters.

The iso_ntoa function always returns a character pointer.

FILES

/usr/include/netiso/iso.h

/usr/include/sys/types.h

SEE ALSO

gethostinfo(3C)

hosts(5), networks(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

kerberos_rpc, authkerb_getucred, authkerb_seccreate, svc_kerb_reg – Library routines for remote procedure calls that use Kerberos authentication

SYNOPSIS

```
cc LDFLAGS += -lkrb -lcraylm
#include <rpc/rpc.h>
#include <sys/types.h>

int authkerb_getucred(struct svc_req *rqst, uid_t *uid, gid_t *gid, short
*grouplen, int grouplist[NGROUPS]);

AUTH *authkerb_seccreate(char *service, char *srv_inst, char *realm, unsigned int
window, char *timehost, int *status);

int svc_kerb_reg(SVCXPRT *xprt, char *name, char *inst, char *realm);
```

IMPLEMENTATION

All Cray Research systems licensed for Open Network Computing Plus (ONC+)

DESCRIPTION

Remote Procedure Call (RPC) library routines let C programs make procedure calls on other machines across the network.

RPC supports various authentication flavors, including the following:

Flavor	Description
AUTH_DES	DES encryption-based authentication
AUTH_KERB	Kerberos encryption-based authentication
AUTH_NULL	(none) No authentication
AUTH_SHORT	Shorthand form of UNICOS credentials
AUTH_UNIX	Traditional UNIX-style authentication

The authkerb_getucred, authkerb_seccreate, and svc_kerb_reg routines implement the AUTH_KERB authentication flavor. The user must have run kinit(1) or ksrvtgt(1) in all cases. This man page discusses only the AUTH_KERB style of authentication.

For more information about the AUTH_NULL, AUTH_UNIX and AUTH_DES styles of authentication, see the rpc(3C) man page. See the *Remote Procedure Call (RPC) Reference Manual*, Cray Research publication SR–2089, for a definition of the AUTH data structure.

authkerb_getucred

The server side routine, authkerb_getucred, converts an AUTH_KERB credential received in an RPC request, which is operating-system independent, into an AUTH_UNIX credential. If this routine succeeds, it returns 1; if it fails, it returns 0.

The *uid* is set to the numerical ID of the user associated with the RPC request referenced by *rqst*. *gid* is set to the numerical ID of the user's group. The numerical IDs of the other groups to which the user belongs are stored in `grouplist`(). *grouplen* is set to the number of valid group ID entries returned in `grouplist`(). All information that this routine returns is based on the Kerberos principal name contained in `rqst`. This principal name is assumed to be the login name of the user, and the IDs returned are the same as if that user had physically logged in to the system.

`authkerb_seccreate`

The client side routine, `authkerb_seccreate`, returns an authentication handle that enables the use of the Kerberos authentication system. The *service* parameter is the Kerberos principal name of the service to be used. This name is generally a constant with respect to the service being used.

The *srv_inst* is the instance of the service to be called. *realm* is the Kerberos realm name of the desired service; if it is `NULL`, the local default *realm* is used.

The *window* parameter validates client credential, with time measured in seconds. If the difference in time between the client's clock and the server's clock exceeds the time value of *window*, the server rejects the client's credentials, and the clock must be resynchronized. On a Cray Research machine the `ntpd`(8) command provides this function. A small window is more secure than a large one.

The *timehost* parameter is optional and does nothing. Client and server should run the network time protocol (NTP) to synchronize time.

The *status* parameter is also optional. If you specify status, it is used to return a Kerberos error status code if an error occurs. If status is `NULL`, no detailed error codes are returned.

If `authkerb_seccreate` fails, it returns `NULL`.

`svc_kerb_reg`

The server routine, `svc_kerb_reg`, performs registration tasks in the server that are required before `AUTH_KERB` requests are processed. *xprt* is the UDP RPC transport handle which is associated with this information. Only the UDP transport handles may be registered with the *xprt* parameter. If *xprt* is `NULL`, this registration is effective for any requests that arrive on transports that have not been specifically registered. If you use the *xprt* parameter to register transports, you must use a separate `svc_kerb_reg` call for each transport.

The *name*, *inst* and *realm* parameters describe the Kerberos principal identity that this server assumes. This identity must be the same identity that the clients use when requesting Kerberos tickets for authentication. The required *name* parameter is the principal name of the service. *inst* is the instance; most common value for *inst* is `*`, which allows the Kerberos library to determine the correct instance to use, (such as the hostname on which the service is running). *realm* is the Kerberos `realm` name to use in validating tickets. If it is `NULL`, the local default *realm* is used.

Generally, `svc_kerb_reg` should be called immediately before `svc_run`. If the routine succeeds, it returns 0; if it fails, it returns -1. Kerberos RPC servers must be run as root to access the `/etc/srvtab` file to decrypt authentication messages.

NOTES

You must be licensed for Open Network Computing Plus (ONC+) to use Kerberos encryption-based authentication.

You must load the following library routines and include files along with your C program:

```
cc LDFLAGS += -lkrb -lcraylm
#include <rpc/rpc.h>
#include <sys/types.h>
```

You must install Kerberos enigma for Kerberized RPC to function. These interfaces are unsafe in multithreaded applications; therefore you should call unsafe interfaces only from the main thread.

SEE ALSO

`kerberos`(3K) in the *Kerberos User's Guide*, Cray Research publication SG–2409

`rpc`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`kinit`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`kerberos`(7) available only online

`ntpd`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

killpg – Sends signal to a process group

SYNOPSIS

```
#include <signal.h>
int killpg (int pgrp, int sig);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The killpg function sends the signal *sig* to the process group *pgrp*. See signal(2) for a list of signals.

The sending process and members of the process group must have the same effective user ID, or the sender must be the super user.

The killpg function is provided as a compatibility function. It is equivalent to the kill(2) system call with arguments *sig*, and *pgrp* multiplied by –1.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned, and errno is set to indicate the error. See kill(2) for a list of error codes.

SEE ALSO

errno.h(3C)

kill(2), signal(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

`l3tol`, `ltol3` – Converts between 3-byte integers and long integers

SYNOPSIS

```
#include <stdlib.h>
void l3tol (long *lp, char *cp, int n);
void ltol3 (char *cp, long *lp, int n);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

AT&T extension

DESCRIPTION

The `l3tol` function converts a list of *n* 3-byte integers packed into a character string to which *cp* points into a list of long integers to which *lp* points.

The `ltol3` function performs the reverse conversion from long integers (*lp*) to 3-byte integers (*cp*).

These functions are useful for file-system maintenance in which the block numbers consist of 3 bytes.

CAUTIONS

Because of possible differences in byte ordering, the numerical values of the long integers are machine-dependent.

SEE ALSO

`fs`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

lgamma, gamma, signgam – Computes log gamma function

SYNOPSIS

```
#include <math.h>
double gamma (double x);
double lgamma (double x);
int signgam;
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The gamma function behaves identically to the lgamma function, which will be referred to on the remainder of this page. Use of the name lgamma is preferred to gamma, which may be withdrawn in a future release.

The lgamma function returns ln($|\Gamma(x)|$), where $\Gamma(x)$ is defined as $\int_0^\infty e^{-t}t^{x-1}dt$. The sign of $\Gamma(x)$ is returned in the external integer *signgam*. If *x* is negative, it must not have an integral value. Argument *x* may not be 0.

The following C program fragment might be used to calculate Γ:

```
if ((y = lgamma(x)) > LN_MAXDOUBLE)
      error( );
y = signgam * exp(y);
```

LN_MAXDOUBLE is the least value that causes exp to return a range error. LN_MAXDOUBLE is defined in the header file values.h(3C).

Vectorization is inhibited for loops containing calls to the lgamma function.

RETURN VALUES

For nonpositive integer arguments, HUGE_VAL is returned, and errno is set to EDOM.

If the correct value would overflow, lgamma returns HUGE_VAL and sets errno to ERANGE.

On Cray MPP systems and CRAY T90 systems with IEEE arithmetic, `lgamma(`*NaN*`)` and `gamma(`*NaN*`)` return NaN and `errno` is set to `EDOM`.

On Cray MPP systems and CRAY T90 systems with IEEE arithmetic, the value returned by the `lgamma` and `gamma` functions when a domain error occurs can be selected by setting the environment variable `CRI_IEEE_LIBM`. The second column describes what is returned when `CRI_IEEE_LIBM` is not set, or is set to a value other than 1. The third column describes what is returned whe `CRI_IEEE_LIB` is set to 1.

Error	`CRI_IEEE_LIB`=0	`CRI_IEEE_LIB`=1
`lgamma(`*x*`)`, where *x* is less than zero	`HUGE_VAL`	NaN
`gamma(`*x*`)`, where *x* is less than zero	`HUGE_VAL`	NaN

SEE ALSO

`exp`(3C), `values.h`(3C)

`values`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

addudb, deleteudb, endudb, getsysudb, gettrustedudb, getudb, getudbchain, getudbdefault, getudbnam, getudbstat, getudbtmap, getudbuid, lockudb, rewriteudb, setudb, setudbdefault, setudbpath, setudbtmap, udbisopen, unlockudb, zeroudbstat – Library of user database access functions

SYNOPSIS

```
#include <udb.h>
int addudb (struct udb *udb);
int deleteudb (char *name);
void endudb (void);
void getsysudb (void);
void gettrustedudb (void);
struct udb *getudb (void);
struct udb *getudbchain (int option);
struct udbdefault *getudbdefault (void);
struct udb *getudbnam (char *name);
struct udbstat *getudbstat (void);
struct udbtmap *getudbtmap (void);
struct udb *getudbuid (int uid);
int lockudb (void);
int rewriteudb (struct udb *udb);
int setudb (void);
int setudbdefault (struct udbdefault *def);
int setudbpath (char *path);
int setudbtmap (struct udbtmap *tmap);
int udbisopen (void);
extern int udb_errno;
void unlockudb (void);
void zeroudbstat (void);
```

The following routines are for Cray Research internal use only:

```
const Udbhdr *udb_header_access(const long magic, const Hdrfield field,
const void *value);

const char *udb_strerror(const int code);

int resetmaxuid(void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The user database (UDB) contains control information for users of the UNICOS operating system and for the fair-share scheduler's resource groups. The UDB files replace the `/etc/passwd` file as the primary source for user validation and control information.

The UDB consists of the following files:

- `/etc/udb`
- `/etc/udb.public`
- `/etc/udb_2/udb.index`
- `/etc/udb_2/udb.priva`
- `/etc/udb_2/udb.pubva`

The files in the directory `/etc/udb_2` extend the capability of the UDB beyond what was available in previous releases.

To allow users access to nonsensitive UDB information, the files `/etc/udb.public`, `/etc/udb_2/index`, and `/etc/udb_2/udb.pubva` are publicly readable. The other files contain privileged information, such as encrypted passwords and security information, and can be read only by privileged callers. Write access to all files is restricted to privileged users.

The UDB files are binary files that are carefully constructed (and carefully accessed) so that multiple reader processes can access them without being disturbed by ongoing modifications of the database. That is, privileged processes can update the database concurrently with other processes reading the database. In addition, the `libudb`(3C) functions use hash lists and a direct-access index to speed the search for locating user information by user ID (UID), resource group ID, user name, and resource group name.

Because of this capability for multiple accesses, the UDB should be accessed only through the provided library functions described in the following paragraphs. These functions provide access to the user information and system defaults contained in the UDB. Other access methods might corrupt the database and require regeneration of the files.

The `getudb`, `getudbnam`, and `getudbuid` functions each return a pointer to an object with the following UDB structure containing the broken-out fields of one entry from the UDB. If the pointer returned has the value `UDB_NULL`, a record could not be returned for the reason set in the `udb_errno` file. When `getudb` is being used for sequential reading of the database, a `UDB_NULL` pointer along with the value `UDBERR_END` in `udb_errno` signals the end of the database.

If more than one record has the same UID, then `getudbuid` returns the most recently added record. (To check for multiple identical UIDs after a call to `getudbuid`, use the `getudbchain` function with the `UDBCHAIN_PRIOR` direction option and check the UID of the returned record.)

The `getudbchain` function is used to take advantage of the UID chain maintained in the database. The chain is bidirectional, ordered by UID value. The direction of ascending value of UID is called *next*, and the direction of descending value of UID is called *prior*. Each `getudb`, `getudbnam`, or `getudbuid` call sets the current position to that of the returned record; a call to `setudb` or to `getudbchain` with *option* set to `UDBCHAIN_FIRSTNR` sets the current position so that the next record is the one with the smallest UID value. Definitions for the information shown in the structure are found in `udb.h`.

Option	**Description**
`UDBCHAIN_FIRST`	UID value 0. Returns the first record in the UID chain and sets the current position to that record.
`UDBCHAIN_FIRSTNR`	UID value 1. Sets the current chain position so that the next record is the first record in the UID chain (smallest numeric UID value). A record is not returned, so the returned pointer is `UDB_NULL` and `udb_errno` is `0`. An error has occurred if `udb_errno` is nonzero.
`UDBCHAIN_LAST`	UID value 2. Returns the last record in the UID chain and sets the current position to that record.
`UDBCHAIN_LASTNR`	UID value 3. Sets the current chain position so that the next record is the last record in the UID chain (largest numeric UID value). A record is not returned, so the returned pointer is `UDB_NULL` and `udb_errno` is `0`. An error has occurred if `udb_errno` is nonzero.
`UDBCHAIN_NEXT`	UID value 4. Returns the next record in the UID chain. The end of the chain is indicated by the returned pointer being set to `UDB_NULL` and `udb_errno` being set to `UDBERR_END`.
`UDBCHAIN_PRIOR`	UID value 5. Returns the prior record in the UID chain. The beginning of the chain is indicated by the returned pointer being set to `UDB_NULL` and `udb_errno` being set to `UDBERR_END`.

The addudb and rewriteudb functions take a struct udb object and create or update a named record in the UDB; deleteudb removes the named record. When the return value is UDB_FAIL, it means that the requested function could not be done for the reason set in udb_errno. Before any of these functions is called, a call to lockudb is required to prevent multiple processes from writing to the database at the same time, thereby possibly corrupting it. The database is automatically unlocked upon return from these functions. Each call to lockudb should be checked for the return value UDB_FAIL, which indicates that the lock could not be obtained. All of these functions require write permission to the database.

A call to setudb has the effect of rewinding the database to allow sequential reading with getudb, and it also positions the chain pointer to the first record (lowest UID value) for use by getudbchain. The endudb function can be called to close the database when processing is complete.

For security reasons, UDB information is divided into protected and public categories; the access method retrieves only the public information unless special library set-up calls are made. A caller needing specific access to the protected category of information (to read or update the information in any way) must make a call to getsysudb or gettrustedudb after the optional setudbpath call but before any other calls to the database are made. If this is not done, the public file will be opened and no updating or access to protected information will be possible unless endudb is called to close the files. To guarantee predictable behavior for the system security programs, the various get requests will fail if the protected copy of the database cannot be accessed. The getsysudb function also causes such behavior.

The difference between getsysudb and gettrustedudb is that no files remain open for writing after an unlockudb call (including the implicit calls from addudb, deleteudb, and rewriteudb) unless gettrustedudb is called. Except for updates of multiple records, use getsysudb when updating the database, because this improves data integrity.

The setudbpath function changes the path name to the database files and can be used to support multiple user databases. The path must be accessible to the caller and must end in the name of a directory. If this function is called after the database is opened and the name changes because of this call, the previously accessed database will be closed. If the pointer or the string is null, the path will be restored to the default. The entire file name is limited in length to the value UDBFNAMELEN (1024 characters). If a stat(2) call using *path_name*/udb_2 returns a directory named udb_2, the library assumes that the files udb.index, udb.priva, and udb.pubva will be found in that directory. If udb_2 is not found, those files are assumed to exist in *path_name* rather than *path_name*/udb_2.

The getudbstat function returns a pointer to a udbstat structure (shown later in this section) that contains statistical and other information related to the UDB access method. The values in the structure are not guaranteed to be accurate and static except immediately following this call and before any other UDB library calls are made. (Make a private copy of the returned structure if historical information is needed.) The zeroudbstat function resets all statistical information and may be used after setudbpath is called to restart the gathering of statistics on the new database.

The udbisopen function returns the open state of UDB files. When no files are open, 0 is returned; otherwise the return values are as follows:

State	Value
Protected open read	00001

Protected open write	00002
Public open read	00004
Public open write	00010
Index open read	00020
Index open write	00040

These values may be returned in combinations.

The `getudbdefault` function returns a pointer to the current default table (`struct udbdefault`). This table contains the defaults currently recorded in the UDB. The `setudbdefault` function replaces the default table in the UDB with a new table. The `setudbdefault` function requires that the calling process have write permission to the UDB.

The `getudbtmap` function returns a pointer to the current global tape name structure (`struct udbtmap`). Each of the eight tape name ordinals is represented in this table. If an ordinal does not have an associated global name, the name string is null. The `setudbtmap` function replaces the existing tape name map in the UDB with a new tape name map. The `setudbtmap` function requires that the calling process have write permission to the UDB.

The `udb_strerror` function returns the message string associated with a given UDB error code.

Description of `struct udb`

```
struct udb {
    char      ue_passwd[MAXUE_EPASSWD + 1];      encrypted password;
    char      ue_comment[MAXUE_COMMENT + 1];     comment
    char      ue_dir[MAXUE_HOMEDIR + 1];         default login directory
    char      ue_shell[MAXUE_SHELL + 1];         default login shell or program
    char      ue_age[MAXUE_AGE + 1];             included for compatibility; not used
    int       ue_acids[MAXVIDS];                 valid account ids
    int       ue_gids[MAXVIDS];                  valid group ids
    char      ue_root[MAXUE_LOGINROOT + 1];      login root directory
    char      ue_logline[MAXUE_LOGLINE + 1];     line used for last login
    char      ue_loghost[MAXUE_HOSTNAME + 1];    hostname for last login
    char      ue_batchhost[MAXUE_HOSTNAME + 1];  hostname of last batch req origin
    long      ue_logtime;                        time of last login (GMT in secs)
    long      ue_batchtime;                      time of last batch request
    long      ue_permbits;                       user permission bits
    long      ue_sitebits;                       site supported permission bits
    long      ue_archlim;                        disk space protected from archiving
    int       ue_archmed;                        archiving medium selector
    long      ue_jproclim[MAXUE_RCLASS];         per job max # processes
    long      ue_jcpulim[MAXUE_RCLASS];          per job cpu limit [seconds]
    long      ue_pcpulim[MAXUE_RCLASS];          per proc. cpu limit [seconds]
    long      ue_jmemlim[MAXUE_RCLASS];          per job mem. limit [512 words]
    long      ue_pmemlim[MAXUE_RCLASS];          per proc. mem. limit [512 words]
    long      ue_pfilelim[MAXUE_RCLASS];         per proc. file size limit: [512 words]
    unsigned char                                per job tape limit
              ue_jtapelim[MAXUE_RCLASS][MAXUE_TAPETYPE];
    int       ue_nice[MAXUE_RCLASS];             user's nice value (0..19)
    int       ue_logfails;                       #of consecutive login failures
    int       ue_deflvl;                         default security level
    int       ue_maxlvl;                         maximum security level
    int       ue_minlvl;                         minumum security level
    long      ue_defcomps;                       default compartments at login
    long      ue_comparts;                       valid compartments
    int       ue_permits;                        valid permissions
    int       ue_disabled;                       user login disabled flag
    int       ue_trap;                           trap on login
    char      ue_name[16];                       user's login name
    int       ue_uid;                            UID
    int       ue_resgrp;                         resource group UID
    long      ue_shflags;                        share flags
    short     ue_shares;                         allocated shares
    short     ue_shplimit;                       included for compatibility; not used
    mlimit_t  ue_shmlimit;                       included for compatibility; not used
    long      ue_jsdslim[MAXUE_RCLASS];          per job sds limit
    long      ue_psdslim[MAXUE_RCLASS];          per process sds limit
    float     ue_shusage;                        decaying accum costs
```

```
    float     ue_shcharge;                      long-term accum cost
    long      ue_shextime;                      time last lnode freed
    long      ue_limflags;                      included for compatibility; not used
    int       ue_umask;                         included for compatibility; not used
    int       ue_warnings;                      included for compatibility; not used
    int       ue_reason;                        included for compatibility; not used
    int       ue_intcls;                        default integrity class (obsolete)
    int       ue_maxcls;                        maximum integrity class (obsolete)
    long      ue_intcat;                        default integrity categories
    long      ue_valcat;                        valid integrity categories
    long      ue_lastlogtime;                   time of last login attempt
    int       ue_cpu_quota;                     CPU quota (seconds * 10)
    int       ue_cpu_quota_used;                accumulated CPU time (seconds * 10)
    long      ue_jfilelim[MAXUE_RCLASS];        per job new file space allocation limit
    struct ue_pwage {
       long   time;                             second clock when password was changed
       int    flags;                            see PWFL_xxx flags
       int    maxage;                           maximum password age in seconds
       int    minage;                           minimum password age in seconds
    } ue_pwage;
    int       ue_parentuid;                     UID of this user's administrator
    int       ue_adminmax;                      number of users this UID can administer
    int       ue_jpelimit[MAXUE_RCLASS];        per job max # MPP PEs
    int       ue_jmpptime[MAXUE_RCLASS];        per job max time MPP rsvd
    int       ue_jmppbarrier[MAXUE_RCLASS];     per job max MPP barriers
    int       ue_pmpptime[MAXUE_RCLASS];        per process max time MPP rsvd
    int       ue_pcorelim[MAXUE_RCLASS];        per proc core file limit [512 words]
    int       ue_pfdlimit[MAXUE_RCLASS];        per process open file limit
    int       ue_mincomps;                      default compartments at login
    int       ue_jshmsegs[MAXUE_RCLASS];        per job max shared memory segments
    int       ue_jshmsize[MAXUE_RCLASS];        per job max shared memory [512 words]
    int       ue_jsocbflim[MAXUE_RCLASS];       per job max socket buffer limit [512 words]
};
```

`ue_passwd` Encrypted password of no more than 15 characters.

`ue_comment` Arbitrary string of no more than 39 characters usually including the user's name, department, and other personal information.

`ue_dir` Default login directory (home directory) name of no more than 63 characters.

`ue_shell` Default login shell or program name up to 63 characters in length.

`ue_age` The `ue_age` field is obsolete, but remains present for compatibility reasons. It is a read-only field which is constructed by the interface library from the newly added `ue_pwage` structure. For more information, see the description of `ue_pwage`.

`ue_acids` List of zero or more (maximum of 64) numerical account IDs (ACIDs) for the user. When fewer than 64 ACIDs are declared, the list is terminated by an entry with the value –1; `udbgen`(8) and `udbsee`(1) present these as comma-separated values.

`ue_gids` List of zero or more (maximum of 64) numerical group IDs (GIDs) for the user. When fewer than 64 GIDs are declared, the list is terminated by an entry with the value –1; `udbgen`(8) and `udbsee`(1) present these as comma-separated values.

`ue_root` Login root directory name consisting of a maximum of 63 characters.

`ue_logline` A string of no more than 15 characters specifying the line or port from which the most recent login originated.

`ue_loghost` A string of no more than 31 characters specifying the host from which the most recent login originated.

`ue_batchhost`
A string of no more than 31 characters naming the host from which the most recently submitted batch job originated.

`ue_logtime` The time at which the most recent login occurred.

`ue_batchtime`
The time at which the most recently submitted batch job arrived.

`ue_permbits` User permission bits. This is a bit list in which the meaning of each bit is defined in `udb.h`; `udbgen`(8) and `udbsee`(1) present these as named bits separated by commas. For a list that maps user permission bits to kernel permbits, see `getpermit`(2).

Bit	Description
`PERMBITS_ACCT`	Accounting permission. Allows the user to run the `accton`(8) and `csaswitch`(8) commands. The `accton`(8) command turns process accounting on and off. The `csaswitch`(8) command checks the status of and enables or disables process, daemon, and record accounting.
`PERMBITS_ACCTID`	Allows the user to use the `diskusg`(8) command to merge intermediate disk accounting records. By using the `chacid`(1) command, the user may set the account ID of a file that is owned by another user and may specify any account ID value. By using the `quota`(1) command, the user can see all acccount IDs, group IDs, and user IDs. By using the `newacct`(1) utility, the user can change the account ID of the calling shell.

PERMBITS_ASKACID	Allows queries for active account ID. When this permbit is assigned to a user login account in the UDB, and the account also has multiple account IDs or the `acctid` (PERMBITS_ACCTID) permbit, `login` prompts the user for the account ID that should be assigned to the session.
PERMBITS_BYPASSLABEL	Allows the user to bypass label processing. This permission bit replaces the `lbypass` permit.
PERMBITS_CHOWN	Allows the user to change owner (`chown`(2)), change group (`chgrp`(1)), or change permissions (`chmod`(1)) for any file owned by that user.
PERMBITS_CHROOT	Allows the user to use the `chroot`(8) command to execute a command relative to a newroot.
PERMBITS_DEDIC	Allows the user to dedicate a CPU to a process.
PERMBITS_DEVMAINT	Allows the user to use the `ddms`(8) (disk diagnostic and maintenance system) command without being super user.
PERMBITS_GROUPADM	Allows the user to be a group administrator. The `xadmin`(8) utility supports group administration and this permission controls which users are group administrators. Other configuration of `xadmin`(8) is also required to support group administration.
PERMBITS_GUARD	Driver DONUT guard mode. When running disk diagnostics, a user with this permission can access user data space. This is denied without this privilege.
PERMBITS_GUEST	Allows use of guest operating system. Allows the user access to most of the functionality provided by the `guest`(1) command. This includes starting, stopping, changing and dumping a guest. This permission is not needed to obtain guest status.
PERMBITS_GUESTADM	Allows administration of guest operating system. The use of certain `guest`(1) command line options is controlled through this permission. The controlled options are those that may have a global system effect, including the following: `-T` Enable or disable extended kernel tracing. `-K` Enable or disable panicking the host if the guest panics. `-D` Dump all active systems. `-O` Usurp guest system from the original owner.

	`-P` Change CPU percentages of active guests.
`PERMBITS_ID`	Allows ID changes. Allows the user to set his or her real, effective, and saved-set user IDs (`setuid`(2)). Allows the user to set his or her real, effective, saved-set group IDs (`setgid`(2)), and group list (`setgroups`(2)). Allows the user to set his or her account ID (`acctid`(2)) and the account ID of a file (`chacid`(2)).
`PERMBITS_IPCPERSIST`	Allows the user to allocate persistent shared memory blocks. When a user with this permission uses the `msgget`(2), `semget`(2) or `shmget`(2) system calls, reserved memory segments are retained beyond the life of the creating session.
`PERMBITS_MKNOD`	Allows the user to use the `mknod`(2) system call with a mode other than `S_IFIFO`.
`PERMBITS_MLSMNT`	Unused. This permbit is available to assign user accounts, but it no longer grants special abilities.
`PERMBITS_MOUNT`	Allows `mount`. The `mount`(2) system call that allows a user to mount a file system, requires this permission or super user privilege and returns an `EPERM` error code if the user is not permitted. The `umount`(2) system call that allows a user to unmount a file system, requires this permission or super user privilege and returns an `EPERM` error code if the user is not permitted.
`PERMBITS_NICE`	Allows `nice` negative values. The `nice`(2) system call that allow users to change their nice value requires this privilege or super-user if the nice value is not zero (0). The system call will return an EPERM error code if the user is not permitted.
`PERMBITS_NOBATCH`	Denies users permission to run batch processes. The `setlimits()` call from the `login`(1) command for a batch job request will fail if the user has this permission and the user will see the message `"setlimits: batch sessions not permitted"`.
`PERMBITS_NOIACTIVE`	Denies users permission to run interactive processes. The `setlimits()` call from the `login`(1) command for an interactive job request will fail if the user has this permission and the user will see the message `"setlimits: interactive sessions not permitted"`.

PERMBITS_PLOCK	Allows a process to lock itself in memory (plock(2)). The plock(2) system call that allows a process to become locked into memory during execution requires this privilege or super-user. The system call will return an EPERM error code if the user is not permitted. This privilege is added to the process permits during a cpuopen() call.
PERMBITS_REALTIME	Allows the user to activate real-time processes. The cpucntl(CPU_SETRT) call that allows the process to set the real-time execution state requires this privilege or super-user. The system call will return an EPERM error code if the user is not permitted.
PERMBITS_RESLIM	Resource limits permission. The limits(2) system call for function L_SETLIM requires this privilege or super-user to connect the process to a new limits structure that is passed as an lnode structure. The system call will return an EPERM error code if the user is not permitted. The limits(2) system call for function L_DEADGROUP requires this privilege or super-user to collect the dead limits structure and returns an lnode structure. The system call will return an EPERM error code if the user is not permitted. The limits(2) system call for function L_SETIDLE requires this privilege or super-user to set the limits fields in an idle limits structure that is passed as an lnode structure. The system call will return an EPERM error code if the user is not permitted. The limits(2) system call for function L_CHNGLIM requires this privilege or super-user to change the limits fields in the limits structure that is passed as an lnode structure with the correct user ID. The system call will return an EPERM error code if the user is not permitted. The limits(2) system call for function L_UPDATEKN requires this privilege or super-user to allow the shrdaemon to update the kernel lnode fields previously calculated by the kernel. The system call will return an EPERM error code if the user is not permitted. The ulimit(2) system call for function UL_SETFSIZE requires this privilege or super-user if the file size limit is being increased. The system call will return an EPERM error code if the user is not permitted. The setpermit(2) system call requires this privilege or

super-user if the permits for a job or process are being increased. The system call will return an EPERM error code if the user is not permitted.

The `cpu`(4) system call with an `ioctl` for function `CPU_RTPERMIT` requires this privilege or super-user. The system call will return an EPERM error code if the user is not permitted.

`PERMBITS_RESTRICTED` Restricts system access (`udbrstrict`(8)). Set and cleared by the `udbrstrict`(8) command to allow or disallow creation of sessions, either batch or interactive, by the user.

`PERMBITS_SIGANY` Allows the user to send signals to any process, regardless of ownership.

`PERMBITS_SUSPRES` Allows the user to suspend and resume processes outside their own session.

`PERMBITS_SYSPARAM` Allows the user to change various system parameters. These include:

- System tick rate
- Maximum user error interrupts
- Memory scheduling parameters (`nschedv`(8))
- System memory size (`chmem`(8))
- Fair-share parameters
- CPU characteristics (`target`(1))
- Time-of-day

`PERMBITS_TAPEMANAGER` Allows the user special tape access privileges such as allowing tape formatting and mounting of tapes owned by other users.

`PERMBITS_WRUNLABEL` Allows the user to read and write unlabeled tapes. This permission bit replaces the `wrunlab` permit. For information on labeling tapes, see `tplabel`(8).

`PERMBITS_YP` Network information services (NIS) reference flag.

`ue_sitebits` There are 32 site-defined permission bits. These bits are named `site1` (octal 01) through `site32` (octal 020000000000). These bits can be set, cleared, and displayed through `udbgen`(8) and `udbsee`(1) by either their generic name or octal value.

`ue_archlim` Disk space immune to data migration.

`ue_archmed` An index to the medium selected for data migration. The meaning of this field is site-specific except for the value 0, which is reserved for the default medium; 0 is always valid in the released system.

`ue_jproclim` Job process limit. Two values exist, one for batch and the other for interactive work; `udbgen`(8) and `udbsee`(1) present the values as `jproclim[b]` for batch and `jproclim[i]` for interactive.

`ue_jcpulim` Job CPU time limit in seconds. Two values exist, one for batch and the other for interactive work; `udbgen`(8) and `udbsee`(1) present the values as `jcpulim[b]` for batch and `jcpulim[i]` for interactive.

`ue_pcpulim` Per-process CPU time limit in seconds. Two values exist, one for batch and the other for interactive work; `udbgen`(8) and `udbsee`(1) present the values as `pcpulim[b]` for batch and `pcpulim[i]` for interactive.

`ue_jmemlim` Job memory limit in units of 512 words (4096 characters). Two values exist, one for batch and the other for interactive work; `udbgen`(8) and `udbsee`(1) present the values as `jmemlim[b]` for batch and `jmemlim[i]` for interactive.

`ue_pmemlim` Per-process memory limit in units of 512 words (4096 characters). Two values exist, one for batch and the other for interactive work; `udbgen`(8) and `udbsee`(1) present the values as `pmemlim[b]` for batch and `pmemlim[i]` for interactive.

`ue_pfilelim` Per-process file allocation limit in units of 4096 characters (512 words). Two values exist, one for batch and the other for interactive work; `udbgen`(8) and `udbsee`(1) present the values as `pfilelim[b]` for batch and `pfilelim[i]` for interactive.

`ue_jtapelim` Job tape assignment limit. Sixteen values exist, 8 for batch and 8 for interactive work. Commands `udbgen`(8) and `udbsee`(1) present the values as `jtapelim[b][`*t*`]` for batch and `jtapelim[i][`*t*`]` for interactive; *t* is a tape type represented by an integer from 0 through 7.

`ue_nice` Job nice increment in the range 0 through 19; default is 0. Two values exist, one for batch and the other for interactive work; `udbgen`(8) and `udbsee`(1) present the values as `nice[b]` for batch and `nice[i]` for interactive. See `nice`(1) for more information.

`ue_logfails` The field in which the number of failed password attempts is recorded. Login is prohibited when the value of this field exceeds the limit defined at installation.

`ue_deflvl` Default security level assigned to the user at login. This is a numeric value with a default of 0.

`ue_maxlvl` Maximum security level allowed for the user; the default value is 0.

`ue_minlvl` Minimum security level allowed for the user; the default value is 0.

`ue_defcomps` Default security compartments, which are compartments assigned to the user at login. The format and meaning are included in the following `ue_comparts` description. The default is no initial security compartment assignment.

`ue_comparts` Authorized security compartments. These are compartments the user may select. The field is actually a bit list, but it is represented externally as a comma-separated list of compartments; the default is no authorized compartments. The valid compartment names and bits are defined in the `sys/secparm.h` include file, as follows:

Name	Description
`crayri`	Cray Research, Inc.
`netadm`	Network administrator
`secadm`	Security administrator
`sysadm`	System administrator
`sysops`	System operator
`unicos`	UNICOS system (obsolete)

`ue_permits` Permissions attributed to the user. The field is actually a bit list, but it is represented externally as a comma-separated list of permission names; the default is no permissions. The valid permissions are defined in the `sys/secparm.h` include file, as follows:

Permission	Description
`install`	This field is obsolete.
`lbypass`	This field is obsolete; see the `bypasslabel` permbit.
`reclsfy`	This field is obsolete.
`rmtaccs`	This field is obsolete.
`suidgid`	Allows the user to set the set-UID or set-GID bits for a file.
`usrtrap`	Sets the user in trap mode during login; all discretionary and mandatory access attempts are logged.
`wrunlab`	This field is obsolete; see the `wrunlab` permbit.

`ue_disabled` User password lock. When this field is nonzero, the user is not allowed to access the system.

`ue_trap` "Trap on login" security feature.

`ue_name` The user name, consisting of up to 8 alphanumeric characters. The first character in this field must be a letter; uppercase letters are allowed but are not recommended. This field **must** be defined and unique.

`ue_uid` Numeric user ID (UID); used internally in UNICOS utilities and the operating system. This field must be defined.

`ue_resgrp` Fair-share resource group UID.

`ue_shflags` Fair-share flags as defined in `sys/share.h`.

`ue_shares` Fair-share user's allocated shares.

`ue_shplimit` Included for compatibility; obsolete.

`ue_shmlimit` Included for compatibility; obsolete.

`ue_jsdslim` Job secondary data segment limit in units of 512 words (4096 characters). Two values exist, one for batch and the other for interactive work; `udbgen`(8) and `udbsee`(1) present the values as `jsdslim[b]` for batch and `jsdslim[i]` for interactive. The field is present but not used on Cray Research systems without an SSD.

`ue_psdslim` Process secondary data segment limit in units of 512 words (4096 characters). Two values exist, one for batch and the other for interactive work; `udbgen`(8) and `udbsee`(1) present the values as `pmemlim[b]` for batch and `pmemlim[i]` for interactive. This field is present but not used on Cray Research systems without an SSD.

`ue_shusage` Fair-share decaying accumulated costs.

`ue_shcharge` Fair-share user's long-term accumulated costs.

`ue_shextime` Time at which this user's last fair-share scheduler lnode was released. The *lnode* is a kernel structure that holds running user control information for the scheduler.

`ue_limflags` Included for compatibility; obsolete.

`ue_umask` Included for compatibility; obsolete.

`ue_warnings` Included for compatibility; obsolete.

`ue_reason` Included for compatibility; obsolete.

`ue_intcls` Default integrity class assigned to an administrator at login. This is a numeric value with a default of 0. This field is obsolete.

`ue_maxcls` Maximum integrity class allowed for an administrator. This is a numeric value with a default of 0. This field is obsolete.

`ue_intcat` Default category assigned to an administrator at login. The format and meaning are described in `ue_valcat`. The default is no initial integrity category assignment.

`ue_valcat` Authorized categories assigned to an administrator. These are categories that an administrator may enable. This field is a word, in which each category is represented by a single bit. A name is associated with each category. Externally, authorized categories are displayed as a comma-separated list of category names. Valid category names and bits are defined in the `sys/tfm.h` include file. Categories that may be assigned to an administrator are as follows:

Category	**Description**
`secadm`	Security administrator
`sysadm`	System administrator
`sysops`	System operator
`unicos`	UNICOS system (obsolete)
`sysfil`	System file (obsolete)

ue_lastlogtime
: The time at which a user's most recent login attempt occurred. This time is updated whether or not the login attempt was successful.

ue_cpu_quota
: CPU quota in seconds * 10. If this field is nonzero, the user will not be permitted to accumulate (in ue_cpu_quota_used) more than the value in this field.

ue_cpu_quota_used
: Accumulated CPU time in seconds * 10. If ue_cpu_quota is nonzero and if the value in this field exceeds ue_cpu_quota, the user will not be permitted to run.

ue_jfilelim
: Per-job file allocation limit in units of 4096 characters (512 words). Two values exist, one for batch and the other for interactive work; udbgen(8) and udbsee(1) present the values as jfilelim[b] for batch and jfilelim[i] for interactive.

ue_pwage
: This is a structure containing a number of fields specifying the password age control for the record. Password age makes it possible for the administrator to control how long passwords remain valid and how they can be changed. Each of the fields in the structure will be described separately.

Field	Description
ue_pwage.time	A copy of the time of day clock when the password was last changed.
ue_pwage.flags	Control flags. If the value is PWFL_FORCE, the user must change the password at the next login. If the value is PWFL_SUPERUSER, only the administrator is allowed the change the password.
ue_pwage.maxage	The maximum age in seconds the password may have. If the difference between ue_pwage.time and the current time exceeds this value, the user will be required to change the password.
ue_pwage.minage	The minimum amount of time that must elapse before a password may be changed.

ue_parentuid
: UID of the group administrator (used by xadmin(8)).

ue_adminmax
: Reserved for future use.

ue_jpelimit
: Per-job maximum number of massively parallel processing (MPP) processing elements (PEs). Two values exist, one for batch and the other for interactive work; udbgen(8) and udbsee(1) present the values as jpelimit[b] for batch and jpelimit[i] for interactive. A value of 0 in this field makes the MPP system inaccessible by the job. This field is present but not used on Cray Research systems without an MPP system.

ue_jmpptime Per-job maximum amount of wall-clock time (in seconds) that the MPP system can be reserved by the job. Two values exist, one for batch and the other for interactive work; udbgen(8) and udbsee(1) present the values as jmpptime[b] for batch and jmpptime[i] for interactive. A value of 0 in this field makes the MPP system inaccessible by the job. This field is present but not used on Cray Research systems without an MPP system.

ue_jmppbarrier
Per-job maximum number of MPP barriers. Two values exist, one for batch and the other for interactive work; udbgen(8) and udbsee(1) present the values as jmppbarrier[b] for batch and jmppbarrier[i] for interactive. This field is present but not used on Cray Research systems without an MPP system.

ue_pmpptime Per-process maximum amount of wall-clock time (in seconds) that the MPP system can be reserved by the process. Two values exist, one for batch and the other for interactive work; udbgen(8) and udbsee(1) present the values as pmpptime[b] for batch and pmpptime[i] for interactive. A value of 0 in this field makes the MPP system inaccessible by the process. This field is present but not used on Cray Research systems without an MPP system.

ue_pcorelim Per-process maximum core file limit in units of 512 words. This represents the maximum size of a core file that the process can create. If the size of the process is larger than this limit, a partial core file will be created. A partial core file contains just the user and user common structures. Two values exist, one for batch and the other for interactive work; udbgen(8) and udbsee(1) present the values as pcorelim[b] for batch and pcorelim[i] for interactive.

ue_pfdlimit Per-process maximum open file limit. This represents the maximum number of file descriptors that a process belonging to this user can allocate. This limit is restricted by the value of OPEN_MAX (64) at the low end, and the system configurable value of K_OPEN_MAX at the high end. Two values exist, one for batch and the other for interactive work; udbgen(8) and udbsee(1) present these values as pfdlimit[b] for batch and pfdlimit[i] interactive.

ue_mincomps Minimum security compartments. The field is actually a bit list but is represented externally as a comma-separated list of compartments. See ue_comparts for more information.

ue_jshmsegs Number of shared memory segments a job can create. This field always present but is used only on CRAY T90 series systems supporting shared memory.

ue_jshmsize Per-job maximum shared memory allocation in units of 512 words. This field always present but is used only on CRAY T90 series systems supporting shared memory.

ue_jsocbflim
Per-job maximum amount of socket buffer space in units of 512 words. If this field is 0, an unlimited amount of socket buffer space is allowed.

The ue_cpu_quota and ue_cpu_quota_used fields hold values expressed in seconds * 10. Externally, udbgen(8) and udbsee(1) accept and display these fields in the form *vvv.v* (seconds and tenths of a second). The internal form is used so that tenth-of-second resolution can be maintained in a 32-bit field. The maximum value allowed in these fields is 4294967295, which limits the maximum CPU quota to 429,496,729.5 seconds.

Description of `struct udbstat`

```
struct udbstat {
    int       add;            number of add record requests
    int       dbhits;         number of data block rereads
    int       dbread;         number of data blocks read
    int       dbwrite;        number of data blocks written
    int       delete;         number of delete by name requests
    int       getnam;         number of get by name requests
    int       getudb;         number of sequential read requests
    int       getudbchain;    number of chain reads
    int       getuid;         number of get by uid requests
    int       hdread;         number of header block reads
    int       hdwrite;        number of header block writes
    int       lock;           number of lock requests
    int       maxuid;         highest value UID in the database
    int       nhread;         number of name hash blocks read
    int       nhwrite;        number of name hash blocks written
    int       rewrite;        number of rewrite requests
    int       uhread;         number of uid hash blocks read
    int       uhwrite;        number of uid hash blocks written
    int       unlock;         number of unlock requests
    int       version;        database version number
    int       maxrecs;        maximum number of records in the udb
};
```

Description of `struct udbdefault`

```
struct udbdefault {
    long      jcpulim[MAXUE_RCLASS];     default job CPU limit
    long      pcpulim[MAXUE_RCLASS];     default process CPU limit
    long      jmemlim[MAXUE_RCLASS];     default job memory limit
    long      pmemlim[MAXUE_RCLASS];     default process memory limit
    long      jsdslim[MAXUE_RCLASS];     default job SDS limit
    long      psdslim[MAXUE_RCLASS];     default process SDS limit
    long      jfilelim[MAXUE_RCLASS];    default job new file space allocation limit
    long      pfilelim[MAXUE_RCLASS];    default process new file space allocation limit
    unsigned char                        default tape limits
              jtapelim[MAXUE_RCLASS][MAXUE_TAPETYPE];
    int       nice[MAXUE_RCLASS];        default nice value
    int       jproclim[MAXUE_RCLASS];    default job process limit
    int       jpelimit[MAXUE_RCLASS];    default MPP PE limit
    int       jmpptime[MAXUE_RCLASS];    default MPP time limit
    int       jmppbarrier[MAXUE_RCLASS]; default MPP barrier limit
    int       pmpptime[MAXUE_RCLASS];    default per process MPP time limit
    int       pcorelim[MAXUE_RCLASS];    default per proc core file limit [512 words]
    int       pfdlimit[MAXUE_RCLASS];    default file descriptor limit
    int       jshmsegs[MAXUE_RCLASS];    default created shared memory segments limit
    int       jshmsize[MAXUE_RCLASS];    default job shared memory size limit
    int       jsocbflim[MAXUE_RCLASS];   default per job max socket buffer limit [512 words]
};
```

Description of `struct udbtmap`

```
        struct udbtmap {
               struct {
            char      name[MAXUE_TNAME + 1];     tape name
               } mt_entry[MAXUE_TAPETYPE];
        };
```

NOTES

The external representation of the records in the UDB is located in the files `libc/udb/uentrydb.c` and `libc/udb/libudb.h`. To save space in the files, the data is packed in `uentrydb.c`, using the structure defined in that file. In the extension files, data is tagged and compressed; all zero-valued fields are discarded. Transformation functions in the library convert between the file and `struct udb` representations.

In the previously described `struct udb`, the following fields are included only for compatibility with previous releases: `ue_shplimit`, `ue_shmlimit`, `ue_mask`, `ue_warnings`, `ue_reason`, `ue_age`, and `ue_limflags`. These fields are obsolete and are not referenced by the UDB.

WARNINGS

Successive calls to function `getudbnam` return a pointer to the same static memory space each time they are called; these calls overwrite the same data area. Use caution when working with more than one UDB structure at a time.

Most functions in this library leave the `udb` file open to assure reasonable performance for multiple calls. If it is important that the program in which the calls are made can be restarted, an `endudb`() call must be made to close the `udb` file after the access is complete.

RETURN VALUES

Successful calls to type `int` functions return with the value `UDB_SUCCESS` (0); unsuccessful calls return `UDB_FAIL` (–1). Structure pointer type requests return with a pointer to a structure if successful, or the pointer value `UDB_NULL ((struct udb *) 0)` if they fail (except for some options of `getudbchain`). In any failing case, the `extern int udb_errno` variable contains a reason code from the following list. The descriptions of some reason codes state that further information for the failure can be found in the `errno` file (see header `errno.h`).

Symbol	Description
`UDBERR_BADPATH`	Return value 1. Path name specified in the `setudbpath( )` call is bad (see `errno.h`(3C)).
`UDBERR_BADUFN`	Return value 2. A UDB file name is illegal. `UDB` or `UDBPUB` is zero length, it ends with '`/`', or the full name is longer than `UDBFNAMELEN`.
`UDBERR_CHANGED`	Return value 3. Another user changed the database while `getudbchain( )` was being used. The current position is lost.
`UDBERR_CORRUPT`	Return value 4. Something is wrong with the content of the database. The files must be regenerated.
`UDBERR_CREATE`	Return value 5. Error in creating the database (see `errno.h`(3C)).
`UDBERR_DEADLK`	Return value 6. Protected user database could not be locked, because a deadlock condition would have resulted.
`UDBERR_END`	Return value 7. No more records in database.
`UDBERR_IDCHG`	Return value 8. An update must be performed as a `deleteudb( )` and `addudb( )` sequence because the name or UID has changed.
`UDBERR_INTERR`	Return value 9. An internal program error occurred.
`UDBERR_IOERR`	Return value 10. An I/O error occurred (see `errno.h`(3C)).
`UDBERR_NAMEINUSE`	Return value 11. The user-name of the record to be created is already in use.

UDBERR_NOLCK	Return value 12. Protected user database could not be locked, because some privileged process has the protected user database locked by a means other than the `lockudb( )` call. If this occurs, an error exists in some non-kernel (user-level) system software.
UDBERR_NOPOS	Return value 13. Undefined current position.
UDBERR_NORW	Return value 14. Caller does not have read or write access on a protected database.
UDBERR_NOSUCHUSER	Return value 15. No such record in the database.
UDBERR_NOTLOCKED	Return value 16. User information cannot be rewritten because the database was not locked.
UDBERR_UDBCHAIN	Return value 17. The call to `getudbchain( )` has an unknown option code.
UDBERR_VERSION	Return value 18. The software level of the database access functions linked with the caller is incompatible with the current version of the UDB.
UDBERR_BADDEFER	Return value 19. The `DEFERTORESGRP` flag was set on more than four chained entries.

FILES

/etc/udb	User validation file containing user control limits
/etc/udb.public	Public version of the user database file
/etc/udb_2/udb.index	Public extention file index
/etc/udb_2/udb.priva	Private field extension file
/etc/udb_2/udb.pubva	Public field extension file
/etc/passwd	Traditional password file

Other path names can be used if `setudbpath` has been called.

SEE ALSO

errno.h(3C), getpwent(3C), perror(3C)

chacid(1), guest(1), login(1), newacct(1), nice(1), quota(1), udbsee(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

acctid(2), chacid(2), getpermit(2), limits(2), mknod(2), mount(2), msgget(2), nice(2), plock(2), semget(2), setgid(2), setgroups(2), setuid(2), shmget(2), ulimit(2), umount(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

cpu(4), udb(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

accton(8), chroot(8), csaswitch(8), ddms(8), diskusg(8), tplabel(8), udbgen(8), xadmin(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`limits.h` – Library header for integral type limits

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

TYPES

None

MACROS

The header `limits.h` defines the various machine-dependent numerical limits for Cray Research systems. Each of these limits expands into constant expressions suitable for use in a `#if` preprocessing directive. The macros and definitions are shown in the following table:

Macro	Standard	Definition
`CHAR_BIT`	ISO/ANSI	Number of bits for smallest object that is not a bit-field (byte).
`_WORD_BIT`	CRI	Number of bits in a word.
`SCHAR_MIN`	ISO/ANSI	Minimum value for an object of type `signed char`.
`SCHAR_MAX`	ISO/ANSI	Maximum value for an object of type `signed char`.
`UCHAR_MAX`	ISO/ANSI	Maximum value for an object of type `unsigned char`.
`CHAR_MIN`	ISO/ANSI	Minimum value for an object of type `char`.
`CHAR_MAX`	ISO/ANSI	Maximum value for an object of type `char`.
`MB_LEN_MAX`	ISO/ANSI	Maximum number of bytes in a multibyte character, for any supported locale.
`SHRT_MIN`	ISO/ANSI	Minimum value for an object of type `short int`.
`SHRT_MAX`	ISO/ANSI	Maximum value for an object of type `short int`.
`USHRT_MAX`	ISO/ANSI	Maximum value for an object of type `unsigned short int`.
`INT_MIN`	ISO/ANSI	Minimum value for an object of type `int`.
`INT_MAX`	ISO/ANSI	Maximum value for an object of type `int`.
`UINT_MAX`	ISO/ANSI	Maximum value for an object of type `unsigned int.`
`LONG_MIN`	ISO/ANSI	Minimum value for an object of type `long int`.
`LONG_MAX`	ISO/ANSI	Maximum value for an object of type `long int`.
`ULONG_MAX`	ISO/ANSI	Maximum value for an object of type `unsigned long int`.

The values in `limits.h` are as shown in the following table. For comparison, the "CRI Value" column in the table is followed by a column listing the minimum value (in magnitude) required by the standard.

Macro	CRI Value	Minimum for ISO/ANSI C
`CHAR_BIT`	8	8
`_WORD_BIT`†	64	–
`SCHAR_MIN`	–128	–127
`SCHAR_MAX`	127	127
`UCHAR_MAX`	255	255
`CHAR_MIN`	0	–
`CHAR_MAX`	255	–
`MB_LEN_MAX`	1	1
`SHRT_MIN`	–8388608 (24-bit A register) –2147483648 (32-bit A register)	–32767
`SHRT_MAX`	8388607 (24-bit A register) 2147483647 (32-bit A register)	32767
`USHRT_MAX`	16777215 (24-bit A register) 4294967295 (32-bit A register)	65535
`INT_MIN`	–35184372088832 (default) –9223372036854775808 (`-h nofastmd`)	–32767
`INT_MAX`	35184372088831 (default) 9223372036854775807 (`-h nofastmd`)	32767
`UINT_MAX`	18446744073709551615	65535
`LONG_MIN`	–9223372036854775808	–2147483647
`LONG_MAX`	9223372036854775807	2147483647
`ULONG_MAX`	18446744073709551615	4294967295

† `_WORD_BIT` is a CRI extension; not specified by the standard

See the *Cray Standard C Reference Manual*, Cray Research publication SR–2074, for information about the `-h fastmd` and `-h nofastmd` command line options. (The `-h fastmd` command-line option is the compiler default.)

FUNCTION DECLARATIONS

None

CAUTIONS

On CRI systems, comparisons between two integer values is done by subtraction. That is, the expression (A > B) is evaluated by performing the operation (A - B) and testing the sign of the result. If, however, A and B are signed variables with different signs and either A or B is greater than (2**62–1) or less than –(2**62), integer overflow can occur and the sign of the result may be incorrect. For this reason, it is not safe to use the values LONG_MAX or LONG_MIN as an arbitrary large number with the relational operators. Instead, pick a smaller number; LONG_MAX/2 is sufficiently small. INT_MAX in the absence of the -h nofastmd command-line option is also safe. Comparison of unsigned integer values is always safe.

SEE ALSO

float.h(3C), values.h(3C)

NAME

`loaded`, `loaded_data` – Tells whether soft external routine/data is loaded

SYNOPSIS

```
#include <infoblk.h>
int loaded ( );
int loaded_data ( );
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The `loaded` function tells whether a soft-referenced function has been loaded into a user program. The `loaded_data` function tells whether a soft-referenced data item has been loaded into a user program. Both take as arguments the address of the specified function or data item.

RETURN VALUES

The `loaded` and `loaded_data` functions return 1 if the specified function or data item has been loaded into the user program; if it has not been loaded, they return 0.

EXAMPLES

The following example shows how `loaded` and `loaded_data` execute:

```
#include <stdio.h>
#include <infoblk.h>

#pragma _CRI soft data, func
extern int data;
extern int func(void);

main()
{
      if (loaded_data(&data))
            printf("data loaded; value %d\n", data);
      else
            printf("data NOT loaded\n");
      if (loaded(func))
            printf("func loaded; returns %d\n", func());
      else
            printf("func NOT loaded\n");
}
```

NAME

`locale` – Introduction to locale information functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The locale information functions and header file `locale.h` provide various means for setting and accessing a specific set of run-time environment variables that may vary with culture, geography, or other factors related to location. The set of locale information variables affects the following:

- Formatting of monetary information
- Representation of date and time
- Classification of characters
- Collation of characters and character strings
- Formatting of numeric values

The ANSI standard defines one structure type, `struct lconv`, which contains members related to the formatting of numeric values. These members are as follows:

`char *decimal_point`
The decimal-point used to format nonmonetary quantities.

`char *thousands_sep`
The characters used to separate groups of digits before the decimal-point character in formatted nonmonetary quantities.

`char *grouping`
The characters that indicate the size of each group of digits in formatted nonmonetary quantities.

The elements of `grouping` are interpreted according to the following:

`CHAR_MAX`	No further grouping will be performed.
`0`	The previous element will be used repeatedly for the remainder of the digits.
other	The integer value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits before the current group.

`char *int_curr_symbol`
The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in ISO 4217 Codes for the Representation of Currency and Funds. The fourth character (immediately preceding the null character) is the character used to separate the international currency symbol from the monetary quantity.

`char *currency_symbol`
The local currency symbol applicable to the current locale.

`char *mon_decimal_point`
The decimal-point used to format monetary quantities.

`char *mon_thousands_sep`
The separator for groups of digits before the decimal-point in formatted monetary quantities.

`char *mon_grouping`
A string whose elements indicate the size of each group of digits in formatted monetary quantities.

The elements of `mon_grouping` are interpreted according to the following:

`CHAR_MAX`	No further grouping will be performed.
`0`	The previous element will be used repeatedly for the remainder of the digits.
other	The integer value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits before the current group.

`char *positive_sign`
The string used to indicate a nonnegative-valued formatted monetary quantity.

`char *negative_sign`
The string used to indicate a negative-valued formatted monetary quantity.

`char int_frac_digits`
The number of fractional digits (those after the decimal-point) to be displayed in a internationally formatted monetary quantity.

`char frac_digits`
The number of fractional digits (those after the decimal-point) to be displayed in a formatted monetary quantity.

`char p_cs_precedes`
Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a nonnegative formatted monetary quantity.

`char p_sep_by_space`
Set to 1 or 0 if the `currency_symbol` respectively is or is not separated by a space from the value for a nonnegative formatted monetary quantity.

`char n_cs_precedes`
Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a negative formatted monetary quantity.

`char n_sep_by_space`
Set to 1 or 0 if the `currency_symbol` respectively is or is not separated by a space from the value for a negative formatted monetary quantity.

`char p_sign_posn`
Set to a value that indicates the positioning of the `positive_sign` for a nonnegative formatted monetary quantity. (See the note following the `n_sign_posn` descriptions.)

`char n_sign_posn`
Set to a value that indicates the positioning of the `negative_sign` for a negative formatted monetary quantity.

The value of `p_sign_posn` and `n_sign_posn` is interpreted according to the following:

0 Parentheses surround the quantity and `currency_symbol`.

1 The sign string precedes the quantity and `currency_symbol`.

2 The sign string succeeds the quantity and `currency_symbol`.

3 The sign string immediately precedes the `currency_symbol`.

4 The sign string immediately succeeds the `currency_symbol`.

The members of the structure with type `char *` are pointers to strings, any of which (except `decimal_point`) can point to "" to indicate that the value is not available in the current locale or is of 0 length. The members with type `char` are nonnegative numbers; to indicate that the value is not available in the current locale, any members can be `CHAR_MAX`.

The method by which users defined their own locales (described in previous releases on this man page) is no longer supported. It is no longer necessary because the `localedef` command provides a superset of this functionality. If you use the old method and try to compile a program to generate a locale, the program will not compile. However, existing binary files that create locales will work through the UNICOS 9.0 release. Any successfully generated locale files will continue to be accepted by `setlocale`(3C).

Associated Headers

<locale.h>

Associated Functions

Function	Description
`iconv`(3C), `iconv_close`(3C), `iconv_open`(3C)	Converts a sequence of characters from one codeset into another codeset
`localeconv`(3C)	Reports program's numeric formatting conventions
`nl_langinfo`(3C)	Points to language information.
`setlocale`(3C)	Selects program's locale

NOTES

The UNICOS C library functions that set or access these variables include the following: `localeconv`(3C), `nl_langinfo`(3C), `printf`(3C), `scanf`(3C), `strcoll`(3C), `strfmon`(3C), `strftime`(3C), `strptime`(3C), `strxfrm`(3C), `wcscoll`(3C), `wcsxfrm`(3C), and the character-handling functions (see `character`(3C)).

NAME

localeconv – Reports program's numeric formatting conventions

SYNOPSIS

```
#include <locale.h>
struct lconv *localeconv (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The localeconv function sets the components of an object with type struct lconv with values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale.

RETURN VALUES

The localeconv function returns a pointer to the filled-in object. The structure pointed to by the return value cannot be modified by the program, but may be overwritten by a subsequent call to localeconv. In addition, calls to setlocale with categories LC_ALL, LC_MONETARY, or LC_NUMERIC can overwrite the contents of the structure.

SEE ALSO

locale(3C), locale.h(3C), setlocale(3C)

NAME

`locale.h` – Library header for locale information functions

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The header file `locale.h` defines locale information functions.

Types

The types declared in `locale.h` are as follows:

Type	Description
`struct lconv`	Structure that contains members related to the formatting of numeric values. This conforms to the ISO/ANSI standard.

Macros

The macros defined in the header file `locale.h` are as follows (unless noted, macros conform to the ISO/ANSI standard):

Macro	Description

`LC_ALL`
`LC_COLLATE`
`LC_CTYPE`
`LC_MONETARY`
`LC_NUMERIC`
`LC_TIME`
	Each of these macros expands to an integral constant expression with distinct values, and is suitable for use as the first argument to the `setlocale`(3C) function.

`LC_MESSAGES`	Same as above. Conforms with POSIX P1003.2.
`NULL`	An implementation-defined null pointer constant, equal to 0 on Cray Research systems.

Function Declarations

The `localeconv` and `setlocale` functions are declared in the header file `locale.h`.

SEE ALSO

`ctype.h`(3C), `locale`(3C)

NAME

LOCKASGN – Identifies an integer variable intended for use as a lock

SYNOPSIS

CALL LOCKASGN(*name* [,*value*])

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

LOCKASGN identifies an integer variable that the program intends to use as a lock. The program must call the LOCKASGN routine for each lock variable before it is used with any other lock routines. The multitasking library gives a lock an initial state of off or cleared. A data statement can initialize the lock to the value in the optional argument, allowing the program to assign it in a routine. The first call assigns the lock, and further calls are ignored.

The following is a list of valid variables for this routine:

Argument	Description
name	Name of an integer variable to be used as a lock. The library stores an identifier into this variable; do not modify this variable after the call to LOCKASGN.
value	The initial integer value of the lock variable. An identifier should be stored into the variable only if it contains the value. If *value* is not specified, an identifier is stored into the variable unconditionally.

CAUTIONS

For SPARC systems, the *value* parameter is optional, and LOCKASGN is not predeclared (not intrinsic). Therefore, if a call is made to it with only the *name* parameter, LOCKASGN must be declared with an INTERFACE block in the calling module.

EXAMPLES

```
      PROGRAM MULTI
      INTEGER LKINPUT,LKOUTPUT,LKCALL
      REAL    INDATA(20000),OUTDATA(20000)
      COMMON  /CBINPUT/  LKINPUT,INDATA
      COMMON  /CBOUTPUT/ LKOUTPUT,OUTDATA
      COMMON  /MISC/     LKCALL
C     ...
      CALL LOCKASGN (LKINPUT)
      CALL LOCKASGN (LKOUTPUT)
      CALL LOCKASGN (LKCALL)
C     ...
      END

      SUBROUTINE SUB1
      COMMON /LOCK1/ LOCK1
      DATA LOCK1 /-1/
C     ...
      CALL LOCKASGN (LOCK1,-1)
C     ...
      END
```

NAME

lockf – Provides record locking on files

SYNOPSIS

```
#include <unistd.h>
int lockf (int fildes, int function, long size);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

AT&T extension

DESCRIPTION

The lockf function allows sections of a file to be locked with advisory or mandatory write locks, depending on the mode bits of the file (see chmod(2)). Locking calls from other processes that attempt to lock the locked file section either return an error value or are put to sleep until the resource becomes unlocked. All locks for a process are removed when the process terminates. (See fcntl(2) for more information about record locking.) The lockf function does not work on NFS files.

The *fildes* operand is an open file descriptor. The file descriptor must have O_WRONLY or O_RDWR permission to establish a lock with this function call.

The *function* operand is a control value that specifies the action to be taken. Permissible values for *function* are defined in the header file unistd.h, as follows:

```
#define  F_ULOCK   0    /* Unlock a previously locked section */
#define  F_LOCK    1    /* Lock a section for exclusive use */
#define  F_TLOCK   2    /* Test and lock a section for exclusive use */
#define  F_TEST    3    /* Test section for other processes locks */
```

All other values of *function* are reserved for future extensions and result in an error return if used.

F_TEST detects whether a lock by another process is present on the specified section. F_LOCK and F_TLOCK both lock a section of a file if the section is available. F_ULOCK removes locks from a section of the file.

The *size* operand is the number of contiguous bytes to be locked or unlocked. The resource to be locked starts at the current offset in the file; it extends forward for a positive size and backward for a negative size (the preceding bytes up to but not including the current offset). If *size* is 0, the section from the current offset through the largest file offset is locked (that is, from the current offset through the present or any future end-of-file). An area need not be allocated to the file to be locked, because such locks can exist past the end-of-file.

The sections locked with F_LOCK or F_TLOCK can, in whole or in part, contain or be contained by a previously locked section for the same process. When this occurs, or if adjacent sections are locked, the sections are combined into a single section. If the request requires that a new element be added to the table of active locks and this table is already full, an error is returned, and the new section is not locked.

F_LOCK and F_TLOCK requests differ only by the action taken if the resource is not available. F_LOCK causes the calling process to sleep until the resource is available. F_TLOCK causes the function to return a -1 and set errno to EACCES error if the section is already locked by another process.

F_ULOCK requests can, in whole or in part, release one or more locked sections controlled by the process. When sections are not fully released, the remaining sections are still locked by the process. Releasing the center section of a locked section requires an additional element in the table of active locks. If this table is full, an EDEADLK error is returned, and the requested section is not released.

A potential for deadlock occurs if a process that controls a locked resource is put to sleep by accessing another process's locked resource. Thus, calls to lockf or fcntl scan for a deadlock before sleeping on a locked resource. An error return is made if sleeping on the locked resource would cause a deadlock.

Sleeping on a resource is interrupted with any signal. The alarm(2) system call can be used to provide a time-out facility in applications that require this facility.

If the lockf function fails, errno is set to one of the following values, defined in the header file errno.h:

Error Code	Description
EBADF	File descriptor *fildes* is not a valid open descriptor.
EACCES	*Cmd* is F_TLOCK or F_TEST, and the section is already locked by another process.
EDEADLK	*Cmd* is F_LOCK, and a deadlock would occur. Also the *cmd* is F_LOCK, F_TLOCK, or F_ULOCK, and the number of entries in the lock table exceeds the number allocated on the system.
ECOMM	File descriptor *fildes* is on a remote machine, and the link to that machine is no longer active.

WARNINGS

Unexpected results can occur in processes that do buffering in the user address space. The process can later read/write data that is/was locked. The standard I/O package is the most common source of unexpected buffering.

Because in the future, variable errno will be set to EAGAIN rather than EACCES when a section of a file is already locked by another process, portable application programs should expect and test for either value.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and errno is set to indicate the error.

SEE ALSO

alarm(2), chmod(2), close(2), creat(2), fcntl(2), intro(2), open(2), read(2), write(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

LOCKOFF – Clears a lock and returns control to the calling task

SYNOPSIS

CALL LOCKOFF(*lock*)

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

LOCKOFF clears a lock and returns control to the calling task. Clearing the lock can allow another task to resume execution, but this is transparent to the task calling LOCKOFF.

The following is a valid argument for this routine:

Argument	Description
lock	Name of an integer variable used as a lock.

EXAMPLES

```
      PROGRAM MULTI
      INTEGER LKOUTPUT
      REAL    OUTDATA(20000)
      COMMON  /CBOUTPUT/ LKOUTPUT,OUTDATA
C     ...
      CALL LOCKASGN (LKOUTPUT)
C     ...
C
      CALL LOCKON (LKOUTPUT)
      DO 100 I=1,20000
        OUTDATA(I)=MAX(OUTDATA(I),0)
 100  CONTINUE
      CALL LOCKOFF (LKOUTPUT)
C
C     ...
      END
```

SEE ALSO

LOCKON(3F), LOCKTEST(3F), multif(3F), NLOCKOFF(3F), NLOCKON(3F),

NAME

LOCKON – Sets a lock and returns control to the calling task

SYNOPSIS

CALL LOCKON(*lock*)

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

LOCKON sets a lock and returns control to the calling task. If the lock is already set when LOCKON is called, the task calling LOCKON waits until the lock is cleared by another task and then sets it. This means that placing LOCKON before a critical region ensures that the code in the region is executed only when the task has unique access to the lock. Calls to LOCKON cannot be nested.

The following is a valid argument for this routine:

Argument	**Description**
lock	Name of an integer variable used as a lock.

SEE ALSO

LOCKOFF(3F), LOCKTEST(3F), multif(3F), NLOCKOFF(3F), NLOCKON(3F)

NAME

LOCKREL – Releases the identifier assigned to a lock

SYNOPSIS

CALL LOCKREL(*name*)

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

LOCKREL releases the identifier assigned to a lock.

The following is a valid argument for this routine:

Argument	Description
name	Name of an integer variable used as a lock.

If the lock is set or a task is waiting for the lock when LOCKREL is called, an error results. This routine detects some errors that arise when a task is waiting for a lock that is never cleared. The lock variable can be reused following another call to LOCKASGN(3F).

EXAMPLES

```
      PROGRAM MULTI
      INTEGER LKOUTPUT
      REAL    INDATA(20000),OUTDATA(20000)
      COMMON  /CBOUTPUT/ LKOUTPUT,OUTDATA
C     ...
      CALL LOCKASGN (LKOUTPUT)
C     ...
C
      CALL LOCKON (LKOUTPUT)
      DO 100 I=1,20000
        OUTDATA(I)=MAX(OUTDATA(I),0)
 100  CONTINUE
      CALL LOCKOFF (LKOUTPUT)
C     ...
      CALL LOCKREL (LKOUTPUT)
C
      END
```

SEE ALSO

LOCKASGN(3F)

NAME

LOCKTEST – Tests a lock to determine its state (locked or unlocked)

SYNOPSIS

```
LOGICAL LOCKTEST
return = LOCKTEST(lock)
```

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

LOCKTEST tests a lock to determine its state. By using this function, a task can avoid blocking on a set lock.

The following is a list of valid variables for this routine:

Arguemnt	Description
return	A logical .TRUE. if the lock was originally set. A logical .FALSE. if the lock was originally clear. The lock variable's state is always set to locked upon return.
name	Name of an integer variable used as a lock.

Unlike a task using LOCKON(3F), the task does not wait if the lock is already locked. A task using LOCKTEST must always test the return value before continuing.

NOTES

LOCKTEST and *return* must be declared type LOGICAL in the calling module.

SEE ALSO

multif(3F), LOCKOFF(3F), LOCKON(3F), NLOCKOFF(3F), NLOCKON(3F)

NAME

logb, logbf, logbl – Returns the signed exponent of its argument

SYNOPSIS

CRAY T90 systems with IEEE floating-point hardware:

```
#include <fp.h>

double logb(double x);
float logbf(float x);
long double logbl(long double x);
```

Cray MPP systems:

```
#include <fp.h>

double logb(double x);
```

IMPLEMENTATION

Cray MPP systems (implemented as a macro)
CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

ANSI/IEEE Std 754-1985
X3/TR-17:199x

DESCRIPTION

The logb function or macro extracts the exponent of *x* as a signed integral value in the format of *x*. If *x* is subnormal, it is treated as though it were normalized; thus, for positive finite *x*, the following is true:

$1 \leq x * \texttt{FLT\_RADIX}^{-\texttt{logb(x)}} < \texttt{FLT\_RADIX}$

RETURN VALUES

Each function or macro returns the signed exponent of its argument.

SEE ALSO

float.h(3C) for a description of the FLT_RADIX macro

Migrating to the CRAY T90 Series IEEE Floating Point, Cray Research publication SN–2194

NAME

logname – Returns the login name of the user

SYNOPSIS

```
#include <stdlib.h>
char *logname (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

AT&T extension

DESCRIPTION

The logname function returns a pointer to the null-terminated login name of the user; it extracts the $LOGNAME variable from the user's environment.

CAUTIONS

This method of determining a login name is subject to forgery.

FILES

/etc/profile	Systemwide shell start-up file

SEE ALSO

env(1), login(1), sh(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

profile(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

`lsearch`, `lfind` – Performs a linear search and update

SYNOPSIS

```
#include <search.h>

void *lsearch (const void *key, void *base, size_t *nelp,
size_t width, int (*compar)(const void *, const void *));

void *lfind (const void *key, const void *base, size_t *nelp,
size_t width, int (*compar)(const void *, const void *));
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The `lsearch` function performs linear searches. It returns a pointer into a table that indicates where data may be found. If data does not occur, it is added at the end of the table. The *key* argument points to the data to be sought in the table. The *base* argument points to the first element in the table. The integer to which *nelp* points contains the current number of elements in the table. (This integer is incremented if the data is added to the table.) The value of *width* is the size in bytes of an element. You must supply the name of the comparison function, *compar* (for example, `strcmp`). It is called with two arguments that point to the elements being compared. If the elements are equal, the function must return 0; otherwise, it returns nonzero.

The `lfind` function is the same as `lsearch` except that if the data is not found, it is not added to the table. Instead, a null pointer is returned.

NOTES

The pointers to the key and the element at the base of the table may be pointers to any type.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

The value required should be cast into type pointer-to-element.

CAUTIONS

Undefined results can occur if not enough room is in the table to add a new item.

RETURN VALUES

If the searched for data is found, both `lsearch` and `lfind` return a pointer to it; otherwise, `lfind` returns null and `lsearch` returns a pointer to the newly added element.

EXAMPLES

The following fragment reads in ≤ `TABSIZE` strings of length ≤ `ELSIZE` and stores them in a table, eliminating duplicates:

```
#include <stdio.h>
#include <string.h>
#include <search.h>

#define TABSIZE 50
#define ELSIZE 120

     char line[ELSIZE], tab[TABSIZE][ELSIZE];
     size_t nel = 0;
     .
     .
     .
     while (fgets(line, ELSIZE, stdin) != NULL && nel < TABSIZE)
      (void) lsearch(line, tab, &nel, ELSIZE, strcmp);
     .
     .
     .
```

SEE ALSO

`bsearch`(3C), `hsearch`(3C), `tsearch`(3C)

NAME

malloc, calloc, free, realloc, malloc_inplace, malloc_expand, malloc_extend, malloc_howbig, malloc_isvalid, malloc_space, malloc_brk, malloc_limit, malloc_check, malloc_stats, malloc_tron, malloc_troff, malloc_etrace, malloc_dtrace, mallopt, mallinfo, malloc_error – Memory management functions

SYNOPSIS

```
#include <stdlib.h> or #include <malloc.h>
void *malloc (size_t size);
void *calloc (size_t nmemb, size_t size);
void free (void *ptr);
void *realloc (void *ptr, size_t size);
#include <malloc.h>
void *malloc_inplace (void *ptr, size_t size);
size_t malloc_expand (void *ptr);
size_t malloc_extend (void *ptr);
size_t malloc_howbig (void *ptr);
int malloc_isvalid (void *ptr);
size_t malloc_space (long nbytes);
int malloc_brk (void *endds);
void malloc_limit (size_t thresh, size_t limit);
int malloc_check (int level);
void malloc_stats (int level);
void malloc_tron (void);
void malloc_troff (void);
void malloc_etrace (long funcs);
void malloc_dtrace (long funcs);
int mallopt (int cmd, int value);
struct mallinfo mallinfo (void);
extern long malloc_error;
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (`malloc`, `calloc`, `free`, and `realloc` only)

AT&T extension (`mallopt` and `mallinfo` only)

CRI extension (all others)

DESCRIPTION

The `malloc` function returns a pointer to a block of at least *size* bytes suitably aligned for any use. `malloc` calls `sbreak` (see `brk`(2)) to get more memory from the system when no suitable space is already free. The space returned is left uninitialized.

The `calloc` function allocates space for an array of *nmemb* objects, each of whose size is *size* bytes. The space is initialized to all bits 0.

The `free` function causes the block to which *ptr* points to be deallocated, that is, made available for further allocation. If *ptr* is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by a memory manager function, or if the space has already been deallocated by a call to `free` or `realloc`, `malloc_error` is set to indicate the error, and `free` returns.

The `realloc` function changes the size of the block to which *ptr* points to the size (in bytes) specified by *size*. The contents of the block are unchanged up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion of the block is indeterminate. If *ptr* is a null pointer, the `realloc` function behaves like the `malloc` function for the specified size. If *size* is 0 and *ptr* is not a null pointer, the block to which it points is freed. Otherwise, if *ptr* does not match a pointer earlier returned by a memory manager function, or if the space has been deallocated by a call to the `free` or `realloc` function, the `malloc_error` variable is set to indicate the error, and `realloc` returns a null pointer. If the space cannot be allocated, the block to which *ptr* points is unchanged.

The `malloc_inplace` function tries to change the size of the block to which *ptr* points to the size (in bytes) specified by *size*. However, if the size of the block cannot be changed without moving the block, the request fails and a null pointer is returned.

The `malloc_expand` function causes the memory block to which *ptr* points to grow as large as possible, without causing an `sbreak` or moving the block. It returns the new size of the block in bytes.

The `malloc_extend` function returns the expanded size of the block to which *ptr* points without actually doing the expansion. If the block is at the end of memory, a very large number is returned.

The `malloc_howbig` function returns the current size (in bytes) of the block to which *ptr* points (which may not be the same as the original *size* argument to `malloc`, et al.)

The `malloc_isvalid` function returns a nonzero value if *addr* points to a valid allocated block of memory used by the memory manager.

The `malloc_space` function tries to return up to *nbytes* bytes of memory to the system (only possible if the last block in the heap is free). If *nbytes* is –1, it returns as much memory as possible. If *nbytes* is 0, it returns the number of bytes of memory that could be freed.

The `malloc_brk` function extends the end of the heap to the address specified in *endds*. If *endds* is already contained within the heap, `malloc_brk` does nothing. Any new space created is turned into a free block. If the heap cannot be extended to the desired address, `malloc_brk` returns –1; otherwise, it returns 0.

The `malloc_limit` function controls the behavior of `free` when the last block in the heap is freed. If the *threshold* argument is nonzero, and the last block in the arena is free and larger than *threshold*, all but the last *limit* number of *nbytes* of that block are returned automatically to the system. The minimum positive value for *threshold* (512 Kwords) is silently enforced. Initially, *threshold* and *limit* are 0.

The `malloc_check` function checks the consistency of `malloc`'s memory structure. If *level* is less than 0, `malloc_check` silently performs validation of the heap, and returns 0 if the heap is consistent, or nonzero if the heap has been corrupted. If *level* equals 0, `malloc_check` prints a message to `stderr` that describes the first inconsistency found. If *level* is greater than 0, `malloc_check` prints a line to `stderr` that describes each heap block in addition to checking the heap.

The `malloc_stats` function prints out memory manager statistics and heap block information to `stdout`. If *level* equals 0, `malloc_stats` reports the number of calls to each memory manager function, as well as summary statistics on the number and total size of the busy blocks, free blocks, and "spec" blocks (that is, blocks that are created by user calls to `sbreak`) in the heap. If *level* equals 1, `malloc_stats` prints a line with a `*` for each busy block, a `.` for each free block, and a `@` for each "spec" block, in addition to the level 0 statistics. If *level* equals 2, `malloc_stats` prints a line describing each heap block, in addition to the level 0 statistics. The number of calls for each function are only available by linking with the `libmalloc` library; all of the other information is available in the default memory manager.

The `malloc_tron`, `malloc_troff`, `malloc_etrace`, and `malloc_dtrace` macros trace calls to the memory manager and to `brk`(2), `sbrk`(2), and `sbreak`(2); however, they are operational only by linking with the `libmalloc` library. When tracing is on, each memory manager function prints a line to `stderr` that shows the function called, its arguments, a one or two-level traceback, and the return value of the function. The `malloc_tron` and `malloc_troff` macros turn tracing on and off, respectively. Tracing is off by default. You can use the `malloc_etrace` and `malloc_dtrace` macros can be used to enable or disable, respectively, a given set of functions. For a list of the functions that can be traced, see the header file `malloc.h`. You may combine the function values using `OR`. The –1 constant refers to all functions.

The `mallopt` function allows you to set various options in the memory manager. The values for the *cmd* argument are defined in the header file `malloc.h` as follows (the values marked (`libmalloc` only) are operational only if the `libmalloc` library has been linked into the program):

Value	**Description**

M_TRACE	(libmalloc only) If *value* ≥ 0, memory tracing is turned on, with the trace being written to file descriptor *value*. If *value* is less than 0, memory tracing is turned off.
M_ETRACE and M_DTRACE	(libmalloc only) These use the malloc_etrace and malloc_dtrace macros, respectively, with *value* passed as an argument to them.
M_LIMIT and M_THRESH	These set the free limit and threshold values, respectively, to *value* words (see malloc_limit).
M_BREAKSZ	If *value* is greater than 0, any sbreak(2) calls from the memory manager are made in multiples of *value* words. If *value* equals 0, the heap is fixed to its current size, and malloc returns 0 rather than calling sbreak(2).
M_MEMCHK	(libmalloc only) If *value* is 1, each call to a memory manager function checks the consistency of the heap; if the heap has been corrupted, it prints an error message to stderr and return an error status from the function called. If *value* is 2, the abort function is called rather than returning an error status (this flushes all open files and performs other cleanup actions before dumping core). If *value* is 3, an immediate core-dump is performed on detection of a corrupted heap. Setting *value* to 0 turns off heap consistency checking. If *value* is nonzero, the memory manager checks all calls to free, and prints an error message to stderr if an invalid pointer is passed as an argument.
M_LOWFIT	If *value* is nonzero, the memory manager keeps the large block free lists sorted by address. This slows down the memory manager, but ensures that blocks are allocated from the lowest address possible (which keeps the heap as small as possible). This can be called anywhere in a program; the free lists will be sorted if they are in an unsorted state when mallopt is called. Setting the environment variable MEMLOWFIT to nonzero has the same effect.
M_INDEF	If *value* is nonzero, calls to malloc and free initialize their blocks to the 'indef' pattern (this causes an operand range error if used as an address, or a floating-point exception if used as a floating-point number). You can also do this by setting the MEMINDEF environment variable to a nonzero number.
M_ABORT	If *value* is nonzero, malloc prints an error message and calls abort if the program runs out of memory (that is, a call to sbreak(2) fails). Setting the environment variable MEMABORT to nonzero has the same effect as setting *value* to nonzero.

The mallinfo function provides information that describes space usage. It returns the structure mallinfo, which contains the following members:

```
int arena;        /* total space in arena */
int ordblks;      /* number of ordinary blocks */
int smblks;       /* number of small blocks */
int hblks;        /* number of holding blocks */
int hblkhd;       /* space in holding block headers */
int usmblks;      /* space in small blocks in use */
int fsmblks;      /* space in free small blocks */
int uordblks;     /* space in ordinary blocks in use */
int fordblks;     /* space in free ordinary blocks */
int keepcost;     /* cost of enabling keep option (unused) */
```

The `malloc_error` variable is set if an error is encountered in the memory manager; it is never reset to 0. `malloc_error` may be set to the following values, defined in the header file `malloc.h`:

Value	Description
ME_EXTEND	Could not extend block in place
ME_BADLEN	Bad length supplied
ME_NOMEM	No memory available
ME_BADADDR	Address is outside bounds of heap
ME_ISFREE	A free block
ME_NOTBLOCK	Address is not a free/busy block
ME_CORRUPT	Corrupt memory arena
ME_BREAK	Arena truncated by user's `sbrk`(2)

NOTES

The `malloc` function uses a two-level allocation strategy for memory and time efficiency. Any requests to `malloc` larger than 64 bytes allocate a *large block*, which has a 2-word header. Free blocks also use the first 2 words of the block as free list pointers. All large blocks are on a doubly linked list, and there are 16 doubly linked free lists (hashed by size of the block). Requests to `malloc` smaller than 64 bytes allocate a large block of 4096+ bytes (called a *holding block*); from this block, many *small blocks* (which have a 1-word header) can be allocated and freed quickly. Different holding blocks are created when needed for different sizes of small blocks. Holding blocks are never freed, even if all of the small blocks within them have been freed.

A `free` of a large block causes any blocks immediately surrounding it to be coalesced into one free block. Each free list is unsorted, and the last block freed is put at the end of its corresponding free list (unless the `M_LOWFIT` option to `mallopt` is used).

The order of the algorithm that `realloc` (for large blocks) uses is as follows:

1. Coalesce any free block following the specified block.
2. Check if the block is at the end of memory, and use `sbreak`(2) to extend the block.
3. Check for a preceding free block, and slide the block lower in memory.
4. Use `malloc` to allocate a new block, and move the data to the new space.

ENVIRONMENT VARIABLES

You can use several environment variables to alter the behavior of the memory manager; their usage corresponds to the options for `mallopt`, as follows: The environment variables are:

Variable	Description
`MEMTRON`=*value*	(`libmalloc` only) If *value* is nonzero, the equivalent of `malloc_tron` is done.
`MEMCHK`=*value*	(`libmalloc` only) If *value* is nonzero, the equivalent of `mallopt(M_MEMCHK,` *value*`)` is done.
`MEMINDEF`=*value*	If *value* is nonzero, the equivalent of `mallopt(M_INDEF, 1)` is done.
`MEMABORT`=*value*	If *value* is nonzero, the equivalent of `mallopt(M_ABORT, 1)` is done.
`MEMLOWFIT`=*value*	If *value* is nonzero, the equivalent of `mallopt(M_LOWFIT, 1)` is done.

RETURN VALUES

The `malloc` and `calloc` functions return a pointer to the allocated space; otherwise, they return a null pointer (with `malloc_error` set).

The `free`, `malloc_limit`, and `malloc_stats` functions return no value.

The `realloc` and `malloc_inplace` functions return a pointer to the allocated space (which may have moved in the case of `realloc`); otherwise, they return a null pointer (with `malloc_error` set).

The `malloc_expand`, `malloc_extend`, and `malloc_howbig` functions return the size of the (possibly expanded) block; otherwise, they return 0 (with `malloc_error` set).

The `malloc_isvalid` function returns nonzero if pointing to a valid block; otherwise, it returns 0.

The `malloc_space` function returns the size of the space available to return to the system; otherwise, it returns 0.

The `malloc_brk` function returns –1 if the heap cannot be extended to the desired address; otherwise, it returns 0.

The `malloc_check` function returns nonzero if the heap is corrupt; otherwise, it returns 0.

The `malloc_tron`, `malloc_troff`, `malloc_etrace`, and `malloc_dtrace` macros return no value.

The `mallopt` function returns –1 if either *cmd* or *value* is invalid; otherwise it returns 0.

The `mallinfo` function returns a `mallinfo` structure, which describes the heap.

EXAMPLES

The following example turns memory tracing on:

```
malloc_tron();
```

The following example disables tracing for all functions but `malloc` and `free`:

```
malloc_dtrace(~(MF_MALLOC|MF_FREE));
```

The following example enables tracing for `realloc`:

```
malloc_etrace(MF_REALLOC);
```

The following example turns memory tracing off:

```
malloc_troff();
```

The following example links C and Fortran programs with `libmalloc`:

```
cc -oprog prog.c -lmalloc
cf77 -oprog prog.f -lmalloc
```

The following example runs programs with memory tracing and checking on:

```
env MEMTRON=1 MEMCHK=3 ./prog
```

SEE ALSO

`malloc.h`(3C)

`brk`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

`malloc.h` – Library header for memory allocation and management functions

IMPLEMENTATION

All Cray Research systems

STANDARDS

AT&T extension

TYPES

The types defined in header `malloc.h` are as follows:

Type	Standards	Description
`struct mallinfo`	AT&T	The structure type that is the type returned by the `mallopt` function. (See the description following this table.)
`size_t`	ISO/ANSI	The unsigned integral type of the result of the `sizeof` operator.

Structure `mallinfo` contains the following members:

```
int arena;        /* total space in arena */
int ordblks;      /* number of ordinary blocks */
int smblks;       /* number of small blocks */
int hblks;        /* number of holding blocks */
int hblkhd;       /* space in holding block headers */
int usmblks;      /* space in small blocks in use */
int fsmblks;      /* space in free small blocks */
int uordblks;     /* space in ordinary blocks in use */
int fordblks;     /* space in free ordinary blocks */
int keepcost;     /* cost of enabling keep option */
```

MACROS

The header file `malloc.h` defines the following macros for use with the `mallopt` function (see `mallopt`(3C) for complete information):

```
M_MXFAST      M_NLBLKS      M_GRAIN       M_KEEP        M_TRACE
M_ETRACE      M_DTRACE      M_LIMIT       M_THRESH      M_BREAKSZ
M_MEMCHK      M_LOWFIT      M_INDEF       M_ABORT
```

The header file `malloc.h` defines the following function-like macros for use with the `libmalloc` debug library (see `malloc`(3C) for complete information):

```
malloc_tron     malloc_troff    malloc_dtrace   malloc_etrace
```

The header file `malloc.h` defines the following macros for use with the `malloc_etrace` and `malloc_dtrace` function-like macros (see `malloc`(3C) for complete information):

```
MF_MALLOC     MF_REALLOC    MF_FREE       MF_EXTEND     MF_INPLACE
MF_EXPAND     MF_HOWBIG     MF_CHECK      MF_ISVALID    MF_SPACE
MF_LIMIT      MF_SBREAK     MF__SBREAK    MF_BRK
```

The header file `malloc.h` defines the following macros, which describe the possible error values for the `malloc_error` variable (see `malloc`(3C) for complete information):

```
ME_EXTEND     ME_BADLEN     ME_NOMEM      ME_BADADDR    ME_ISFREE
ME_NOTBLOCK   ME_CORRUPT    ME_BREAK
```

FUNCTION DECLARATIONS

The following functions are declared in the header file `malloc.h`:

```
calloc          free            malloc          realloc
mallopt         mallinfo        malloc_check    malloc_inplace
malloc_expand   malloc_extend   malloc_howbig   malloc_isvalid
malloc_space    malloc_limit    malloc_stats    malloc_brk
```

OBJECT DECLARATIONS

The following object is declared in the header file `malloc.h`:

```
long malloc_error
```

NOTES

If only ISO/ANSI standard functions are used (i.e. `malloc`, `calloc`, `free`, `realloc`), the header file `stdlib.h` is preferred for use with Cray Standard C.

SEE ALSO

`stdlib.h`(3C)

NAME

`math` – Introduction to math functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The math functions provide various means for computing the results of common mathematical functions applied to specific arguments. The functions available include the common trigonometric and hyperbolic functions, exponential and logarithmic functions, and several others. Many more mathematical functions are available in the UNICOS math and scientific libraries, described in the λ8, and *Scientific Libraries Reference Manual*, Cray Research publication SR–2081.

Function Argument and Return Types

The arguments to and the values returned by these functions are all, with a few noted exceptions, floating types: `float`, `double`, `long double`, and `double complex`. In the presence of the function prototypes in the header file `math.h` or `complex.h`, if arguments of type integral are given where type `double` arguments are required, the compiler automatically promotes them to type `double`.

Double Complex and Long Double Functions

Math functions for `long double` and `double complex` values are provided. See `complex.h`(3C) and the *Cray Standard C Reference Manual*, Cray Research publication SR–2074, for more information.

Domain and Range Checking

For all functions, a *domain* error occurs if an input argument is outside the domain over which the mathematical function is defined. The description of each function describes the valid domain. Similarly, a *range* error occurs if the result of the function cannot be represented by the return type. The behavior of each of these mathematical functions when there is a domain or range error depends on the compilation mode; that is, the command-line options specified. Loops containing calls to these functions are candidates for vectorization, if compiled in extended mode.

When code containing calls to these functions is compiled by the Cray Standard C compiler in extended mode (the default), `errno` is not set on error and the functions do not return to the caller on error. If a domain error occurs, the program aborts with a run-time error. The reasons for this behavior being the default behavior are discussed in the next subsection, Fast Calling Sequence and Vector Functions. When calls to these functions are compiled in extended mode on CRAY T90 series systems with IEEE floating-point arithmetic, `errno` is not set on error. The functions do return to the caller on error; the return value for each function is documented on the corresponding man page in `libm` (see the *Intrinsic Procedures Reference Manual*, Cray Research publication SR–2138, or the online man page).

In strict conformance mode (specified by the cc(1) command-line option -hstdc), each function must execute as if it were a single operation, without generating any externally visible exceptions. This means that for those functions for which a domain or range error is possible, the function arguments are checked before the result is computed. If the value is such that it is outside the valid domain, the result is not computed; instead, errno is set to EDOM. If the value is such that the result of the function would overflow (the magnitude of the result is so large that it cannot be represented in an object of the specified type), the function returns the value of the macro HUGE_VAL with the same sign as the correct value of the function. If the value is such that the result of the function would overflow (that is, the magnitude of the result is so large that it cannot be represented in an object of the specified type), the function returns the value of the macro HUGE_VAL with the same sign as the correct value of the function. If the function returns a float, it returns the value of (+/-)HUGE_VALF. If the function returns a long double, it returns the value of (+/-)HUGE_VALL on Cray MPP systems and on CRAY T90 systems with IEEE arithmetic; on other Cray PVP machines, the function returns HUG_VAL for long double. The value of the macro ERANGE is stored in errno. On CRI systems, if the result underflows (the magnitude of the result is so small that it cannot be represented in an object of the specified type), the function returns 0; the integer expression errno does not acquire the value of the macro ERANGE, although it can in other implementations. (An exception to this rule is the function ldexp.)

In strict conformance mode, it is up to the calling function to set errno to 0 before the call and to check errno after the call to see if an error occurred. (See the prog_diag(3C) man page.)

Fast Calling Sequence and Vector Functions

Unfortunately, checking the arguments (and possibly setting errno) takes considerable time and prevents vectorization. For this reason, the Cray Standard C compiler offers a faster alternative. In extended mode (the default mode), argument checking is not done. The functions assume that the arguments are valid; if they are not, a program on any system (except the CRAY T90 system with IEEE floating-point arithmetic) aborts during computation, either because of a floating-point exception or because a low-level function detects the error. When calls to these functions are compiled in extended mode on CRAY T90 series systems with IEEE floating-point arithmetic, no error checking is done. The functions do return to the caller on a computation error; the return value for each function is documented on the corresponding man page in libm (see the *Intrinsic Procedures Reference Manual*, Cray Research publication SR–2138, or the online man page). In extended mode, if all the arguments are valid, the compiler generates code that uses the call-by-register calling sequence to call many of the standard math functions. Further, if the function call is in a vectorizable loop, the call will be made with vector arguments to vector versions of the functions. To compile in extended mode, do not specify the -hstdc option on the cc(1) command line (that is, extended mode is the default).

The slower mode of execution may be appropriate if you are not sure that the arguments are all valid. If you are sure about the validity of the arguments, the performance gain with the vector versions is significant.

ISO/ANSI Standard Reserved Names

All ISO/ANSI standard functions are reserved external names to the implementation. If you define an external with the same name as an ISO/ANSI library function, the behavior is undefined.

ASSOCIATED HEADERS

<complex.h>
<math.h>

ASSOCIATED FUNCTIONS

Function	Description
acos(3C)	Determines the arccosine of a double value (see asin(3C))
acosl(3C)	Determines the arccosine of a long double value (see asin(3C))
asin(3C)	Determines the arcsine of a double value
asinl(3C)	Determines the arcsine of a long double value (see asin(3C))
atan(3C)	Determines the arctangent of a double value (see asin(3C))
atanl(3C)	Determines the arctangent of a long double value (see asin(3C))
atan2(3C)	Determines the arctangent of a double value *x/y* (see asin(3C))
atan2l(3C)	Determines the arctangent of a long double *x/y* (see asin(3C))
ccos(3C)	Determines the cosine of a double complex value (see sin(3C))
cos(3C)	Determines the cosine of a double value (see sin(3C))
cosl(3C)	Determines the cosine of a long double value (see sin(3C))
csin(3C)	Determines the cosine of a double complex value (see sin(3C))
hypot(3C)	Determines the hypotenuse of a value (see sqrt(3C))
sin(3C)	Determines the sine of a double value
sinl(3C)	Determines the sine of a long double value (see sin(3C))
tan(3C)	Determines the tangent of a double value (see sin(3C))
tanl(3C)	Determines the tangent of a long double value (see sin(3C))

Hyperbolic Functions

Function	Description
cosh(3C)	Determines the hyperbolic cosine of a double value (see sinh(3C))
coshl(3C)	Determines the hyperbolic cosine of a long double value (see sinh(3C))
sinh(3C)	Determines the hyperbolic sine of a double value
sinhl(3C)	Determines the hyperbolic sine of a long double value (see sinh(3C))
tanh(3C)	Determines the hyperbolic tangent of a double value (see sinh(3C))
tanhl(3C)	Determines the hyperbolic tangent of a long double value (see sinh(3C))

Exponential and Logarithmic Functions

Function	Description
cexp(3C)	Determines the exponential for double complex values (see exp(3C))
ldexp(3C)	Multiplies a double floating-point number by an integral power
ldexpl(3C)	Multiplies a long double floating-point number by an integral power of 2 (see frexp(3C))
exp(3C)	Determines the exponential for double values
expl(3C)	Determines the exponential for long double values (see exp(3C))

frexp(3C)	Breaks a double floating-point number into a normalized fraction and an integral power of 2
frexpl(3C)	Breaks a long double floating-point number into a normalized fraction and an integral power of 2 (see frexp(3C))
gamma(3C)	Computes the log gamma function for double values
log(3C)	Determines the logarithm for double values (see exp(3C))
logl(3C)	Determines the logarithm for long double values (see exp(3C))
clog(3C)	Determines the logarithm for double complex values (see exp(3C))
log10(3C)	Determines the base 10 logarithm values for double (see exp(3C))
log10l(3C)	Determines the base 10 logarithm values for long double values (see exp(3C))
modf(3C)	Breaks the double argument *value* into integral and fractional parts (see frexp(3C))
modfl(3C)	Breaks the long double argument *value* into integral and fractional parts (see frexp(3C))
pow(3C)	Raises the specified double value to a given power
powl(3C))	Raises the specified long double value to a given power (see pow(3C))
cpow(3C)	Raises the specified double complex value to a given power (see pow(3C))
sqrt(3C)	Determines the square root of a double value
sqrtl(3C)	Determines the square root of a long double value (see sqrt(3C))
csqrt(3C)	Determines the square root of a double complex value (see sqrt(3C))

Nearest Integer, Absolute Value, and Remainder

Function	Description
ceil(3C)	Provides type double math functions for ceiling (see floor(3C))
ceill(3C)	Provides type long double math functions for ceiling (see floor(3C))
fabs(3C)	Computes the absolute value of a double floating-point number (see floor(3C))
fabsl(3C)	Computes the absolute value of a long double floating-point number (see floor(3C))
cabs(3C)	Computes the absolute value of a double complex floating-point number (see floor(3C))
floor(3C)	Provides type double math functions for floor
floorl(3C)	Provides type long double math functions for floor (see floor(3C))
fmod(3C)	Provides type double math functions for remainder (see floor(3C))
fmodl(3C)	Provides type long double math functions for remainder (see floor(3C))

Bessel Functions

Function	Description
j0(3C), j1(3C), jn(3C), y0(3C), y1(3C), yn(3C)	Return Bessel functions (see bessel(3C))

Statistical Functions

Function	Description
`erf`(3C)	Returns error function
`erfc`(3C)	Returns complementary error function (see `erf`(3C))

Complex Functions

Function	Description
`creal`(3C)	Computes the real part of the `double complex` number *x* (see `cimag`(3C))
`cimag`(3C)	Computes the imaginary part of the `double complex` number *x*
`conj`(3C)	Computes the conjugate of the `double complex` number *x* (see `cimag`(3C))

SEE ALSO

`complex.h`(3C), `prog_diag`(3C), `utilities`(3C) for integer arithmetic functions

Intrinsic Procedures Reference Manual, Cray Research publication SR–2138

Scientific Libraries Reference Manual, Cray Research publication SR–2081

Cray Standard C Reference Manual, Cray Research publication SR–2074, for discussion of complex arithmetic

NAME

math.h – Library header for math functions

IMPLEMENTATION

All Cray Research systems

STANDARDS

ANSI

TYPES

None

MACROS

The macros defined in header math.h are as follows. Unless noted as ISO/ANSI, all items are XPG4 compatible.

Macro	Description
HUGE_VAL, HUGE_VALF, HUGE_VALL	Expands to a positive double expression, not necessarily representable as a float; a positive float expression; and a positive long double expression, respectively. On all systems except CRAY T90 systems with IEEE arithmetic, HUGE_VAL is the same as DBL_MAX, which is the largest double value representable. On CRAY T90 systems with IEEE arithmetic, HUGE_VAL expands to positive infinity. On machines with IEEE arithmetic, HUGE_VAL is also defined, with the same value, in the IEEE floating-point header file, fp.h. ISO/ANSI standard.
HUGE	Symbolic constant whose value is the largest representable double value.
M_E	2.7182818284590452354
M_LOG2E	1.4426950408889634074
M_LOG10E	0.43429448190325182765
M_LN2	0.69314718055994530942
M_LN10	2.30258509299404568402
M_PI_2	Expands to the machine's best representation of *pi*/2 or 1.57079632679489661923.
M_PI_4	Expands to the machine's best representation of *pi*/4 or 0.78539816339744830962.
M_1_PI	Expands to the machine's best representation of 1/*pi* or 0.31830988618379067154.
M_2_PI	Expands to the machine's best representation of 2/*pi* or 0.63661977236758134308.
M_2_SQRTPI	Expands to the machine's best representation of 2/*sqrt pi* or 1.12837916709551257390.
M_SQRT1_2	Expands to the machine's best representation of *sqrt (1/2)* or 0.70710678118654752440.

When compiling in extended mode, header file values.h is included in math.h. Thus, all macros defined in values.h are available in addition to the previous list. See values.h(3C) for more information about these additional macros.

FUNCTION DECLARATIONS

Math functions take `double` or `long double` arguments and return `double` or `long double` values, respectively. Functions declared in header `math.h` are as follows:

acos(3C)	cosf(3C)	floorl(3C)	log(3C)	sinhl(3C)
acosf(3C)	cosh(3C)	fmod(3C)	logf(3C)	sinl(3C)
acosl(3C)	coshf(3C)	fmodf(3C)	log10(3C)	sqrt(3C)
asin(3C)	coshl(3C)	fmodl(3C)	log10f(3C)	sqrtf(3C)
asinf(3C)	cosl(3C)	frexp(3C)	log10l(3C)	sqrtl(3C)
asinl(3C)	erf(3C)	frexpf(3C)	logl(3C)	tan(3C)
atan(3C)	erfc(3C)	frexpl(3C)	modf(3C)	tanf(3C)
atanf(3C)	exp(3C)	gamma(3C)	modff(3C)	tanh(3C)
atan2(3C)	expf(3C)	hypot(3C)	modfl(3C)	tanhf(3C)
atan2l(3C)	expl(3C)	j0(3C)	pow(3C)	tanhl(3C)
atof(3C)	fabs(3C)	j1(3C)	powf(3C)	tanl(3C)
ceil(3C)	fabsf(3C)	jn(3C)	powl(3C)	y0(3C)
ceilf(3C)	fabsl(3C)	ldexp(3C)	sin(3C)	y1(3C)
ceill(3C)	floor(3C)	ldexpf(3C)	sinf(3C)	yn(3C)
cos(3C)	floorf(3C)	ldexpl(3C)	sinh(3C)	

Function `atof`(3C) is declared in header file `math.h` only when compiling in extended mode. See `strtod`(3C) for more information about function `atof`(3C).

CAUTIONS

Some function declarations that were previously in `math.h` and `stdio.h`(3C) are now in `stdlib.h`(3C). Therefore, check to see that your code includes the proper header.

If you have selected strict ANSI conformance mode and, during compilation, the compiler complains about incompatible types for a function and its return value, first check the reference manual to determine if in fact the function is ANSI standard. If it is not and you still want to use it, you will have to explicitly declare it because it will not be declared in the header file you have included.

SEE ALSO

`stdio.h`(3C), `stdlib.h`(3C), `strtod`(3C), `values.h`(3C)

Scientific Libraries Reference Manual, Cray Research publication SR–2081

Intrinsic Procedures Reference Manual, Cray Research publication SR–2138

NAME

mbtowc, mblen, wctomb – Multibyte character handling

SYNOPSIS

```
#include <stdlib.h>
int mbtowc (wchar_t *pwc, const char *s, size_t n);
int mblen (const char *s, size_t n);
int wctomb (char *s, wchar_t wchar);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

If *s* is not a null pointer, the mbtowc function determines the number of bytes that comprise the multibyte character to which *s* points. It then determines the code for the value of type wchar_t that corresponds to that multibyte character. (The value of the code that corresponds to the null character is 0.) If the multibyte character is valid, and pwc is not a null pointer, mbtowc stores the code in the object to which *pwc* points. At most *n* bytes of the array to which *s* points are examined.

If *s* is not a null pointer, the mblen function determines the number of bytes comprising the multibyte character to which *s* points. Except that the shift state of the mbtowc function is not affected, it is equivalent to the following:

```
mbtowc((wchar_t *)0, s, n);
```

The wctomb function determines the number of bytes needed to represent the multibyte character that corresponds to the code that has the value *wchar* (including any change in shift state). It stores the multibyte character representation in the array object to which *s* points (if *s* is not a null pointer). At most MB_CUR_MAX characters are stored. If the value of *wchar* is 0, wctomb is left in the initial shift state.

NOTES

The LC_CTYPE category of the current locale affects the behavior of the multibyte character functions. For a state-dependent encoding, each function is placed into its initial state by a call for which its character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null pointer cause the internal state of the function to be altered as necessary. A call with *s* as a null pointer causes these functions to return a nonzero value if encodings have state dependency; otherwise, these functions return 0. Changing the LC_CTYPE category causes the shift state of these functions to be indeterminate.

RETURN VALUES

If *s* is a null pointer, `mblen` and `mbtowc` return a nonzero value if multibyte character encodings have state-dependent encodings; they return 0 if multibyte character encodings do not have state-dependent encodings.

If *s* is not a null pointer, `mblen` or `mbtowc` return 0 if *s* points to the null character. They return the number of bytes that comprise the multibyte character if the next *n* or fewer bytes form a valid multibyte character. They return –1 if they do not form a valid multibyte character.

The value returned is never greater than *n* or the value of the `MB_CUR_MAX` macro.

If *s* is a null pointer, `wctomb` returns a nonzero value if multibyte character encodings have state-dependent encodings. It returns 0 if multibyte character encodings do not have state-dependent encodings.

If *s* is not a null pointer, `wctomb` returns –1 if the value of *wchar* does not correspond to a valid multibyte character; otherwise, it returns the number of bytes that comprise the multibyte character that correspond to the value of *wchar*.

In the cases in which the preceding functions return –1, they also may set `errno` to the value `EILSEQ` (a byte/character sequence that is not valid is detected).

SEE ALSO

`locale.h`(3C), `mbstring`(3C)

NAME

mbstowcs, wcstombs – Multibyte string functions

SYNOPSIS

```
#include <stdlib.h>
size_t mbstowcs (wchar_t *pwcs, const char *s, size_t n);
size_t wcstombs (char *s, const wchar_t *pwcs, size_t n);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The mbstowcs function converts a sequence of multibyte characters that begins in the initial shift state from the array to which *s* points into a sequence of corresponding codes and stores not more than *n* codes into the array pointed to by *pwcs*. No multibyte characters that follow a null character (which is converted into a code with value 0) are examined or converted. Each multibyte character is converted as if by a call to mbtowc, except that the shift state of the mbtowc function is not affected. If *pwcs* is a null pointer, the mbstowcs function returns the number of elements required for the wide character code array.

No more than *n* elements are modified in the array to which *pwcs* points. If copying occurs between objects that overlap, the behavior is undefined.

The wcstombs function converts a sequence of codes that correspond to multibyte characters from the array to which *pwcs* points into a sequence of multibyte characters that begins in the initial shift state and stores these multibyte characters into the array to which *s* points, stopping if a multibyte character would exceed the limit of *n* total bytes or if a null character is stored. Each code is converted as if by a call to the wctomb(3C) function, except that the shift state of the wctomb function is not affected. If *s* is a null pointer, the wcstombs function returns the number of bytes required for the character array.

No more than *n* bytes are modified in the array to which *s* points. If copying takes place between objects that overlap, the behavior is undefined.

The LC_CTYPE category of the current locale affects the behavior of the multibyte string functions.

RETURN VALUES

If a multibyte character that is not valid is encountered, mbstowcs returns (size_t) –1, and may set errno to EILSEQ (a character sequence that is not valid is detected). Otherwise, mbstowcs returns the number of array elements modified, not including a terminating 0 code, if any.

On encountering a code that does not correspond to a valid multibyte character, `wcstombs` returns `(size_t)-1`, and it may set `errno` to `EILSEQ`. Otherwise, `wcstombs` returns the number of bytes modified, not including a terminating null character, if any.

SEE ALSO

`locale.h`(3C), `mbchar`(3C)

NAME

memchr, memcmp, memcpy, memccpy, memmove, memset – Performs memory operations

SYNOPSIS

```
#include <string.h>
void *memchr(const void *s, int c, size_t n);
int memcmp(const void *s1, const void *s2, size_t n);
void *memcpy(void *s1, const void *s2, size_t n);
void *memccpy(void *s1, const void *s2, int c, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
void *memset(void *s, int c, size_t n);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (except memccpy)
XPG4 (memccpy only)

DESCRIPTION

The memchr function locates the first occurrence of *c* (converted to an unsigned char) in the initial *n* characters (each interpreted as unsigned char) of the object pointed to by *s*.

The memcmp function compares the first *n* characters of the object pointed to by *s1* to the first *n* characters of the object pointed to by *s2*.

The memcpy function copies *n* characters from the object pointed to by *s2* into the object pointed to by *s1*. If copying takes place between objects that overlap, the behavior is undefined.

The memccpy function copies characters from memory area *s2* into *s1*, stopping after the first occurrence of character *c* has been copied, or after *n* characters have been copied, whichever comes first.

The memmove function copies *n* characters from the object pointed to by *s2* into the object pointed to by *s1*. Copying takes place as if the *n* characters from the object pointed to by *s2* are first copied into a temporary array of *n* characters that does not overlap the objects pointed to by *s1* and *s2*, and then the *n* characters from the temporary array are copied into the object pointed to by *s1*.

The memset function copies the value of *c* (converted to an unsigned char) into each of the first *n* characters of the object pointed to by *s*.

RETURN VALUES

The `memchr` function returns a pointer to the located character, or a null pointer if the character does not occur in the object.

The `memcmp` function returns an integer that is greater than, equal to, or less than 0, according to whether the object pointed to by *s1* is greater than, equal to, or less than the object pointed to by *s2*.

The `memcpy` and `memmove` functions return the value of *s1*.

Function `memccpy` returns a pointer to the character after the copy of *c* in *s1*, or a null pointer if *c* was not found in the first *n* characters of *s2*.

The `memset` function returns the value of *s*.

SEE ALSO

`bstring`(3C)

NAME

`memory.h` – Library header for string-handling functions

IMPLEMENTATION

All Cray Research systems

STANDARDS

AT&T extension

DESCRIPTION

Header `memory.h` is identical to header `string.h`; it is included only for compatibility with prior use.

SEE ALSO

`string.h`(3C)

NAME

memwcpy, memwset, memstride, memwchr, memwcmp – Performs word-oriented memory operations

SYNOPSIS

```
#include <string.h>
void *memwcpy(long *s1, long *s2, int n);
long *memwset(long *s, long w, int n [, int str]);
long *memstride(long *s1, int str1, long *s2, int str2, int n);
long *memwchr(long *s, long w, int n [, int str]);
int memwcmp(long *s1, long *s2, int n);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The memwcpy function copies *n* words in the memory area addressed by *s2* to the memory area addressed by *s1*. It handles overlapping moves correctly.

The memwset function sets the first *n* words in memory area *s* to the value of word *w*; it returns *s*. If the optional *str* argument is used, it sets every *str*'th word to the value of *w*.

The memstride function copies *n* words from *s2* (with a stride of *str2*) to *s1* (with a stride of *str1*); it returns *s1*.

The memwchr function returns a pointer to the first occurrence of the word *w* in the first *n* words of *s*; it returns 0 if *w* is not found in the first *n* words of *s*. If the optional *str* argument is used, it compares *w* with every *str*'th word of *s*.

The memwcmp function compares the first *n* words of its arguments; it returns 0 if they are identical, or the index of the first detected difference. The index is one-based.

NOTES

For user convenience, all these functions are declared in the optional <string.h> header file. The memwcpy, memstride, and memwcmp functions are declared as fast in-line functions in <string.h>.

NAME

message – Introduction to UNICOS message system functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

These functions provide means for accessing basic system resources affecting the UNICOS message system.

ASSOCIATED HEADERS

<nl_types.h>

ASSOCIATED FUNCTIONS

Function	Description
catclose(3C)	Closes a message catalog (see catopen(3C))
catgetmsg(3C)	Reads a message from a message catalog
catgets(3C)	Gets message from a message catalog
catmsgfmt(3C)	Formats an error message
catopen(3C)	Opens a message catalog

SEE ALSO

file(3C), multic(3C), password(3C), terminal(3C) (all introductory pages to other operating system service functions)

nl_types(5) for a description of header nl_types.h in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

Cray Message System Programmer's Guide, Cray Research publication SG–2121

NAME

mktemp – Makes a unique file name

SYNOPSIS

```
#include <stdlib.h>
char *mktemp (char *template);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

AT&T extension

DESCRIPTION

The mktemp function replaces the contents of the string to which *template* points with a unique file name, and it returns the address of *template*. The string in *template* should look like a file name with six trailing X's; mktemp replaces the X's with a letter and the current process ID. The letter is chosen so that the resulting name does not duplicate that of an existing file.

NOTES

You may run out of letters. If mktemp cannot create a unique name, it assigns the null string to *template*.

FORTRAN EXTENSIONS

You also can all the mktemp function from Fortran programs, as follows:

```
CHARACTER*n template
INTEGER*8 MKTEMP, I
I = MKTEMP(template)
```

The *template* argument also may be an integer variable. In this case, the data must be packed 8 characters per word and terminated with a null (0) byte.

SEE ALSO

tmpfile(3C), tmpnam(3C)

getpid(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

mktime – Converts local time to calendar time

SYNOPSIS

```
#include <time.h>
time_t mktime (struct tm *timeptr);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The mktime function converts the broken-down time, expressed as local time, in the structure pointed to by *timeptr*, into a calendar time value with the same encoding as that of the values returned by the time(3C) function. The original values of the tm_wday and tm_yday components of the structure are ignored, and the original values of the other components are not restricted to the ranges indicated in the time.h entry. On successful completion, the values of the tm_wday and tm_yday components of the structure are set appropriately, and the other components are set to represent the specified calendar time, but their values are forced to the ranges indicated above; the final value of tm_mday is not set until tm_mon and tm_year are determined.

NOTES

A positive or zero value for tm_isdst causes the mktime function to presume initially that daylight saving time, is (+) or is not (0) in effect for the specified time. A negative value for tm_isdst causes the mktime function to attempt to determine whether daylight saving time is in effect for the specified time.

RETURN VALUES

The mktime function returns the specified calendar time encoded as a value of type time_t. If the calendar time cannot be represented, the function returns the value (time_t)-1.

SEE ALSO

time(3C)

time(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

mldlist – Obtains the list of mandatory access control (MAC) labels currently represented in a multilevel directory

SYNOPSIS

```
#include <sys/types.h>
#include <sys/mac.h>

int mldlist(char *path, mls_t *labels, int count);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The mldlist routine scans the directory structure indicated by the multilevel symbolic link file *path*. If the label list buffer pointer specified by *labels* is a non-null pointer, mldlist merely counts the labels in the multilevel directory and returns that number.

The labels placed in the array to which *labels* points are of the opaque type, mls_t. This data type is created by using mls_create and destroyed by using mls_free. Before discarding the *labels* array, the user should destroy all labels in it by using mls_free.

If *path* is not a multilevel symbolic link, but is a directory or a normal symbolic link to a directory, mldlist returns the label of the directory specified in *path*. This allows mldlist to be used on wildcard directories as well as multilevel directories.

NOTES

The ability to obtain an accurate list of labels depends on MAC access to all subdirectories in the multilevel directory structure. The list of labels returned always represents the total list of labels to which the calling process has MAC read access.

WARNINGS

This routine calls stat(2), lstat(2), and readlink(2), among other system calls. As a result, the routine can sleep or hang if a needed file system resource is unavailable.

RETURN VALUES

If *path* specifies a multilevel symbolic link file, and the target of the symbolic link is an accessible directory, mldlist returns the number of labels represented in the multilevel directory structure.

If *path* is a normal symbolic link to a directory, or *path* is a directory itself, `mldlist` returns a 1 (in this case, the number of labels represented is 1).

If *path* is a not a symbolic link or *path* is not a directory, or access is denied to *path* or its symbolic link target for some reason, `mldlist` returns a –1.

SEE ALSO

`mldname`(3C), `mldwalk`(3C), `mls_create`(3C), `mls_free`(3C), `pathname`(3C)

NAME

mldname – Expands a multilevel symbolic link reference at an arbitrary mandatory access control (MAC) label

SYNOPSIS

```
#include <sys/types.h>
#include <dirent.h>
#include <sys/mac.h>

char *mldname(char *path, mls_t label);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The mldname routine determines the actual path name to which a multilevel symbolic link redirects a path name search at an arbitrary MAC label. The routine takes a path name (*path*) and a MAC label (*label*) as its arguments. If the file to which *path* points is a multilevel symbolic link, mldname computes the labeled subdirectory name based on the MAC label provided in *label* and appends it to the contents of the multilevel symbolic link file. If the file to which *path* points is not a multilevel symbolic link, but is either a directory or a symbolic link to a directory, mldname simply returns the path name *path*.

This routine allocates space for the path name it returns by using malloc and copies the result into that space. It then returns a pointer to the newly composed path name. The allocated space can be freed by using free.

WARNINGS

This routine calls stat(2), lstat(2), and readlink(2), among other system calls. As a result, the routine can sleep or hang if a needed file system resource is unavailable.

RETURN VALUES

This routine returns a pointer to a path name. If an error occurs, it returns a null pointer.

SEE ALSO

free(3C), malloc(3C), mldlist(3C), mldwalk(3C), mls_create(3C), mls_free(3C), opendir(3C), pathname(3C)

NAME

mldwalk – Walks the labeled subdirectories of a multilevel directory (MLD)

SYNOPSIS

```
#include <ftw.h>
#include <sys/stat.h>
#include <sys/secstat.h>
#include <sys/mac.h>

int mldwalk(char *path, int(*fn)(char*, struct stat*, struct secstat*,
int));
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The mldwalk routine traverses the labeled subdirectories of the multilevel directory specified by the multilevel symbolic link file, *path*. For each entry found in a label subdirectory, mldwalk calls the user-supplied function *fn* with a character pointer that points to the full path name of the entry, a stat structure that contains the status of the file corresponding to the entry, a secstat structure that contains the security status of the file corresponding to the entry, and an integer with the following possible values (found in ftw.h):

Value	Description
FTW_F	Nondirectory
FTW_D	Directory
FTW_NS	Object for which stat(2) or secstat(2) system call could not be successfully executed

Unlike the ftw routine, mldwalk does not walk down the directory tree, but walks across the labeled subdirectories and visits each entry in each labeled subdirectory. Therefore, it does not concern itself with the FTW_DNR value (which is a directory that can not be read).

When walking a multilevel directory, mldwalk silently skips all labeled subdirectories it cannot open for reading.

If the file specified in *path* is a directory or a normal symbolic link to a directory rather than a multilevel symbolic link, mldwalk visits each file in the directory and returns.

WARNINGS

This routine calls `stat`(2), `lstat`(2), and `readlink`(2), among other system calls. As a result, the routine can sleep or hang if a needed file system resource is unavailable.

RETURN VALUES

If `mldwalk` exhausts its traversal, it returns a 0. If *fn* returns a nonzero value, `mldwalk` stops its traversal and returns whatever value was returned from *fn*. If an error occurs, `mldwalk` returns –1.

SEE ALSO

`ftw`(3C) `mldlist`(3C), `mldname`(3C), `pathname`(3C)

`secstat`(2), `stat`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

mls_create – Creates an opaque security label structure

SYNOPSIS

```
#include <sys/types.h>
#include <sys/secparm.h>
#include <sys/mac.h>

mls_t mls_create(int level, long comparts);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The mls_create routine allocates and creates an opaque security label for use with the security label comparison routines (for example, mac_equal). *level* is the desired level and *comparts* is the desired compartment bit mask.

RETURN VALUES

On successful completion, the routine allocates space for and returns the desired security label. Otherwise, no space is allocated, a null pointer is returned, and *errno* is set to indicate the error.

ERRORS

mls_create fails if the following error condition occurs:

Error Code	Description
ENOMEM	The label to be returned required more memory than was allowed for the calling process.

SEE ALSO

mls_extract(3C), mls_free(3C)

NAME

mls_dominate – Performs a security label domination test

SYNOPSIS

```
#include <sys/types.h>
#include <sys/secparm.h>
#include <sys/mac.h>

int mls_create(mls_t mac_pl, mls_t mac_p2);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The mls_dominate routine determines whether *mac_p1* dominates *mac_p2*.

NOTES

Dominance includes equivalence. Therefore, if one label equals another, each shall dominate the other. The two labels may not dominate each other (that is, the labels are disjoint).

RETURN VALUES

A value of –1 is returned and *errno* is set to indicate the error. Otherwise, the mls_dominate function returns a 1 if *mac_pl* dominates *mac_p2*, or a 0 if *mac_pl* does not dominate *mac_p2*.

ERRORS

The mls_dominate routine fails if the following error condition occurs:

Error Code	Description
EINVAL	At least one of the labels, *mac_p1* or *mac*_p2, is not a valid security label.

SEE ALSO

mls_create, mls_equal(3C), mls_free(3C), mls_glb(3C), mls_lub(3C)

NAME

mls_equal – Performs a security label equality test

SYNOPSIS

```
#include <sys/types.h>
#include <sys/secparm.h>
#include <sys/mac.h>

int mls_equal(mls_t mac_p1, mls_t mac_p2);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The mls_equal routine determines whether *mac_p1* equals *mac_p2*.

RETURN VALUES

The mls_equal routine returns a 1 if *mac_p1* is equal to *mac_p2*, or 0 if *mac_p1* does not equal *mac_p2*. If an error occurs, a value of –1 is returned, and *errno* is set to indicate the error.

ERRORS

mls_equal fails if the following error condition occurs:

Error Code	Description
EINVAL	At least one of the labels, *mac_p1* or *mac*_p2, is not a valid security label.

SEE ALSO

mls_create(3C), mls_dominate(3C), mls_free(3C), mls_glb(3C), mls_lub(3C)

NAME

mls_export – Converts internal security label to text representation

SYNOPSIS

```
#include <sys/types.h>
#include <sys/secparm.h>
#include <sys/mac.h>

char *mls_export(mls_t mac_p1);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The mls_export routine converts the internal representation of the security label stored in *mac_p1* into its text representation. The routine allocates space for the text representation, copies the text representation into that space, and returns a character pointer to that space.

The format of the text security label is as follows:

level,*compartment*[,*compartment*[...]]

level is the name that represents the appropriate level; *compartment* is the name that represents the appropriate compartment. If the label does not have any compartments specified, the text none is used.

RETURN VALUES

On successful completion, the mls_export routine allocates storage for and returns the text representation of the label. Otherwise, no storage space is allocated, a null pointer is returned, and *errno* is set to indicate the error.

ERRORS

mls_export fails if the following error condition occurs:

Error Code	Description
ENOMEM	The label to be returned required more memory than was allowed for the calling process.

SEE ALSO

mls_import(3C)

NAME

`mls_extract` – Extracts label from an opaque security label structure

SYNOPSIS

```
#include <sys/types.h>
#include <sys/secparm.h>
#include <sys/mac.h>

void mls_extract(mls_t macp1, int *level, long *comparts);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `mls_extract` routine extracts the level and compartment information from the opaque security label specified by *macp1*; the label information is stored in *level* and *comparts*.

RETURN VALUES

The `mls_extract` routine does not return a value.

SEE ALSO

`mls_create`(3C), `mls_free`(3C)

NAME

`mls_free` – Frees security label storage space

SYNOPSIS

```
#include <sys/types.h>
#include <sys/secparm.h>
#include <sys/mac.h>

void mls_free(mls_t mac_p1);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `mls_free` routine frees memory previously allocated by calls made to any security label function that allocates memory on the caller's behalf (for example `mls_create`). The *mac_p1* argument is the security label to be freed.

RETURN VALUES

The `mls_free` function does not return a value.

SEE ALSO

`mls_create`(3C), `mls_extract`(3C)

NAME

mls_glb – Computes the greatest lower bound

SYNOPSIS

```
#include <sys/types.h>
#include <sys/secparm.h>
#include <sys/mac.h>

mls_t mls_glb(mls_t mac_p1, mls_t mac_p2);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The mls_glb routine allocates space for and returns the security label (if it exists) that is dominated by both the labels specified by *mac_p1* and *mac_p2*, and dominates all other security labels that are dominated by both *mac_p1* and *mac_p2*.

RETURN VALUES

On successful completion, this routine allocates space for and returns the newly allocated bounding security label. Otherwise, no space is allocated, a null pointer is returned, and *errno* is set to indicate the error.

ERRORS

mls_glb fails if the following error conditions occur:

Error Code	Description
ENOMEM	The label to be returned required more memory than was allowed for the calling process.
EINVAL	The bounding security label does not exist, or *mac_p1* and/or *mac_p2* has a wildcard label.

SEE ALSO

mls_create(3C), mls_dominate(3C), mls_free(3C), mls_lub(3C)

NAME

mls_import – Converts text security label to internal representation

SYNOPSIS

```
#include <sys/types.h>
#include <sys/secparm.h>
#include <sys/mac.h>

mls_t mls_import(char *text);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The mls_import routine converts the text representation of the security label specified by *text* into its internal representation. When mls_import is called, the routine allocates storage space for the security label, that may be freed with a call to mls_free(3C).

The format of the text security label is as follows:

level[,*compartment*[,*compartment*[...]]]

Where *level* is the name or numeric value that represents the appropriate level; *compartment* is the name or numeric value that represents the appropriate compartment. If no compartments are specified or the text string none or 0 is used, and the compartment bit mask is set to 0.

RETURN VALUES

On successful completion, the mls_import routine allocates storage for and returns the security label. Otherwise, no storage space is allocated, a null pointer is returned, and *errno* is set to indicate the error.

ERRORS

mls_import fails if the following error conditions occur:

Error Code	Description
ENOMEM	The label to be returned required more memory than was allowed for the calling process.
EINVAL	The string *text* is not a valid text representation of a security label.

SEE ALSO

mls_export(3C), mls_extract(3C), mls_free(3C)

NAME

mls_lub – Computes the least upper bound

SYNOPSIS

```
#include <sys/types.h>
#include <sys/secparm.h>
#include <sys/mac.h>

mls_t mls_lub(mls_t mac_p1, mls_t mac_p2);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The mls_lub routine allocates space for and returns the security label (if it exists) that is dominated by both the security labels specified by *mac_p1* and *mac_p2*, and is dominated by all other security labels that dominate both *mac_p1* and *mac_p2*.

RETURN VALUES

On successful completion, this function allocates space for and returns the newly allocated bounding security label. Otherwise, no space is allocated, a null pointer is returned, and *errno* is set to indicate the error.

Error Code	Description
ENOMEM	The label to be returned required more memory than was allowed for the calling process.
EINVAL	The bounding security label does not exist, or *mac_p1* and/or *mac_p2* has a wildcard label.

SEE ALSO

mls_create(3C), mls_dominate(3C), mls_free(3C), mls_glb(3C),

NAME

MTIMESX, MTIMESCN, MTIMESUP – Returns multitasking overlap time

SYNOPSIS

```
CALL MTIMESX(overlap)
ncpus = MTIMESCN()
irtc = MTIMESUP()
```

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

MTIMESX, MTIMESCN, and MTIMESUP all return fields in the structure that is made known to the system by the mtimes(2) system call. The structure contains execution timing information about multitasking programs.

The following is a list of valid arguments:

Argument	Description
ncpus	Integer that specifies the number of physical CPUs connected at that instant.
irtc	Integer real-time clock value when mtimes structure was last updated by the operating system.
rarray	Real array to hold the timing values returned by MTIMESX.
overlap	Address of the real array to hold the overlap time array of the mtimes structure. The length of this array must be the number of CPUs on the target machine. The elements of the array denote how much CPU time was accumulated by the program with a particular number of CPUs executing. For example, overlap(3) denotes the amount of CPU time accumulated when three CPUs were executing in the program, overlap(8) the amount accumulated when eight CPUs were executing in the program, and so on.

MTIMESX fills the array whose address is passed with the overlap time array of the mtimes structure. Summing the elements of the array filled in by MTIMESX yields total execution time for a multitasking program.

MTIMESCN returns the field of the mtimes structure that denotes how many physical CPUs are connected to the program at that point. The contents of the field may change at any time.

MTIMESUP returns the field of the mtimes structure that denotes the real-time clock value at which the mtimes structure was last updated by the operating system. The contents of the field may change at any time.

MTIMESCN and MTIMESUP are useful for getting accurate timings of small multitasking codes. By using them together, as shown in the following example, you can ensure that your program has as many CPUs as it requires and that none of the CPUs you are using is interrupted by the operating system during the execution of the loop.

NOTES

This routine is available on SPARC systems, so that user codes do not need to be rewritten, but it has no effect.

EXAMPLES

```
 886  CONTINUE
      CPUS = MTIMESCN( )
C       if (not all CPUs here) loop
      IF (CPUS. NE. 8) GO TO 886
      BEFOREUP = MTIMESUP( )

C       ...(work to be timed)...

      AFTERUP = MTIMESUP( )
C       if (interrupted) loop
      IF (AFTERUP .NE. BEFOREUP) GO TO 886
```

SEE ALSO

SECOND(3F), TSECND(3F)

mtimes(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

MTTIMES – Prints CPU timing information to stdout

SYNOPSIS

```
CALL  MTTIMES
```

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

The MTTIMES subroutine prints CPU timing information to stdout. MTTIMES can tell you how long multiple CPUs were concurrently active on the job. The average is not a speedup factor, but rather an indication of average overlap. If the amount of time busy-waiting is small relative to the total job time, the average may be close to the actual speedup. This information is the same as that available from the ja(1) command.

The following is a sample of the output from MTTIMES:

```
     CPU Utilization
CPUS |  Time(sec)|   Total
1x   |  0.551=   |  0.551
2x   |  14.203=  |  28.407
3x   |  35.926=  |  107.778
4x   |  3.342=   |  13.368
_____|___________|________
2.78x|  54.022=  |  150.103
```

NOTES

This routine is available on SPARC systems, so that user codes do not need to be rewritten, but it has no effect.

NOTES

ja(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

multic – Introduction to multitasking functions in C

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

These C functions provide means for accessing basic system resources affecting multitasking.

ASSOCIATED HEADERS

<tfork.h>

ASSOCIATED FUNCTIONS

Function	Description
t_exit	Exits a multitasking process
tfork	Creates a multitasking sibling
t_fork	Creates a multitasking sibling (see tfork)
t_gettid	Returns the TID for specified process (see tid)
t_id	Returns the PID of the caller (see tid)
t_lock	Blocks until the lock is free (see tlock)
t_nlock	Sets a nested lock (see tlock)
t_nunlock	Releases a nested lock (see tlock)
t_testlock	Tests the lock and locks it if necessary (see tlock)
t_tid	Returns the TID of the called function (see tid)
t_unlock	Releases the lock (see tlock)

SEE ALSO

file(3C), message(3C), password(3C), terminal(3C), multif(3F), (all introductory pages to other operating system service functions)

UNICOS File Formats and Special Files Reference Manual, Cray Research publication SR–2014, for more complete descriptions of UNICOS header files

CF77 Optimization Guide, Cray Research publication SG–3773

NAME

multif – Introduction to multitasking routines

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

Multitasking routines create and synchronize parallel tasks within programs. The information in this man page describes these routines on Cray PVP systems and SPARC systems.

Macrotasking Environment Variables

The following environment variables are available to tune macrotasked applications on Cray PVP systems. These environment variables control TSKTUNE tuning keywords, which allow you to tune an application for macrotasking without recompiling or relinking your code. For more information about these keywords, see the TSKTUNE man page.

Variable	Description
MP_DBACTIVE	Specifies the number of additional macrotasks that can be readied for execution before an additional logical CPU is acquired; this is called the *activation deadband value*. The value of MP_DBACTIVE can be any positive integer value. The initial value is 0.
MP_DBRELEAS	Specifies the number of logical CPUs retained by the job if there are more CPUs than macrotasks; this is called the *release deadband value*. Any CPUs in excess of this number are released to the system. The initial value is set to 1 less than the number of physical CPUs available on the system. Setting MP_DBRELEAS to less than this value may cause an excessive number of CPUs to be deleted and acquired, and a correspondingly long list of CPUs in the log file. The value of MP_DBRELEAS can be any positive integer value.
MP_MAXCPU	Specifies the maximum number of logical CPUs allowed for macrotasking. The default is the number of physical CPUs on the system. The maximum value is 63.
MP_PRIORITY	Specifies the scheduling priority for macrotasks. Legal values are 0 to 63, 0 being the lowest priority. The default is 31.
MP_STACKSZW	Specifies the initial stack size (in words) for macrotasks.
MP_STACKINW	Specifies the stack increment (in words) for macrotasks.

The MP_STACKSZW variable is supported on SPARC systems. The other environment variables have no effect.

Task Subroutines

The following are the task routines:

Routine	Description
TSKSTART	Initiates a task
TSKTEST	Indicates if a task exists
TSKTUNE	Modifies tuning parameters within the library scheduler

TSKWAIT	Waits for a task to complete execution
TSKVALUE	Retrieves the user identifier specified in the task control array
TSKLIST	Lists the status of each existing task

Lock Routines

Lock routines protect critical regions of code and shared memory. The following are the lock routines:

Routine	Description
LOCKASGN	Identifies an integer variable to be used as a lock
LOCKREL	Releases the identifier assigned to a lock
LOCKON	Sets a lock
LOCKOFF	Clears a lock
LOCKTEST	Returns the state of a lock
NLOCKON	Sets a nested lock and returns control to the calling task
NLOCKOFF	Clears a nested lock and returns control to the calling task
ISELFADD, ICRITADD	Perform *ivar=ivar+ivalue* under protection of ISELFADD hardware semaphore
ISELFMUL, ICRITMUL	Perform *ivar=ivar*ivalue* under protection of hardware semaphore
ISELFSCH	Performs *ivar=ivar*+1 under protection of hardware semaphore
XSELFADD, XCRIADD	Performs *xvar=xvar+xvalue* under protection of hardware semaphore
XSELFMUL, XCRITMUL	Perform *xvar=xvar+xvalue* under protection of hardware semaphore

Event Routines

Event routines signal and synchronize between tasks. The following are the event routines:

Routine	Description
EVASGN	Identifies a variable to be used as an event
EVCLEAR	Clears an event
EVREL	Releases the identifier assigned to a task
EVPOST	Posts an event
EVTEST	Returns the state of an event
EVWAIT	Suspends task execution until an event is posted

Multitasking History Trace Buffer Routines

The user-level routines for the multitasking history trace buffer can be called from a user program to control what is recorded in the buffer and to dump the contents of the buffer to a file. The following are the multitasking history trace buffer routines:

Routine	Description
BUFTUNE	Modifies parameters used to control which multitasking actions are recorded in the history trace buffer
BUFPRINT	Writes a formatted dump of the history trace buffer to a dataset
BUFDUMP	Writes an unformatted dump of the history trace buffer to a file

BUFUSER	Adds entries to the history trace buffer

These routines are present on SPARC systems, but they have no effect.

Barrier Routines

A barrier is a synchronization point in an application, beyond which no task will proceed until a specified number of tasks have reached the barrier. The following are the barrier routines:

Routine	Description
BARASGN	Identifies an integer variable to use as a barrier
BARREL	Releases the identifier assigned to a barrier
BARSYNC	Registers the arrival of a task at the barrier

Timing Routines

Timing routines return a variety of timing information that is helpful when evaluating and tuning multitasked programs. The following are the timing routines:

Routine	Description
MTIMESX, MTIMESUP, MTIMESCN	Return multitasked overlap time
TSECND	Returns user CPU time in seconds
MTTIMES	Prints CPU timing information

The TSECND routine is available for use on SPARC systems. The other routines are available on SPARC platforms so that user codes do not need to be rewritten, but they have no effect.

NAME

ndbm, dbm_open, dbm_close, dbm_fetch, dbm_store, dbm_delete, dbm_firstkey, dbm_nextkey, dbm_error, dbm_clearerr – Database subroutines

SYNOPSIS

```
#include <ndbm.h>

typedef struct {
    void *dptr;
    size_t desize;
} datum;

DBM *dbm_open(const char *file, int flags, mode_t mode);

void dbm_close(DBM *db);

datum dbm_fetch(DBM *db, datum key);

int dbm_store(DBM *db, datum key);

int dbm_store(DBM *db, datum key, datum content, int flags);

int dbm_delete(DBM *db, datum key);

datum dbm_firstkey(DBM *db);

datum dbm_nextkey(DBM *db);

int dbm_error(DBM *db);

int dbm_error(DBM *db);

int dbm_clearerr(DBM *db);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

These functions maintain key/content pairs in a database. The ndbm functions will handle very large (a billion blocks) databases and will access a keyed item in one or two file system accesses. This package supercedes the dbm(3c) library, which managed only a single database.

Keys and contents are described by the datum typedef:

```
typedef struct {
  void *dptr;
  size_t dsize;
} datum;
```

A datum specifies data of dsize bytes pointed to by `dptr`. Arbitrary binary data, as well as normal ASCII strings, are allowed.

The database is stored in two files. One file is a directory that contains a bit map and has `.dir` as its suffix. The second file contains all data and has `.pag` as its suffix.

Before a database can be accessed, it must be opened by `dbm_open`. This will open and/or create the `file.dir` and `file.pag` files, depending on the flags parameter (see the `open`(2) man page). The mode argument is the same as the third argument in `open`(2).

Once open, the data stored under a key is accessed by `dbm_fetch` and data is placed under a key by `dbm_store`. The flags field can be either `DBM_INSERT` or `DBM_REPLACE`. `DBM_INSERT` will insert only new entries into the database and will not change an existing entry with the same key. `DBM_REPLACE` will replace an existing entry if it has the same key. A key (and its associated contents) is deleted by `dbm_delete`. A linear pass through all keys in a a database may be made in a random order by use of `dbm_firstkey` and `dbm_nextkey`. `dbm_firstkey` will return the first key in the database. `dbm_nextkey` will return the next key in the database. The following code will traverse the database:

```
for (key = dbm_firstkey(db);
     key.dptr != NULL;
     key = dbm_nextkey(db))
```

`dbm_error` returns nonzero when an error has occurred while reading or writing the database. `dbm_clearerr` resets the error condition on the named database.

NOTES

The `.pag` file is designed to contain holes in files. Holes will make this file appear larger than its actual contents.

`dptr` pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 1024 bytes). All key/content pairs that hash together must fit on a single block. `dbm_store` will return an error in the event that a disk block fills with inseparable data.

`dbm_delete` does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by `dbm_firstkey()` and `dbm_nextkey()` rely on the `ndbm` hashing function.

The ndbm functions, except dbm_error, provide interlocks per database to provide a level of thread safe use. Although concurrent updating and reading of a database may lead to unpredictable or unexpected behavior.

RETURN VALUES

All functions that return an int indicate errors with negative values. A zero return indicates that there are no errors. Routines that return a datum indicate errors with a null (0) dptr. If dbm_store called with a flags value of DBM_INSERT and finds an existing entry with the same key, it returns 1.

SEE ALSO

dbm(3C)

NAME

network – Introduction to the network access functions

IMPLEMENTATION

All Cray PVP systems

DESCRIPTION

This subsection describes the functions that make up the network library, the UNICOS network information service facility (NIS), the remote procedure call (RPC) library, and the external data representation (XDR) library.

Associated Headers

Some of the following header files are documented in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014:

```
<sys/socket.h>
<sys/types.h>
<rpcsvc/ypclnt.h>
<netdb.h>
<netinet/in.h>
<net/route.h>
<net/if.h>
```

Associated Functions

Function	Description
authkerb_getucred	Converts a Kerberos encryption-based authentication received in a Remote Procedure Call (RPC) request into a traditional UNIX-style authentication (see kerberos_rpc(3C)). Server routine.
authkerb_seccreate	Returns an authentication handle that enables the use of the Kerberos authentication system (see kerberos_rpc(3C)). Client routine.
bindresvport	Binds a socket to a prvileged IP port
dn_comp	Domain name service resolver functions (see resolver)
dn_expand	Domain name service resolver functions (see resolver)
dn_skipname	Domain name service resolver functions (see resolver)
endhostent	Closes /etc/hosts file (see gethost)
endnetent	Closes /etc/networks file (see getnet)
endnetinfo	Closes /etc/networks file (see getnetinfo)
endprotoent	Closes /etc/protocols file (see getprot)
endrpcent	Closes /etc/rpc file (see getrpcent)
endservent	Closes /etc/services file (see getserv)
endtosent	Closes /etc/iptos file (see gettos)
getdomainname	Gets or sets name of current domain (see getdomain)
gethostbyaddr	Searches for host address (see gethost)

gethostbyname	Searches for host name (see gethost)
gethostent	Gets network host entry (see gethost)
gethostinfo	Gets network host and service entry by host name or host address
gethostinfo	Gets network host and service entry by host name or host address
gethostlookup	Gets network host entry (see gethost)
getnetbyaddr	Searches for network entry by address (see getnet)
getnetbyname	Searches for network entry by name (see getnet)
getnetent	Gets network entry (see getnet)
getnetibyaddr	Searches for network entry by address (see getnetinfo)
getnetibyname	Searches for network entry by name (see getnetinfo)
getnetinfo	Gets network entry
getprotobyname	Searches for protocol name (see getprot)
getprotobynumber	Searches for protocol number (see getprot)
getprotoent	Reads entry in /etc/protocol file (see getprot)
getrpcbyname	Gets remote procedure call entry
getrpcbynumber	Gets remote procedure call entry
getrpcent	Gets remote procedure call entry
getservbyname	Searches for service name (see getserv)
getservbyport	Searches for port number (see getserv)
getservent	Gets service entry (see getserv)
gettosbyname	Searches for Type Of Service name (see gettos)
gettosent	Reads next entry in Type Of Service database (see gettos)
herror	Produces host lookup error messages
hostalias	Domain name service resolver functions (see resolver)
htonl	Converts values between host and network byte order (see byteorder)
htonl	Converts values between host and network byte order (see byteorder)
htons	Converts values between host and network byte order (see byteorder)
htons	Converts values between host and network byte order (see byteorder)
inet_addr	Interprets dot notation and returns Internet address (see inet)
inet_addr	Interprets dot notation and returns Internet address (see inet)
inet_lnaof	Separates Internet host addresses and returns local network address (see inet)
inet_lnaof	Separates Internet host addresses and returns local network address (see inet)
inet_makeaddr	Constructs Internet address (see inet)
inet_netof	Separates Internet host addresses and returns network number (see inet)
inet_netof	Separates Internet host addresses and returns network number (see inet)
inet_network	Interprets dot notation and returns Internet number (see inet)
inet_network	Interprets dot notation and returns Internet number (see inet)
inet_ntoa	Interprets Internet address and returns dot notation (see inet)
inet_ntoa	Interprets Internet address and returns dot notation (see inet)
inet_subnetmaskof	Returns subnet of the Internet address (see inet)
inet_subnetof	Returns subnet mask of the Internet address (see inet)

iso_addr	Manipulates ISO/OSI address
iso_ntoa	Manipulates ISO/OSI address (see iso_addr)
kerberos_rpc(3C)	Make remote procedure calls using Kerberos authentication.
nqeapi(3)	General Network Queuing Environment (NQE) functions for formatted lists
nqe_get_policy_list(3)	Queries the Network Load Balancer(NLB) to retrieve a formatted list of hosts that match a specified policy
nqe_get_request_ids(3)	Returns a list of all Network Queuing System (NQS) request IDs known to a specified NLB server
nqe_get_request_info(3)	Returns a list of all information known about a specific NQS request ID from a specified NLB server
ntohl	Converts values between host and network byte order (see byteorder)
ntohl	Converts values between host and network byte order (see byteorder)
ntohs	Converts values between host and network byte order (see byteorder)
ntohs	Converts values between host and network byte order (see byteorder)
parsetos	Gets network Type Of Service information (see gettos)
rcmd	Returns a stream to a remote command
rcmdexec	Returns a stream to a remote command
res_init	Domain name service resolver functions (see resolver)
res_mkquery	Domain name service resolver functions (see resolver)
res_query	Domain name service resolver functions (see resolver)
res_querydomain	Domain name service resolver functions (see resolver)
res_search	Domain name service resolver functions (see resolver)
res_send	Domain name service resolver functions (see resolver)
resolver	Domain name service resolver functions
rexec	Returns a stream to a remote command
rpc	Makes a remote procedure call
rresvport	Returns a descriptor to a socket (see rcmd)
ruserok	Authenticates remote users (see rcmd)
setdomainname	Gets or sets name of current domain (see getdomain)
sethostent	Opens/rewinds /etc/hosts file (see gethost)
setnetent	Opens/rewinds /etc/networks file (see getnet)
setnetinfo	Opens/rewinds /etc/networks file (see getnetinfo)
setprotoent	Opens/rewinds /etc/protocols file (see getprot)
setrpcent	Gets remote procedure call entry
setservent	Opens/rewinds /etc/services file (see getserv)
settosent	Opens/rewinds /etc/iptos file (see gettos)
svc_kerb_reg	Performs registration tasks that are required before Kerberos encryption-based authentication requests are processed (see kerberos_rpc(3C)). Server routine.
xdr	Achieves machine-independent data transformation
yp_all	Network information service (NIS) client interface (see ypclnt)
yp_bind	NIS client interface (see ypclnt)
ypclnt	NIS client interface

yperr_string	NIS client interface (see ypclnt)
yperr_string	NIS client interface (see ypclnt)
yp_first	NIS client interface (see ypclnt)
yp_get_default_domain	NIS client interface (see ypclnt)
yp_get_default_domain	NIS client interface (see ypclnt)
yp_master	NIS client interface (see ypclnt)
yp_match	NIS client interface (see ypclnt)
yp_next	NIS client interface (see ypclnt)
yp_order	NIS client interface (see ypclnt)
ypprot_err	NIS client interface (see ypclnt)
yp_unbind	NIS client interface (see ypclnt)

SEE ALSO

errno.h(3C), intro(3C), perror(3C)

dup(2), intro(2), open(2), pipe(2), read(2), recv(2), send(2), socket(2), write(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

hosts(5), icmp(4P), inet(4P), ip(4P), iptos(5), networks(5), protocols(5), services(5), tcp(4P), udp(4P) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

nextafter, nextafterf, nextafterl – Returns the next value in the direction of the second argument

SYNOPSIS

CRAY T90 systems with IEEE hardware:

```
#include <fp.h>

double nextafter (double x, double y);
float nextafterf (float x, float y);
long double nextafterl (long double x, long double y);
```

Cray MPP systems:

```
#include <fp.h>

double nextafter (double x, double y);
```

IMPLEMENTATION

Cray MPP systems (implemented as a macro)
CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

ANSI/IEEE Std 754-1985
X3/TR-17:199x

DESCRIPTION

The nextafter function and macro and the nextafterf and nextafterl functions determine the next representable value, in the type of the function, after *x* in the direction of *y*. If *x*==*y*, the functions return *y*.

RETURN VALUES

These functions return the next representable value after *x* in the direction of *y*.

It is sometimes desirable to find the next representation after a value in the direction of a previously computed value, either smaller or larger. The nextafter functions have a second floating-point argument so that the program will not have to include floating-point tests for determining the direction in such situations.

For the case *x*==*y*, the IEEE standard recommends that *x* be returned. This specification differs so that nextafter(-0.0,+0.0) returns +0.0, and nextafter(+0.0,-0.0) returns -0.0.

SEE ALSO

Migrating to the CRAY T90 Series IEEE Floating Point, Cray Research publication SN–2194

NAME

`nlimit` – Provides an interface to setting or obtaining resource limit values

SYNOPSIS

```
#include <errno.h>
#include <sys/category.h>
#include <sys/resource.h>

int nlimit (int id, struct resclim *rptr);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The `nlimit` library interface provides a means to establish or view resource limit information from the kernel based on the following arguments:

Argument	Description
id	The *pid*, *sid*, or *uid* corresponding to the `resclim` field `resc_category`. A 0 indicates the current *pid*, *sid*, or *uid*.
rptr	A pointer to the `resclim` structure. The `resclim` structure to which *rptr* points includes the following members (for a complete list, see `/usr/include/sys/resource.h`):

```
struct resclim {
  int   resc_resource;  /* One of:  L_CPU                                  */
  int   resc_category;  /* One of:  C_PROC, C_SESS, C_UID, C_SESSPROCS  */
  int   resc_type;      /* One of:  L_T_HARD, L_T_SOFT    */
  int   resc_action;    /* One of:  L_A_TERMINATE, L_A_CHECKPOINT       */
  int   resc_used;      /* Current amount of resource used              */
  int   resc_value[R_NLIMTYPES];
                        /* Current resource limit value for */
                        /* L_T_HARD AND L_T_SOFT            */
};
```

The `resclim` structure contains fields used to establish or view resource limits. The `nlimit` function sets up the structure according to the parameters passed to it and calls `setlim(2)` to change limit values, or `getlim(2)` to obtain information about limit values.

Obtaining Resource Limits

You must set the `resclim` structure fields `resc_resource`, `resc_category`, and `resc_type` to return limit values. The `resc_resource` field represents the resource to be queried. Currently, only CPU resources are supported; therefore, the value of `resc_resource` must be `L_CPU`. The `resc_category` identifies the category of resource that will be queried. Acceptable values are `C_PROC`, `C_SESS`, `C_UID`, and `C_SESSPROCS`, as follows:

Value	Description
`C_PROC`	Returns process limits
`C_SESS`	Returns session limits
`C_UID`	Returns user limits
`C_SESSPROCS`	Returns default process limits for the session

The `resclim` field `resc_category` determines whether the *id* argument is a *pid*, *sid*, or *uid*. The `resc_category` of `C_SESSPROCS` requires a *sid*.

You must set the `resc_type` field to `NULL` to return its limits.

If the call succeeds, `nlimit` fills in the missing information in the `resclim` structure, including the following fields:

Field	Description
`resc_action`	Returns a value of `L_A_TERMINATE` or `L_A_CHECKPOINT`. When a hard limit is reached, this value determines whether the process is checkpointed before termination.
`resc_used`	Returns the amount of resource currently accumulated at the time of the call. For `L_CPU`, this value is the amount of CPU seconds accumulated.
`resc_value[R_NLIMTYPES]`	Returns two values. The `resc_value[L_T_HARD]` field is the hard resource limit, and `resc_value[L_T_SOFT]` is the soft resource limit.

Setting Resource Limits

To set a limit value, all `resclim` fields must be set to either a value or a null. To set a value to be unlimited, use `CPUUNLIM`. To set a value to be null, use `NULL`.

The following list describes each of the fields in the `resclim` structure and their acceptable values:

Field	Description
`resc_resource`	Represents the resource for which a limit will be established. Currently, only CPU resources are supported; therefore, the value of `resc_resource` must be `L_CPU`.
`resc_category`	Identifies the category of resource that will be set. The `resc_category` determines whether the *id* argument is a *pid*, *sid*, or *uid*. Acceptable values are `C_PROC`, `C_SESS`, `C_UID`, and `C_SESSPROCS`. The `resc_category` of `C_SESSPROCS` requires a *sid*. A short description follows:

`C_PROC`	Sets process limits
`C_SESS`	Sets session limits
`C_UID`	Sets user limits
`C_SESSPROCS`	Sets default process limits for the session

`resc_type` Identifies the type of limit that will be set. Acceptable values are: `L_T_HARD` and `L_T_SOFT`.

`resc_action` When a hard limit is reached, this value determines whether the process is checkpointed before termination. Acceptable values are `NULL`, `L_A_TERMINATE`, or `L_A_CHECKPOINT`. If you set the `resc_action` field to `L_A_TERMINATE` or `L_A_CHECKPOINT`, the `resc_type` must be `L_T_HARD`.

`resc_used` Not used with the `nlimit` call when setting limits; the only acceptable value is `NULL`.

`resc_value[R_NLIMTYPES]`
To set hard limits, set the field `resc_type` to `L_T_HARD` and place a value in `resc_value[L_T_HARD]`. To set soft limits, set the field `resc_type` to `L_T_SOFT` and place a value in `resc_value[L_T_SOFT]`. The values in `resc_value[R_NLIMTYPES]` for `resc_resource L_CPU` must be in seconds. You can set only one of `resc_value[L_T_HARD]` or `resc_value[L_T_SOFT]` with each `nlimit` call.

The `nlimit` function fails and no information is updated in the `resclim` structure or no resource limits are set if one or more of the following error conditions occur:

Error Code	Description
EFAULT	The address specified for *rptr* was invalid.
EINVAL	One of the arguments contains a value that is not valid.
EPERM	The user ID of the requesting process is not that of a super user.
EPERM	An attempt was made to change a limit on a system process; this is not allowed.
ESRCH	No processes were found that matched the request.

NOTES

A functional description of `nlimit` is in `nlimit`(1).

RETURN VALUES

On successful completion, a value of 0 indicates that the call succeeded, and the `resclim` structure was filled in with appropriate returned values. Otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

EXAMPLES

The following example shows the execution of the `nlimit` function:

```
#include <stdio.h>
#include <errno.h>
#include <sys/resource.h>
#include <sys/category.h>

   ...

      int retn;
```

```
   struct resclim r;
   struct resclim *rptr;
   rptr = &r;

...

   /*
    *  Set up fields to return current process limits
    */
   rptr->resc_resource        = L_CPU;
   rptr->resc_category        = C_PROC;
   rptr->resc_type            = NULL;
   retn = nlimit(0, rptr);
   if (retn == -1) {
         fprintf(stderr,"nlimit failed with errno %d\n",errno);
   }

...

   /*
    *  Set current process hard limit to 400 seconds and the hard action
    *  to checkpoint.
    */
   rptr->resc_resource        = L_CPU;
   rptr->resc_category        = C_PROC;
   rptr->resc_type            = L_T_HARD;
   rptr->resc_action          = L_A_CHECKPOINT;
   rptr->resc_value[L_T_HARD] = 400;
   retn = nlimit(0, rptr);
   if (retn == -1) {
         fprintf(stderr,"nlimit failed with errno %d\n",errno);
   }
```

SEE ALSO

`nlimit`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`getlim`(2), `setlim`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`NLIMIT`(3F) in the

NAME

`nlist` – Gets entries from name list

SYNOPSIS

```
#include <nlist.h>
int nlist (char *filename, struct nlist *nl);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

AT&T extension

DESCRIPTION

The `nlist` function examines the name list in the executable file whose name is pointed to by *filename*, selectively extracts a list of values, and puts them in the array of `nlist` structures to which *nl* points. The name list *nl* consists of an array of structures that contains names of variables, types, and values. The list is terminated with a null name; that is, a null pointer is in the name position of the structure. Each variable name is looked up in the name list of the file. If the name is found, the remaining fields in its `nlist` are filled with the corresponding values from the symbol table. If the name is not found, the fields are set to 0. For a discussion of the symbol table structure, see `relo`(5).

This function is useful for examining the system name list kept in the `/unicos` file. In this way, programs can obtain system addresses that are current.

NOTES

The `nlist` structure in header `nlist.h` does not correspond exactly to the actual symbol table structure (compare the `nlist` structure with the `gse` structure in header `symbol.h`). The `nlist` structure is used primarily for convenience and compatibility.

RETURN VALUES

If the file cannot be read or if it does not contain a valid name list, all type entries are set to `0`.

On error, `nlist` returns –1; otherwise, it returns 0.

SEE ALSO

`relo`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

nl_langinfo – Points to language information

SYNOPSIS

```
#include <langinfo.h>
char *nl_langinfo (nl_item item);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The nl_langinfo function returns a pointer to a string containing information relevant to the particular language or cultural area defined in the program's locale. The manifest constant names and values of the *item* argument are defined in the langinfo.h header file. For example, the following returns a pointer to the string Dom if the identified language is Portuguese, and Sun if the identified language is English:

```
nl_langinfo (ABDAY_1)
```

Following are the currently defined constants. Unless otherwise noted, all are in the LC_TIME category.

Constant	Meaning
CODESET	Codeset name. LC_TIME category.
D_T_FMT	String for formatting date and time
D_FMT	Date format string
T_FMT	Time format string
T_FMT_AMPM	a.m. or p.m. time format string
AM_STR	Ante meridian affix
PM_STR	Post meridian affix
DAY_1	Name of the first day of the week (for example, Sunday)
DAY_2	Name of the second day of the week
DAY_3	Name of the third day of the week
DAY_4	Name of the fourth day of the week
DAY_5	Name of the fifth day of the week
DAY_6	Name of the sixth day of the week
DAY_7	Name of the seventh day of the week
ABDAY_1	Abbreviated name of the first day of the week
ABDAY_2	Abbreviated name of the second day of the week
ABDAY_3	Abbreviated name of the third day of the week
ABDAY_4	Abbreviated name of the fourth day of the week

ABDAY_5	Abbreviated name of the fifth day of the week
ABDAY_6	Abbreviated name of the sixth day of the week
ABDAY_7	Abbreviated name of the seventh day of the week
MON_1	Name of the first month of the year
MON_2	Name of the second month
MON_3	Name of the third month
MON_4	Name of the fourth month
MON_5	Name of the fifth month
MON_6	Name of the sixth month
MON_7	Name of the seventh month
MON_8	Name of the eighth month
MON_9	Name of the ninth month
MON_10	Name of the tenth month
MON_11	Name of the eleventh month
MON_12	Name of the twelfth month
ABMON_1	Abbreviated name of the first month
ABMON_2	Abbreviated name of the second month
ABMON_3	Abbreviated name of the third month
ABMON_4	Abbreviated name of the fourth month
ABMON_5	Abbreviated name of the fifth month
ABMON_6	Abbreviated name of the sixth month
ABMON_7	Abbreviated name of the seventh month
ABMON_8	Abbreviated name of the eighth month
ABMON_9	Abbreviated name of the ninth month
ABMON_10	Abbreviated name of the tenth month
ABMON_11	Abbreviated name of the eleventh month
ABMON_12	Abbreviated name of the twelfth month
ERA	Era description segment
ERA_D_FMT	Era date format string
ERA_D_T_FMT	Era date and time format string
ERA_T_FMT	Era time format string
ALT_DIGITS	Alternative symbols for digits
RADIXCHAR	Radix character. LC_NUMERIC category.
THOUSEP	Separator for thousands. LC_NUMERIC category.
YESEXPR	Affirmative response expression. LC_MESSAGES category.
NOEXPR	Negative response expression. LC_MESSAGES category.
YESSTR	Affirmative response for yes/no queries. LC_MESSAGES category.
NOSTR	Negative response for yes/no queries. LC_MESSAGES category.
CRNCYSTR	Currency symbol, preceded by – if the symbol should appear before the value, + if the symbol should appear after the value, or . if the symbol should replace the radix character. LC_MONETARY category.

RETURN VALUES

In a locale where language information data is not defined, `nl_langinfo` returns a pointer to the corresponding string in the POSIX locale. In all locales, `nl_langinfo` returns a pointer to an empty string if *item* contains a setting that is not valid.

SEE ALSO

`localeconv`(3C), `setlocale`(3C)

`nl_types.h`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

NLOCKOFF – Clears a nested lock and returns control to the calling task

SYNOPSIS

CALL NLOCKOFF(*name*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

NLOCKOFF clears a nested lock and returns control to the calling task. When NLOCKOFF is called, the nesting level is decremented. If this is the last active nest level, the task clears the lock. Clearing the lock may allow another task to resume execution, but this is transparent to the task calling NLOCKOFF. NLOCKOFF must always be called to clear a lock that has been set by NLOCKON(3F).

The following is a valid argument for this routine:

Argument	Description
name	Name of a 2-word integer array; the first word is the lock, and the second word is the nesting level.

SEE ALSO

NLOCKON(3F)

NAME

NLOCKON – Sets a nested lock and returns control to the calling task

SYNOPSIS

CALL NLOCKON(*name*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

NLOCKON sets a nested lock and returns control to the calling task. If the lock is already set when NLOCKON is called, the task ID is checked. If this task already holds the lock, the nesting level is incremented, and control returns to the calling task. If this task does not hold the lock, the task is suspended until another task clears the lock. This task then sets the lock when it next resumes execution of user code. This means that placing NLOCKON before a critical region ensures that the code in the region is executed only when the task has unique access to the lock. NLOCKON should be used instead of LOCKON(3F) for codes where critical regions may be nested, usually across subroutine boundaries. NLOCKOFF(3F) must always be called to clear a lock that has been set by the NLOCKON routine.

The following is a valid argument for this routine:

Argument	Description
name	Name of a 2-word integer array; the first word is the lock, and the second word is the nesting level.

CAUTIONS

The LOCKTEST(3F) routine cannot be used on nested locks. The same lock cannot be used for calls to the LOCKON(3F) and NLOCKON(3F) routines. These situations result in job aborts.

EXAMPLES

```
      PROOGRAM MULTI
      INTEGER LOCKWD(2)
      INTEGER REALDATA(1000)
      COMMON /MULTITST/ LOCKWD, REALDATA
C     ...
      CALL LOCKASGN(LOCKWD)
C     ...
      CALL NLOCKON(LOCKWD)
      DO 100 I=1,1000
        IF(REALDATA(I).GE.0)
          CALL FIXIT(I)
        ELSE
          REALDATA(I)=0
        ENDIF
 100  CONTINUE
      CALL NLOCKOFF(LOCKWD)
C     ...
      DO 200 I=1,1000
       CALL FIXIT(I)
 200  CONTINUE
C     ...
      END

      SUBROUTINE FIXIT(X)
      INTEGER X
      CALL NLOCKON(LOCKWD)
      REALDATA(X)=MAX(REALDATA(X),50)
      CALL NLOCKOFF(LOCKWD)
      RETURN
      END
```

SEE ALSO

LOCKON(3F), NLOCKOFF(3F)

NAME

numeric_lim – Introduction to numerical limits headers

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The numerical limits headers provide simple macros that expand to numerical limits and parameters, many of which are machine-specific values. Many of the macros specify maximum and minimum values for data types. Using these macros gives you the correct values. You do not need to know the specific value. Use of these macros also greatly increases the portability of your program.

ASSOCIATED HEADERS

<limits.h>
<float.h>
<values.h>

ASSOCIATED FUNCTIONS

Functions that use the values in these headers are primarily in the Mathematics and the General Utility sections. See math(3C) and utilities(3C) for a list of functions.

SEE ALSO

math(3C), utilities(3C)

NAME

_pack, _unpack – Packs or unpacks 8-bit bytes to/from Cray 64-bit words

SYNOPSIS

```
#include <stdlib.h>
long  _pack (const long *up, char *cp, long bc, long tc);
long _unpack (const char *cp, long *up, long bc, long tc);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

These vectorized functions pack or unpack 8-bit bytes to/from Cray 64-bit words. They can be used, for example, to pack lines from a line buffer to a packed buffer, or unpack lines from a packed buffer to a line buffer. A line buffer contains one byte per word, and a packed buffer contains 8 bytes per word. Arguments are as follows:

up Pointer to unpacked data. When packing, bytes are retrieved from this buffer, one right-justified byte per word. When unpacking, bytes are placed in this buffer, one right-justified byte per word.

cp Pointer to packed data. When packing, bytes are placed in this buffer, 8 bytes per word. When unpacking, bytes are retrieved from this buffer, 8 bytes per word. Packing and unpacking need not start or end on a word boundary.

bc Byte count. When packing, this is the number of bytes to pack from *up* to *cp*, excluding a termination character, if specified. When unpacking, this is the maximum number of bytes to unpack from *cp* to *up*. A termination character, if specified and if encountered, terminates unpacking before the byte count is exhausted.

tc Termination character (an integer). Integer value corresponding to a termination character that will terminate unpacking or which will be appended to the end of the packed bytes. This is an optional parameter. If it is omitted or if its value is –1, unpacking will be terminated only after *bc* bytes are unpacked and packing will not append any characters.

RETURN VALUES

No processing takes place and a –1 is returned if any of the following conditions are true:

- *bc* is less than 0.
- *tc* is invalid (it must be in the range 0 through UCHAR_MAX or 1).

- The routine is called with fewer than three arguments.

If the preceding conditions are not true, processing takes place and the number of bytes packed or unpacked is returned. When unpacking, the termination character, if specified and if encountered, is not unpacked nor is it counted as a unpacked byte. When packing, the termination character, if specified, is packed and is counted as a packed byte.

EXAMPLES

Example 1: This example unpacks bytes from `char_buffer` to `line_buffer`. The `_unpack()` routine unpacks 80 bytes or until a new-line character (`'0`) is encountered, whichever occurs first. The new-line character, if it is encountered, will not be unpacked. The variable `nb` will contain the number of bytes actually unpacked

```
int  nb;
long *line_buffer;
unsigned char *char_buffer;

nb = _unpack(char_buffer, line_buffer, 80, '\n');
```

Example 2: This example packs 80 bytes from `words` to `chars`. No termination character is appended to the end of the bytes.

```
int  nb;
long words[80];
unsigned char chars[80];

nb = _pack(words, chars, 80, -1);
```

NAME

password – Introduction to password and security functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The password and security functions provide means for accessing basic system resources affecting passwords and system security.

ASSOCIATED HEADERS

<grp.h>
<pwd.h>
<rpc/netdb.h>
<stdlib.h>
<sys/sitesec.h>
<sys/types.h> (see sys_types.h)
<udb.h>

ASSOCIATED FUNCTIONS

acid2nam
: Maps IDs to names (see id2nam)

acidnamfree
: Maps IDs to names (see id2nam)

addudb
: Library of user database access functions (see libudb)

deleteudb
: Library of user database access functions (see libudb)

endgrent
: Gets group file entry (see getgrent)

endpwent
: Gets password file entry (see getpwent)

endrpcent
: Gets remote procedure call (RPC) entry (see getrpcent)

endudb
: Library of user database access functions (see libudb)

`fgetgrent`
Gets group file entry (see `getgrent`)

`fgetpwent`
Gets password file entry (see `getpwent`)

`getgrent`
Gets group file entry

`getgrgid`
Gets group file entry (see `getgrent`)

`getgrnam`
Gets group file entry (see `getgrent`)

`getpass`
Reads a password

`getpwent`
Gets password file entry

`getpw` Gets name from UID

`getpwnam`
Gets password file entry (see `getpwent`)

`getpwuid`
Gets password file entry (see `getpwent`)

`getrpcbyname`
Gets remote procedure call (RPC) entry (see `getrpcent`)

`getrpcbynumber`
Gets remote procedure call (RPC) entry (see `getrpcent`)

`getsysudb`
Library of user database access functions (see `libudb`)

`gettrustedudb`
Library of user database access functions (see `libudb`) `getudbchain` Library of user database access functions (see `libudb`) `getudb` Library of user database access functions (see `libudb`)

`getudbnam`
Library of user database access functions (see `libudb`)

`getudbstat`
Library of user database access functions (see `libudb`)

`getudbuid`
Library of user database access functions (see `libudb`)

`gid2nam`
Maps IDs to names (see `id2nam`)

`gidnamfree`
Maps IDs to names (see `id2nam`)

`initgroups`
Initializes group access list

`lockudb`
Library of user database access functions (see `libudb`)

`nam2acid`
Maps IDs to names (see `id2nam`)

`nam2gid`
Maps IDs to names (see `id2nam`) `nam2uid` Maps IDs to names (see `id2nam`)

`putpwent`
Writes password file entry

`rewriteudb`
Library of user database access functions (see `libudb`) `secbits` Returns a bit pattern representing names of security compartments, categories, flags, or permission names (see `secnames`)

`secnames`
Returns a list of security compartments, categories, flags, or permission names

`secnums`
Returns numeric value of given security level or class (see `secnames`)

`secwords`
Returns security level or class given its corresponding numeric value (see `secnames`)

`setdomainname`
Sets name of current domain (see `getdomainname`)

`setgrent`
Gets group file entry (see `getgrent`)

`setpwent`
Gets password file entry (see `getpwent`)

`setrpcent`
Gets remote procedure call (RPC) entry (see `getrpcent`)

`setudb`
Library of user database access functions (see `libudb`)

`setudbpath`
Library of user database access functions (see `libudb`)

`udbisopen`
Library of user database access functions (see `libudb`)

`uid2nam`
: Maps IDs to names (see `id2nam`)

`unlockudb`
: Library of user database access functions (see `libudb`)

`zeroudbstat`
: Library of user database access functions (see `libudb`(3C))

SEE ALSO

`file`(3C), `libudb`(3C), `message`(3C), `network`(3C), `multic`(3C), `terminal`(3C) (all introductory pages to other operating system service functions)

UNICOS File Formats and Special Files Reference Manual, Cray Research publication SR–2014

NAME

pathname – Computes a true path name from a specified path

SYNOPSIS

```
#include <sys/types.h>
#include <pathlib.h>
#include <errno.h>

char *pathname(char *path, mls_t label, unsigned flags, unsigned *pathinfo)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The pathname routine translates a path name specified in *path* that may contain symbolic link references, multilevel directory (MLD) references, and . or .. references into a true path to the specified file.

The *label* argument allows the caller to specify the security label to be used when resolving MLD references. If a null security label is passed (that is, (mls_t) 0), the security label of the current process is used. If a nonnull security label is passed, the specified security label is used instead of the current process security label. This can be useful when the caller must handle requests at a different security label.

The *flags* argument allows the caller to control the nature of the expansion. The following flags can be combined in any call to pathname:

Flag	Description
PN_ABSOLUTE	If this flag is ON, pathname always resolves the specified path to an absolute path, regardless of whether the specified path is absolute or relative. If this flag is OFF, pathname does not try to produce an absolute path. If the path passed in is absolute or some component of the path is a symbolic link that causes the path to become absolute, pathname produces an absolute path even if PN_ABSOLUTE is OFF. If the result of the translation is relative, pathname produces a relative path if PN_ABSOLUTE is OFF.
PN_FULLMLD	If this flag is ON and the final component of the resulting path is a multilevel symbolic link, pathname expands the last component out to the labeled subdirectory. If this flag is OFF under the same conditions, pathname only resolves the path to the root of the MLD. For example, if /tmp is a multilevel symbolic link to /tmp.mld and pathname is called to resolve /tmp with the PN_FULLMLD flag and a label with a zero compartment set and zero level, pathname returns /tmp.mld/000. Under the same circumstances, if the PN_FULLMLD flag is not used, the path returned is /tmp.mld.

PN_NOFOLLOW If this flag is ON and the final component of the resulting path is a symbolic link or a multilevel symbolic link, pathname does not expand the final component. If this flag is OFF, pathname follows all symbolic links and multilevel symbolic links it encounters when resolving *path*.
For example, if /usr/symlink is a symbolic link to /usr/target and pathname is called to resolve /usr/symlink with the PN_NOFOLLOW flag, the path returned is /usr/symlink. Without the PN_NOFOLLOW flag, the path returned is /usr/target.

PN_KEEPERR If this flag is ON and pathname encounters some kind of error while resolving *path*, pathname returns a buffer containing the path that produced the error. If this flag is OFF, pathname returns a null pointer on error.
It is possible for pathname to return a null pointer even if this flag is ON. This can result from dynamic memory exhaustion or corruption within the calling program. The caller must handle a null return even when the PN_KEEPERR flag is specified.

The *pathinfo* argument provides a pointer to a space into which pathname can place flags that describe conditions encountered while translating *path*. If a null pointer is passed for *pathinfo*, these flags are not returned to the caller. The following flags can be set in the location pointed to by *pathinfo* on return from pathname:

Flag	Description
PI_MLSLINK	The final component of *path* resolves to a multilevel symbolic link.
PI_NOTTHERE	The file specified by the final component of *path* does not actually exist, but the path translation was successful up to the last component.
PI_ERROR	An error occurred during translation of *path*. The value of errno describes the error.

WARNINGS

The pathname routine calls stat(2), lstat(2), and readlink(2), among other system calls. As a result, it may sleep or hang if a needed file system resource is unavailable.

RETURN VALUES

If pathname succeeds in translating the provided path, it returns a pointer to a buffer that is allocated by pathname using malloc(3C) and contains the translated path. This buffer can be released by the caller using free(3C). Each call to pathname allocates a new buffer and does not affect the contents of any previously returned buffer.

If pathname fails to translate the specified path, it normally returns a null pointer. If the PN_KEEPERR flag is specified and pathname fails to translate the specified path, pathname returns a pointer to a buffer containing the name of the file that caused the translation to fail.

If pathname is unable to allocate a buffer, it returns a null pointer even if PN_KEEPERR is set, so a null return must be handled by all callers.

Regardless of the setting of the PN_KEEPERR flag, if pathname fails, it sets the PI_ERROR flag in the location specified by *pathinfo* and sets the global variable errno to indicate the error.

FORTRAN EXTENSIONS

None.

EXAMPLES

The following example shows several different ways that pathname can be called to resolve paths. The program runs through all arguments on the command line, resolving each as a pathname in three different ways and printing each result. Failures fall through to the next example to demonstrate the different ways that failure may be handled by pathname.

```
#include <stdio.h>

#include <sys/types.h>
#include <pathlib.h>
#include <errno.h>

main(argc,argv)
int   argc;
char  *argv[];
{
      char        *result;
      unsigned    info;
      int         errs;
      int         i;

      for (i = 1; i < argc; ++i) {
            /*
             * Obtain the translated pathname as a relative path
             * with full MLD expansion and print it.
             */
            printf("Relative resolution with full MLDs\n");
            fflush(stdout);

            result = pathname(argv[i], (mls_t)0, PN_FULLMLD, &info);
            if (result == (char *)0) {
                  fprintf(stderr, "pathname failed for '%s' ", argv[i]);
                  perror("");
                  errs = 1;
            } else {
                  printf("%s\n", result);
                  free(result);
            }
```

```
/*
 * Obtain the pathname as an absolute path with full
 * MLD expansion and print it.
 */

printf("Absolute resolution with full MLDs\n");
fflush(stdout);

result = pathname(argv[i], (mls_t)0, PN_ABSOLUTE | PN_FULLMLD, &info);
if (result == (char *)0) {
      fprintf(stderr, "pathname failed for '%s' ", argv[i]);
      perror("");
      errs = 1;
} else {
      printf("%s\n", result);
      free(result);
}

/*
 * Obtain the pathname as an absolute path with full
 * MLD expansion, preserving the failed pathname on failure
 * and print the result.
 */
printf("Absolute resolution, full MLDs, keeping error path\n");
result = pathname(argv[i], (mls_t)0,
                  PN_KEEPERR | PN_ABSOLUTE | PN_FULLMLD, &info);
/*
 * First check for a null return, in case pathname(3) had
 * trouble allocating its return buffer.
 */
if (result == (char *)0) {
      fprintf(stderr, "pathname failed for '%s' ", argv[i]);
      perror("");
      errs = 1;
      continue;
}

/*
 * Now check for an error indication in the 'info' flags.
 */
if ((info & PI_ERROR) != 0) {
      fprintf(stderr, "pathname failed for '%s' ", argv[i]);
      perror("");
      errs = 1;
```

```
            } else {
                  printf("%s\n", result);
            }
            free(result);
      }
      exit(errs);
}
```

SEE ALSO

`errno.h`(3C), `free`(3C), `malloc`(3C), `mldname`(3C), `mls_create`(3C), `mls_free`(3C)

`lstat`(2), `readlink`(2), and `stat`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

perror, sys_errlist, sys_nerr – Generates system error messages

SYNOPSIS

```
#include <stdio.h>
void  perror (const char *s);
#include <errno.h>
extern char *sys_errlist[  ];
extern int sys_nerr;
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (perror only)
AT&T extension (sys_errlist and sys_nerr)

DESCRIPTION

The perror function produces on the standard error output a message that describes the last error encountered during a call to a system or library function. The argument string *s* is printed first, then a colon and a blank, followed by the message and a newline character. To be of most use, the argument string should include the name of the program incurring the error. The error number is taken from errno, which is set when errors occur but not cleared when error-free calls are made.

To simplify variant formatting of messages, sys_errlist, an array of message strings, is provided; you can use errno as an index in this table to get the message string without the newline character.

Argument sys_nerr is the largest message number provided for in the table plus 1; it should be checked, because new error codes may be added to the system before they are added to the table.

SEE ALSO

intro(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

Cray Message System Programmer's Guide, Cray Research publication SG–2121

NAME

popen, pclose – Initiates a pipe to or from a process

SYNOPSIS

```
#include <stdio.h>
FILE *popen (const char *command, const char *mode);
int pclose (FILE *stream);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX

DESCRIPTION

The arguments to popen are pointers to null-terminated strings that contain, respectively, a shell command line and an I/O mode, either "r" for reading or "w" for writing. The popen function creates a pipe between the calling program and the command to be executed. The value returned is a stream pointer. If the I/O mode is "w", you can write to the standard input of the command by writing to the file *stream*; if the I/O mode is "r", you can read from the standard output of the command by reading from the file *stream*.

A stream opened by popen should be closed by pclose, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, you can use a *mode* "r" command as an input filter and a *mode* "w" command as an output filter.

If the shell cannot be executed, the status returned by pclose is the same as if the shell terminated using _exit(127).

CAUTIONS

If the original process, and the process opened with popen, concurrently read or write a common file, neither should use buffered I/O, because the buffering gets all mixed up. To forestall problems with an output filter, flush the buffer carefully (that is, with fflush(3C)).

RETURN VALUES

If files or processes cannot be created, or if the shell cannot be accessed. the popen function returns a null pointer. Otherwise, it returns a stream pointer as described previously.

On successful return, `pclose` returns the termination status of the shell that ran the command; otherwise, `pclose` returns `-1`, and sets `errno` to indicate the error.

SEE ALSO

`fclose`(3C), `fflush`(3C), `fopen`(3C), `system`(3C)

`pipe`(2), `vfork`(2), `waitpid`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

pow, powf, powl, cpow – Raises the specified value to a given power

SYNOPSIS

```
#include <math.h>
#include <complex.h> (for function cpow only)
double pow (double x, double y);
float powf (float x, float y);
long double powl (long double x, long double y);
double complex cpow (double complex x, double complex y);
```

IMPLEMENTATION

All Cray Research systems (pow, cpow only)
Cray MPP systems (powf only)
Cray PVP systems (powl only)

STANDARDS

ISO/ANSI (pow only)
CRI extension (all others)

DESCRIPTION

The pow, powf, powl, and cpow functions compute *x* raised to the power *y* for double, float, long double, and double complex numbers, respectively. A domain error occurs if *x* is negative and *y* is not an integral value. A domain error also occurs if the result cannot be represented when *x* is 0 and *y* is less than or equal to 0. A range error may occur.

When code containing calls to these functions is compiled by the Cray Standard C compiler in extended mode, domain checking is not done, errno is not set on error, and the functions do not return to the caller on error. If an error occurs, the program aborts, producing a traceback and a core file. On CRAY T90 systems with IEEE floating-point arithmetic only, in extended mode, errno is not set, but the functions do return to the caller on error. For more information, see the corresponding libm man page (for example, POW(3M)).

Specifying the cc(1) command-line option -h stdc (signifying strict conformance mode) or -h matherr=errno causes these functions to perform domain and range checking, set errno on error, and return to the caller on error.

In strict conformance mode, vectorization is inhibited for loops containing calls to any of these functions. Vectorization is not inhibited in extended mode.

RETURN VALUES

The pow, powf, powl, and cpow functions return the value of *x* raised to the power *y*.

When a program is compiled with -hstdc or -hmatherror=errno on Cray MPP systems and CRAY T90 systems with IEEE arithmetic, under certain error conditions the functions perform as follows:

- pow(*x*,*NaN*) returns NaN, and errno is set to EDOM.
- pow(*NaN*,*y*) returns NaN, and errno is set to EDOM.
- powl(*x*,*NaN*) returns NaN, and errno is set to EDOM.
- powl(*NaN*,*y*) returns NaN, and errno is set to EDOM.
- cpow(*x*,*y*), where either the real or imaginary part of *x* or *y* is NaN, returns NaN+NaN*1.0*i*, and errno is set to EDOM.

SEE ALSO

errno.h(3C)

cc(1) in the *Cray Standard C Reference Manual*, Cray Research publication SR–2074

power(3M) in the *Intrinsic Procedures Reference Manual*, Cray Research publication SR–2138

NAME

printf, fprintf, sprintf, snprintf – Prints formatted output

SYNOPSIS

```
#include <stdio.h>
int printf (const char *format, ... );
int fprintf (FILE *stream, const char *format, ... );
int sprintf (char *s, const char *format, ... );
int snprintf (char * restrict s, size_t n const char * restrict format,
... );
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The `printf` function places output on the standard output stream `stdout` and returns the number of characters transmitted or a negative value if an output error was encountered. The `fprintf` function is equivalent to `printf`(3C) with output written to the stream to which *stream* points instead of `stdout`.

The `sprintf` function is equivalent to `printf`, except that the argument *s* specifies an array into which the generated output is written, rather than to `stdout`. You must ensure enough storage space is available. A null character is written at the end of the characters written; it is not counted as part of the returned sum. If copying occurs between objects that overlap, the behavior is undefined.

The `snprintf` function is equivalent to `fprintf`, except that argument *s* specifies an array into which the generated output is to be written, rather than to a stream. If *n* is zero, nothing is written, and *s* may be a null pointer. Otherwise, output characters beyond the *n-1st* are discarded rather than being written to the array, and a null character is written at the end of the characters actually written into the array. If copying takes place between objects that overlap, the behavior is undefined.

Function `printf` converts, formats, and prints its arguments under the control of *format*. The *format* is a multibyte character string that begins and ends in its initial shift state. It contains two types of objects: ordinary multibyte characters (not %), which are simply copied to the output stream; *conversion specifications*, each of which results in the fetch of 0 or more arguments. The results are undefined if insufficient arguments for the format exist. If the format is exhausted while arguments remain, the excess arguments are evaluated, but otherwise ignored.

Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than to the next unused argument. In this case, the % symbol is replaced by the %n$ symbol, where n is a decimal integer in the range [1, NL_ARGMAX], giving the position of the argument in the argument list. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages.

In format strings containing the %n$ symbol, numbered arguments in the argument list can be referenced from the format string as many times as required.

Each conversion specification is introduced by either the % or the %n$ symbol. After the % or %n$, the following appear in sequence:

1. Zero or more *flags*, which modify the meaning of the conversion specification.
2. An optional decimal digit string that specifies a minimum *field width*. If the converted value has fewer characters than the field width, it is padded on the left (or right, if the left-adjustment flag (-) has been given) to the field width. The field width takes the form of an asterisk * (described later) or a decimal integer.
3. A *precision* that gives the minimum number of digits to appear for the d, i, o, u, x, or X conversions; the number of digits to appear after the decimal point for the e and f conversions; the maximum number of significant digits for the g conversion; or the maximum number of characters to be printed from a string in the s conversion. The precision takes the form of a period (.), followed by either an asterisk * (described later) or an optional decimal integer; if you specify only the period, the precision is taken as 0. If a precision appears with any other conversion specifier, the behavior is undefined.
4. An optional h specifying that a following d, i, o, u, x, or X conversion specifier applies to a short int or unsigned short int argument (the argument will have been promoted according to the integral promotions, and its value is converted to short int or unsigned short int before printing); an optional h specifying that a following n conversion specifier applies to a pointer to a short int argument; an optional l (ell) specifying that a following d, i, o, u, x, or X conversion specifier applies to a long int or unsigned long int argument; an optional ll (ell ell) specifying that the following d, i, o, u, x, or X conversion specifier applies to a long long int or unsigned long long int argument; an optional ll, specifying that a following n conversion specifier applies to a pointer or a long int argument; an optional l specifying that a following n conversion specifier applies to a pointer to a long int argument; or an optional L specifying that a following e, E, f, g, or G conversion specifier applies to a long double argument. If an h, l, or L appears with any other conversion specifier, the behavior is undefined.
5. A character that indicates the type of conversion to be applied.

A field width or precision may be indicated by an asterisk (*) instead of a digit string. In this case, an integer argument supplies the field width or precision. The argument that is actually converted is not fetched until the conversion letter is seen, so the argument specifying field width or precision must appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a - flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

In format strings containing the %n$ symbol, a field width or precision may be indicated by the sequence %n$, where m is a decimal integer in the range [1, NL_ARGMAX] giving the position in the argument list (after the format argument) of an integer argument containing the field width or precision. The following is an example:

```
printf ("%1$d:%2$.*$d:%4$.*3$d0, hour, min, precision, sec);
```

The *format* can contain either numbered argument specifications (that is, %n$, and *m$, or unnumbered argument specifications, that is, % and *), but usually not both. The only exception to this is that %% can be mixed with the %n$ form. The results of mixing numbered and unnumbered argument specifications in a *format* string are undefined. When numbered argument specifications are used, specifyng the *n*th argument requires that all the leading arguments, from the first to the (*n–1*th), are specified in the format string.

The flag characters and their meanings are as follows:

Flag	Description
’	The integer portion of the result of a decimal conversion (%i, %d, %u, %f, %g, or %G) are formatted with thousands’ grouping characters. For other conversions, the behavior is undefined. The nonmonetary grouping character is used.
-	Conversion result is left-justified within the field.
+	A signed conversion result always begins with a + or - sign.
space	If the first character of a signed conversion is not a sign, a space is prefixed to the result. This implies that if the space and + flags both appear, the space flag is ignored.
#	Specifies that the value will be converted to an alternative form. For o conversion, it increases the precision to force the first digit of the result to be a 0. For x (X) conversion, a nonzero result has 0x (0X) prefixed to it. For e, E, f, g, and G conversions, the result always contains a decimal point, even if no digits follow the point (usually, a decimal point appears in the result of these conversions only if a digit follows it). For g and G conversions, trailing 0’s are not removed from the result. For other conversions, the behavior is undefined.

For d, i, o, u, x, X, e, E, f, g, and G conversions, leading 0’s (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the 0 and - flags both appear, the 0 flag is ignored. For d, i, o, u, x, and X conversions, if a precision is specified, the 0 flag is ignored. For other conversions, the behavior is undefined.

The conversion characters and their meanings are as follows:

Character	Description
d,i	The integer argument is converted to signed decimal in the style [-]*dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading 0’s. The default precision is 1. The result of converting a 0 value with a precision of 0 is no characters.

o, u, x, X, B The `unsigned int` argument is converted to unsigned octal (`o`), unsigned decimal (`u`), unsigned hexadecimal notation (`x` or `X`), or unsigned binary notation (`B`) in the style *dddd*; the letters `abcdef` are used for `x` conversion, and the letters `ABCDEF` are used for `X` conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading 0's. The default precision is 1. The result of converting a 0 value with a precision of 0 is no characters.

f The `double` argument is converted to decimal notation in the style [-]*ddd.ddd*, in which the number of digits after the decimal point is equal to the precision specification. If the precision is missing, 6 digits are output; if the precision is explicitly 0 and the # flag is not specified, no decimal point appears. If a decimal point character appears, at least 1 digit appears before it. The value is rounded to be the appropriate number of digits.

e, E The `double` argument is converted in the style [-]*d.ddd*e±*dd*, in which a 1 digit is before the decimal point, and the number of digits after it is equal to the precision; when the precision is missing, it is assumed to be 6; if the precision is 0 and the # flag is not specified, no decimal point appears. The value is rounded to the appropriate number of digits. The `E` format code produces a number with `E` instead of `e` introducing the exponent. The exponent always contains at least 2 digits. If the value is 0, the exponent is 0.

g, G The `double` argument is printed in style `f` or `e` (or in style `E` in the case of a `G` format code), with the precision specifying the number of significant digits. If the precision is 0, it is taken as 1. The style used depends on the value converted: style `e` is used only if the exponent resulting from the conversion is less than −4 or greater than or equal to the precision. Trailing 0's are removed from the fractional portion of the result; a decimal point appears only if it is followed by a digit.

c The `int` argument is converted to an `unsigned char`, and the resulting character is written.

s The argument is taken to be a string (character pointer), and characters from the string are printed until a null character (`\0`) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed.

p The argument is a pointer to `void`. The value of the pointer is converted to a sequence of printable characters, in an implementation-defined manner. On Cray Research systems, the conversion is the same as the `o` conversion.

n The argument is a pointer to an integer into which is written the number of characters written to the output stream so far by this call to `fprintf`. No argument is converted.

C The `wchar_t` argument is converted to an array of bytes representing a character, and the resulting character is written. If the precision is specified, its effect is undefined. The conversion is the same as the expected the `wctomb()` function.

S The argument must be a pointer to an array of type `wchar_t`. Wide character codes from the array up to, but not including any terminating null wide-character code, are converted to a sequence of bytes, and the resulting bytes written. If the precision is specified no more than that, many bytes are written and only complete characters are written. If the precision is not specified, or is greater than the size of the array of converted bytes, the array of wide characters must be terminated by a null wide character. The conversion is the same as that expected from the `wcstombs()` function.

% This flag prints a %; no argument is converted.

If any argument is, or points to, a union or an aggregate (except for an array of character type using `%s` conversion, or a pointer using `%p` conversion), the behavior is undefined.

If a conversion specification is not valid, the behavior is undefined. In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by `printf` and `fprintf` are printed as if `putc`(3C) had been called.

For machines with IEEE arithmetic, the `e`, `E`, `f`, `g`, and `G` formats print infinity as `Inf` and "not-a-number" as `NaN`.

RETURN VALUES

The `printf` and `fprintf` functions return the number of characters transmitted, or a negative value if an output error occurred.

The `sprintf` function returns the number of characters written in the array, not counting the terminating null character.

EXAMPLES

To print a date and time in the form ‘‘Sunday, July 3, 10:02’’, where *weekday* and *month* are pointers to null-terminated strings, enter the following command line:

```
printf("%s, %s %d, %.2d:%.2d", weekday, month, day, hour, min);
```

To print *pi* to 5 decimal places, enter the following command line:

```
printf("pi = %.5f", 4*atan(1.0));
```

SEE ALSO

`ecvt`(3C), `putc`(3C), `scanf`(3C), `stdio.h`(3C), `vprintf`(3C)

NAME

priv_clear_file – Clears all privilege sets in a file privilege state

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priv.h>

int priv_clear_file(priv_file_t *privstate);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_clear_file routine clears all privilege sets in the file privilege state to which *privstate* points.

RETURN VALUES

A return value of 0 indicates success. A return value of –1 indicates that an error has occurred, and an error code is stored in *errno*. If the return value is –1, the contents of the privilege state to which *privstate* points is not affected.

ERRORS

priv_clear_file fails if the following error condition occurs:

Error Code	Description
EINVAL	An illegal or undefined value was supplied for *privstate*.

NAME

priv_clear_proc – Clears all privilege sets in a process privilege state

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priv.h>

int priv_clear_proc(priv_proc_t *privstate);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_clear_proc routine clears all privilege sets in the process privilege state to which *privstate* points.

RETURN VALUES

A return value of 0 indicates success. A return value of –1 indicates that an error has occurred, and an error code is stored in *errno*. If the return value is –1, the contents of the privilege state to which *privstate* points is not affected.

ERRORS

priv_clear_proc fails if the following error condition occurs:

Error Code	Description
EINVAL	An illegal undefined value was supplied for *privstate*.

SEE ALSO

priv_init_proc(3C)

NAME

priv_dup_file – Creates a copy of a file privilege state

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priv.h>

priv_file_t *priv_dup_file(priv_file_t *source);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_dup_file routine creates a copy of the file privilege state to which *source* points. This routine allocates any memory necessary to hold the new file privilege state and returns a pointer to that privilege state. Once duplicated, an operation on either privilege state does not affect the other.

RETURN VALUES

If successful, returns a pointer to the new file privilege state. A return value of null indicates that an error has occurred, and an error code is stored in *errno*.

ERRORS

priv_dup_file fails if any of the following error conditions occur:

Error Code	Description
EINVAL	An illegal or undefined value was supplied for *source*.
ENOMEM	Insufficient memory was available to allocate the new file privilege state.

NAME

priv_dup_proc – Creates a copy of a process privilege state

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priv.h>

priv_proc_t *priv_dup_proc(priv_proc_t *source);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_dup_proc routine creates a copy of the process privilege state to which *source* points. This routine allocates any memory necessary to hold the new process privilege state and returns a pointer to that privilege state. Once duplicated, an operation on either privilege state does not affect the other.

RETURN VALUES

If successful, priv_dup_proc returns a pointer to the new process privilege state. A return value of null indicates that an error has occurred, and an error code is stored in *errno*.

ERRORS

priv_dup_proc fails if any of the following error conditions occur:

Error Code	Description
EINVAL	An illegal or undefined value was supplied for *source*.
ENOMEM	Insufficient memory was available to allocate the new process.

NAME

priv_free_file – Deallocates file privilege state space

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priv.h>

priv_free_file(priv_file_t *privstate);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_free_file routine deallocates space associated with the file privilege state to which *privstate* points.

RETURN VALUES

A return value of 0 indicates success. A return value of –1 indicates that an error has occurred, and an error code is stored in *errno*.

ERRORS

priv_free_file fails if the following error condition occurs:

Error Code	Description
EINVAL	An illegal or undefined value was supplied for *privstate*.

SEE ALSO

priv_init_file(3C)

NAME

priv_free_proc – Deallocates process privilege state space

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priv.h>

int priv_free_proc(priv_proc_t *privstate);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_free_proc routine deallocates the space associated with the process privilege state to which *privstate* points.

RETURN VALUES

A return value of 0 indicates success. A return value of –1 indicates that an error has occurred, and an error code is stored in *errno*.

ERRORS

priv_free_proc fails if the following error condition occurs:

Error Code	Description
EINVAL	An illegal or undefined value was supplied for *privstate*.

SEE ALSO

priv_init_proc(3C)

NAME

priv_get_fd – Gets the privilege state of a file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priv.h>

priv_file_t *priv_get_fd(int fdes);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_get_fd routine uses the fgetpal(2) system call to get the privilege state of the file identified by the file descriptor *fdes*. This function allocates any memory necessary to hold the file privilege state and returns a pointer to that privilege state.

The caller must have MAC read access to the file or have PRIV_MAC_READ in its effective privilege set.

RETURN VALUES

If successful, priv_get_fd returns a pointer to the file privilege state. A return value of null indicates that an error has occurred, and an error code is stored in *errno*.

ERRORS

priv_get_fd fails if any of the following error conditions occur:

Error Code	Description
EBADF	An illegal or undefined value was specified for *fdes*.
EACCES	The caller does not have MAC read access to the file.
ENOMEM	Insufficient memory was available to allocate the file privilege state.

SEE ALSO

priv_get_file(3C), priv_set_fd(3C), priv_set_file(3C)

fgetpal(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

priv_get_file – Gets the privilege state of a file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priv.h>

priv_file_t *priv_get_file(char *path);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_get_file routine uses the getpal(2) system call to get the privilege state of the file identified by *path*. This function allocates any memory necessary to hold the file privilege state and returns a pointer to that privilege state.

The caller must have MAC read access to the file or have PRIV_MAC_READ in its effective privilege set.

RETURN VALUES

If successful, priv_get_file returns a pointer to the file privilege state. A return value of null indicates that an error has occurred, and an error code is stored in *errno*.

ERROR

priv_get_file fails if any of the following error conditions occur:

Error Code	Description
EFAULT	The *path* argument points outside the process address space.
EACCES	Search permission is denied for a component of the *path* prefix or the caller does not have MAC read access to the file.
ENOENT	A component of the specified *path* does not exist.
ENOTDIR	A component of the *path* prefix is not a directory.
ENAMETOOLONG	The length of the *path* argument exceeds PATH_MAX, or a path name component is longer than NAME_MAX while POSIX_NO_TRUNC is in effect.
ENOMEM	Insufficient memory was available to allocate the file privilege state.

SEE ALSO

priv_get_fd(3C), priv_set_fd(3C), priv_set_file(3C)

getpal(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

priv_get_file_flag – Indicates the existence of a privilege in a file privilege set

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priv.h>

int priv_get_file_flag(priv_file_t *privstate, priv_value_t priv,
priv_fflag_t flag, priv_flag_value_t *value_p);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_get_file_flag routine indicates whether the privilege identified by *priv* exists in the file privilege set, identified by *flag*, of the file privilege state to which *privstate* points. A value that indicates whether the privilege exists is placed in the location to which *value_p* points.

If the value placed in the location to which *value_p* points is PRIV_SET, the specified privilege exists in the privilege set. If the privilege does not exist, the value placed in the location to which *value_p* points is PRIV_CLEAR.

The *priv* argument is the privilege identifier (for example, PRIV_MAC_READ). The *flag* argument is a privilege set identifier. PRIV_ALLOWED, PRIV_FORCED, and PRIV_SETEFF identify the file's allowed, forced, and set-effective privilege sets, respectively.

RETURN VALUES

A return value of 0 indicates success. A return value of –1 indicates that an error has occurred, and an error code is stored in *errno*. If the return value is –1, the contents of the location to which *value* points is not affected.

ERRORS

priv_get_file_flag fails if the following error condition occurs:

Error Code	Description
EINVAL	An illegal or undefined value was supplied for *privstate*, *priv*, *value_p*, or *flag*.

SEE ALSO

priv_set_file_flag(3C)

NAME

priv_get_proc – Gets the privilege state of the calling process

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priv.h>

priv_proc_t *priv_get_proc();
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_get_proc routine uses the getppriv(2) system call to get the privilege state of the calling process. This routine allocates any memory necessary to hold the process privilege state and returns a pointer to that privilege state.

RETURN VALUES

If successful, priv_get_proc returns a pointer to the process privilege state. A return value of null indicates that an error has occurred, and an error code is stored in *errno*.

ERRORS

priv_get_proc fails if the following error condition occurs:

Error Code	Description
ENOMEM	Insufficient memory was available to allocate the privilege state.

SEE ALSO

getppriv(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

priv_get_proc_flag – Indicates the existence of a privilege in a process privilege state

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priv.h>

int priv_get_proc_flag(priv_proc_t *privstate, priv_value_t priv,
priv_pflag_t flag, priv_flag_value_t *value_p);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_proc_flag routine indicates whether the privilege identified by *priv* exists in the process privilege set, identified by *flag*, of the process privilege state to which *privstate* points. A value that indicates whether the privilege exists is placed in the location to which *value_p* points.

If the value placed in the location to which *value_p* points is PRIV_SET, then the specified privilege exists in the privilege set. If the privilege does not exist, the value placed in the location to which *value_p* points is PRIV_CLEAR.

The *priv* argument is the privilege identifier (for example, PRIV_MAC_READ). The *flag* argument is a privilege set identifier. PRIV_PERMITTED and PRIV_EFFECTIVE identify the process permitted and effective privilege sets, respectively.

RETURN VALUES

A return value of 0 indicates success. A return value of –1 indicates that an error has occurred, and an error code is stored in *errno*. If the return value is –1, the contents of the location to which *value_p* points is not modified.

ERRORS

priv_get_proc_flag fails if the following error condition occurs:

Error Code	Description
EINVAL	An illegal or undefined value was supplied for *privstate*, *priv*, *value_p*, or *flag*.

SEE ALSO

priv_set_proc_flag(3C)

NAME

priv_init_file – Allocates space to hold a file privilege state

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priv.h>

priv_file_t *priv_init_file();
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_init_file routine allocates space to hold a file privilege state and returns a pointer to that privilege state. The allocated space is cleared.

RETURN VALUES

If successful, priv_init_file returns a pointer to the allocated space. A return value of null indicates that an error has occurred, and an error code is stored in *errno*.

ERRORS

priv_init_file fails if the following error condition occurs:

Error Code	Description
ENOMEM	Insufficient memory was available to allocate the file privilege state.

SEE ALSO

priv_free_file(3C)

NAME

priv_init_proc – Allocates space to hold a process privilege state

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priv.h>

priv_proc_t *priv_init_proc();
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_init_proc routine allocates space to hold a process privilege state and returns a pointer to that privilege state. The allocated space is cleared.

RETURN VALUES

If successful, priv_init_proc returns a pointer to the allocated space. A return value of null indicates that an error occurred and an error code is stored in *errno*.

ERRORS

priv_init_proc fails if the following error condition occurs:

Error Code	Description
ENOMEM	Insufficient memory was available to allocate the process privilege state.

SEE ALSO

priv_free_proc(3C)

NAME

priv_set_fd – Sets the privilege state of a file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priv.h>

int priv_set_fd(int fdes, priv_file_t *privstate);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_set_fd routine uses the fsetpal(2) system call to set the privilege state of the file identified by the file descriptor *fdes* to the file privilege state to which *privstate* points.

The calling process must have PRIV_SETFPRIV in its effective privilege set, each privilege whose state is being altered must exist in the permitted privilege set of the calling process, and the caller must either own the file or have the privilege PRIV_FOWNER in its effective privilege set.

This routine retrieves the file's current privilege assignment list (PAL) records, combines the records with the supplied privilege state, and passes the result to fsetpal(2). This routine does not modify the PAL records of a file. The caller must have MAC read and write access to the file, or have PRIV_MAC_READ and PRIV_MAC_WRITE in its effective privilege set.

RETURN VALUES

A return value of 0 indicates success. A return value of –1 indicates that an error has occurred and an error code is stored in *errno*. If the return value is –1, the privilege state of the file is not affected.

ERRORS

priv_set_fd fails if any of the following error conditions occur:

Error Code	Description
EINVAL	An illegal or undefined value was supplied for *privstate*.
EPERM	The calling process does not have PRIV_SETFPRIV in its effective privilege set, is not the file's owner, or is attempting to change the state of a privilege that it does not have in its permitted privilege set.
EROFS	The named file resides on a read-only file system.
EBADF	An illegal or undefined value was specified for *fdes*.
EACCES	The caller does not have MAC read or MAC write access to the file.

SEE ALSO

`priv_get_fd`(3C), `priv_get_file`(3C), `priv_set_file`(3C)

`fsetpal`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

priv_set_file – Sets the privilege state of a file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priv.h>

int priv_set_file(char *path, priv_file_t *privstate);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_set_file routine uses the setpal system call to set the privilege state of the file identified by *path* to the file privilege state to which *privstate* points.

The calling process must have PRIV_SETFPRIV in its effective privilege set, each privilege whose state is being altered must exist in the calling process' permitted privilege set, and the caller must either own the file or have the PRIV_FOWNER privilege in its effective privilege set.

This routine retrieves the file's current privilege assignment list (PAL) category records, combines the records with the supplied privilege state, and passes the result to setpal(2). This routine does not modify the PAL category records of a file. The caller must have MAC read and write access to the file, or have PRIV_MAC_READ and PRIV_MAC_WRITE in its effective privilege set.

RETURN VALUES

A return value of 0 indicates success. A return value of –1 indicates that an error has occurred, and an error code is stored in *errno*. If the return value is –1, the privilege state of the file is not affected.

ERRORS

priv_set_file fails if any of the following error conditions occur:

Error Code	Description
EFAULT	The *path* argument points outside the process address space.
EINVAL	An illegal or undefined value was supplied for *privstate*.
EACCES	Search permission is denied for a component of the *path* prefix.
ENOENT	A component of the specified *path* does not exist.
ENOTDIR	A component of the *path* prefix is not a directory.
ENAMETOOLONG	The length of the *path* argument exceeds PATH_MAX, or a path name component is longer than NAME_MAX while POSIX_NO_TRUNC is in effect.

EPERM	The calling process does not have PRIV_SETFPRIV in its effective privilege set, is not the file's owner, or is attempting to change the state of a privilege that it does not have in its permitted privilege set.
EROFS	The specified file resides on a read-only file system.
EACCES	The caller does not have both MAC read and MAC write access to the file.

SEE ALSO

priv_get_fd(3C), priv_get_file(3C), priv_set_fd(3C)

setpal(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

priv_set_file_flag – Adds or removes privileges of a file privilege set

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priv.h>

int priv_set_file_flag(priv_file_t *privstate, priv_fflag_t flag, int npriv,
priv_value_t *privs, priv_flag_value_t value);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_set_file_flag routine adds (or removes) the privileges specified in the *privs* array to (or from) the file privilege set, identified by *flag*, of the file privilege state to which *privstate* points.

To add the specified privileges, *value* must be PRIV_SET. To remove the specified privileges, *value* must be PRIV_CLEAR.

Each element in the *privs* array is a privilege identifier (for example, PRIV_MAC_READ). The *flag* argument is a privilege set identifier. The PRIV_ALLOWED, PRIV_FORCED, and PRIV_SETEFF flags identify the file's allowed, forced, and set-effective privilege sets, respectively.

RETURN VALUES

A return value of 0 indicates success. A return value of –1 indicates that an error has occurred, and an error code is stored in *errno*. If the return value is –1, the contents of the specified privilege set is not affected.

ERRORS

priv_set_file_flag fails if the following error condition occurs:

Error Code	Description
EINVAL	An illegal or undefined value was supplied for *privstate*, *flag*, *npriv*, *privs*, or *value*.

SEE ALSO

priv_get_file_flag(3C)

NAME

priv_set_proc – Sets the privilege state of the calling process

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priv.h>

int priv_set_proc(priv_proc_t *privstate);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_set_proc routine uses the setppriv(2) system call to set the privilege state of the calling process to the process privilege state to which *privstate* points. The function returns an error if an attempt is made to modify the state of any privilege that is not in the calling process' permitted privilege set.

RETURN VALUES

A return value of 0 indicates success. A return value of –1 indicates that an error has occurred, and an error code is stored in *errno*. If the return value is –1, the privilege state of the calling process is not affected.

ERRORS

priv_set_proc fails if any of the following error conditions occur:

Error Code	Description
EINVAL	An illegal or undefined value was supplied for *privstate*.
EPERM	The calling process attempted to modify the state of a privilege that was not in its permitted privilege set.

SEE ALSO

priv_get_proc(3C)

setppriv(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

priv_set_proc_flag – Adds or removes privileges of a process privilege state

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priv.h>

int  priv_set_proc_flag(priv_proc_t *privstate, priv_pflag_t flag, int npriv,
priv_value_t *privs, priv_flag_value_t value);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The priv_set_proc_flag routine adds (or removes) the privileges specified in array *privs* to (or from) the process privilege set identified by *flag* of the process privilege state to which *privstate* points. The number of privileges specified in the array is *npriv*.

To add the specified privileges, *value* must be PRIV_SET. To remove the specified privileges, *value* must be PRIV_CLEAR.

Each element in the *privs* array is a privilege identifier (for example, PRIV_MAC_READ). The *flag* argument is a privilege set identifier. PRIV_PERMITTED and PRIV_EFFECTIVE identify the process permitted and effective privilege sets, respectively.

RETURN VALUES

A return value of 0 indicates success. A return value of –1 indicates that an error has occurred, and an error code is stored in *errno*. If the return value is –1, the contents of the specified privilege set is not affected.

ERRORS

priv_set_proc_flag fails if the following error condition occurs:

Error Code	Description
EINVAL	An illegal or undefined value was supplied for *privstate*, *flag*, *npriv*, *privs*, or *value*.

SEE ALSO

priv_get_proc_flag(3C)

NAME

prog_diag – Introduction to program diagnostics and error handling functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The program diagnostics and error handling functions provide various means for aiding the programmer in diagnosing programming errors; detecting, reporting, and diagnosing run-time errors; and measuring program performance. These functions differ from other debugging and performance measuring facilities in that these are executed within the program at run time.

errno

The value of errno is 0 at program startup, but is never set to 0 by any library function. A library function may set errno to a positive value to indicate the type of error. All those library functions that set errno on error are so documented in that manual entry. If the calling function wishes to check for errors, it is the caller's responsibility to set errno to 0 before the call and then check after the call.

errno is a macro that expands to an expression that can be used anywhere a simple variable can be used. In strict ANSI terminology, it is a modifiable lvalue. If a program defines an identifier with the name errno, the behavior is undefined.

By default, Standard C library math functions do not support errno to provide significantly better performance. You must specify the -hstdc option or the -h matherr=errno option on the cc command line to force the math functions to support errno.

Function sys_errlist provides an array of message strings; function sys_nerr provides the largest message number in system table (see perror).

Error Messages

If errno has been set, the associated error message can be formatted with either the perror or the strerror function.

Program Termination

The exit(2) system call allows a process to return its error status to its parent process.

ASSOCIATED HEADERS

<assert.h>
<errno.h>

ASSOCIATED FUNCTIONS

Function	Description
assert	Verifies program assertion
FLOWMARK	Provides a more detailed flowtrace
STKSTAT	Collects stack statistics
STACKSZ	Reports stack statistics (see STKSTAT)
tracebk	Prints a traceback

SEE ALSO

clock(3C), perror(3C), rtclock(3C)

performance(7) (available only online)

Guide to Parallel Vector Applications, Cray Research publication SG–2182

Scientific Libraries Reference Manual, Cray Research publication SR–2081

Intrinsic Procedures Reference Manual, Cray Research publication SR–2138

NAME

pthread_create, pthread_detach, pthread_join, pthread_exit, pthread_self, pthread_equal, pthread_once, pthread_attr_init, pthread_attr_destroy, pthread_attr_setdetachstate, pthread_attr_getdetachstate, pthread_attr_setstacksize, pthread_attr_getstacksize, pthread_attr_setstackaddr, pthread_attr_getstackaddr – Thread management

SYNOPSIS

```
#include <pthread.h>

int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void
*(*start_routine)(void *), void *arg);

int pthread_detach (pthread_t thread);

int pthread_join (pthread_t thread, void **value_ptr);

void pthread_exit (void *value_ptr);

pthread_t pthread_self (void);

int pthread_equal (pthread_t t1, pthread_t t2);

pthread_once_t once_control = PTHREAD_ONCE_INIT;

int pthread_once (pthread_once_t *once_control, void (*init_routine)(void));

int pthread_attr_init (pthread_attr_t *attr);

int pthread_attr_destroy (pthread_attr_t *attr);

int pthread_attr_setdetachstate (pthread_attr_t *attr, int detachstate);

int pthread_attr_getdetachstate (const pthread_attr_t *attr,
int *detachstate);

int pthread_attr_setstacksize (pthread_attr_t *attr, size_t stacksize);

int pthread_attr_getstacksize (const pthread_attr_t *attr,
size_t *stacksize);

int pthread_attr_setstackaddr (pthread_attr_t *attr, void *stackaddr);

int pthread_attr_getstackaddr (const pthread_attr_t *attr,
void **stackaddr);
```

IMPLEMENTATION

Cray PVP systems systems

STANDARDS

PThreads

DESCRIPTION

The `pthread_create` function creates a new thread, with attributes (see below) specified by *attr*, within a process. If *attr* is a null value, the default attributes are used. Upon successful completion, `pthread_create` stores the ID of the created thread in the location referenced by *thread*.

The thread begins execution in *start_routine* with *arg* as its sole argument. If *start_routine* returns, its effect is as though `pthread_exit` had been called using the return value of *start_routine* as the exit status. The new thread inherits its signal mask from the creating thread.

The `pthread_join` function suspends execution of the calling thread until the target *thread* terminates, unless the target thread has already terminated. On return from a successful `pthread_join` call with a non-null *value_ptr* argument, the value passed to `pthread_exit` by the terminating thread is made available in the location referenced by *value_ptr*. When a `pthread_join` function returns successfully, the target thread has been terminated. An attempt to call `pthread_join` on a detached target thread returns an error. Only one thread can call `pthread_join` for a given thread.

The `pthread_exit` function terminates the calling thread and makes the value *value_ptr* available to any successful join with the terminating thread. Any cancellation cleanup handlers that have been pushed and not yet popped shall be popped in the reverse order that they were pushed and then executed. After all cancellation cleanup handlers have been executed, if the thread has any thread-specific data, any destructor functions previously specified are called.

The `pthread_exit` function cannot be called from a cancellation cleanup handler or destructor function that was invoked as a result of either an implicit or explicit call to `pthread_exit`.

The `pthread_self` routine returns the thread ID of the caller.

The `pthread_equal` function compares the thread IDs *t1* and *t2*.

The `pthread_detach` function detaches the calling thread. Storage for the thread *thread* is reclaimed when that thread terminates.

The first call to `pthread_once` by any thread in a process, with a given *once_control*, calls the *init_routine* with no arguments. Subsequent calls to `pthread_once` with the same *once_control* do not call the *init_routine*. On return from `pthread_once`, *init_routine* is guaranteed to be completed. The *once_control* parameter determines whether the associated initialization routine has been called.

The behavior of `pthread_once` is not as described here if *once_control* has automatic storage duration or is not initialized by `PTHREAD_ONCE_INIT`.

The `pthread_attr_init` function initializes a thread attributes object *attr* with the default value for all of the individual attributes used by a given implementation.

The resulting thread attributes object (possibly modified by setting individual attribute values), when used by `pthread_create`, defines the attributes of the thread created. A single attributes object can be used in multiple simultaneous calls to `pthread_create`.

Once an attributes object is no longer needed, `pthread_attr_destroy` should be called to ensure that any system resources are released. A thread attributes object can be destroyed while threads that were created with that attributes object are executing. After calling `pthread_attr_destroy`, the thread attributes object cannot be used as an argument to `pthread_create`.

The *detachstate* attribute controls whether the thread is created in a detached state. If the thread is created detached, then the ID of the newly created thread cannot be specified to the `pthread_join` function.

The `pthread_attr_setdetachstate` and `pthread_attr_getdetachstate` functions set and get the *detachstate* attribute, respectively, in the *attr* object. The *detachstate* attribute is set to either `PTHREAD_CREATE_DETACHED` or `PTHREAD_CREATE_JOINABLE`. A value of `PTHREAD_CREATE_DETACHED` causes all threads created with *attr* to be in the detached state; using a value of `PTHREAD_CREATE_JOINABLE` causes all threads created with *attr* to be in the joinable state. The default value of the *detachstate* attribute is `PTHREAD_CREATE_JOINABLE`.

The `pthread_attr_setstacksize` and `pthread_attr_getstacksize` functions set and get the *stacksize* attribute, respectively, in the *attr* object. The `pthread_attr_setstackaddr` and `pthread_attr_getstackaddr` functions set and get the *stackaddr* attribute, respectively, in the *attr* object. Since neither of these attributes are supported, these functions return `ENOSYS` if called.

RETURN VALUES

The `pthread_exit` function does not return a value.

The `pthread_equal` function returns a nonzero value if *t1* and *t2* are equal; otherwise, 0 is returned.

If successful, all the other functions return 0. Otherwise, an error number is returned to indicate the error.

ERRORS

If any of the following conditions occur, the `pthread_create` function returns the corresponding error numbers:

`EAGAIN` The system lacked the necessary resources to create another thread, or the system-imposed limit on the total number of threads in a process was exceeded.

`EINVAL` The specified attributes are invalid.

If any of the following conditions occur, the `pthread_join` and `pthread_detach` functions return the corresponding error number:

`EINVAL` The specified thread is detached.

`ESRCH` No thread could be found corresponding to that specified by the given thread ID.

If any of the following conditions occur, the `pthread_join` function returns the corresponding error number:

`EDEADLK` The value of *thread* specifies the calling thread.

If any of the following conditions occur, the `pthread_attr_setdetachstate` function returns the corresponding error number:

`EINVAL` The value of *detachstate* was invalid.

If any of the following conditions occur, the `pthread_attr_init` function returns the corresponding error number:

`ENOMEM` Insufficient memory exists to create the thread attributes object.

If any of the following conditions occur, the `pthread_attr_getstacksize`, `pthread_attr_setstacksize`, `pthread_attr_getstackaddr`, and `pthread_attr_setstackaddr` functions return the corresponding error number:

`ENOSYS` The *stacksize* or *stackaddr* attributes are not defined.

SEE ALSO

`pthread_atfork`(3C), `pthread_cancel`(3C), `pthread_cond`(3C), `pthread_kill`(3C), `pthread_mutex`(3C), `pthread_spec`(3C)

NAME

pthread_atfork – Register fork handlers

SYNOPSIS

```
#include <sys/types.h>
#include <pthread.h>

int pthread_atfork (void (*prepare)(void), void (*parent)(void), void
(*child)(void));
```

IMPLEMENTATION

Cray PVP systems systems

STANDARDS

PThreads

DESCRIPTION

The pthread_atfork function shall declare fork handlers to be called before and after fork, in the context of the thread that called fork. The *prepare* fork handler shall be called before fork processing commences. The *parent* fork handler shall be called after fork processing completes in the parent process. The *child* fork handler shall be called after fork processing completes in the child process. If no handling is desired at one or more of these three points, the corresponding fork handler address(es) may be set to null.

The order of calls to pthread_atfork is significant. The *parent* and *child* fork handlers shall be called in the order in which they were established by calls to pthread_atfork. The *prepare* fork handlers shall be called in the opposite order.

RETURN VALUES

Upon successful completion, the pthread_atfork function shall return 0. Otherwise, an error number is returned to indicate the error.

ERRORS

If any of the following conditions occur, the pthread_atfork function shall return the corresponding error number:

ENOMEM Insufficient table space exists to record the fork handler addresses.

SEE ALSO

fork(2) in the

NAME

pthread_condattr_init, pthread_condattr_destroy, pthread_cond_init, pthread_cond_destroy, pthread_cond_signal, pthread_cond_broadcast, pthread_cond_wait, pthread_cond_timedwait – Condition variables

SYNOPSIS

```
#include <pthread.h>

int pthread_condattr_init (pthread_condattr_t *attr);

int pthread_condattr_destroy (pthread_condattr_t *attr);

int pthread_cond_init (pthread_cond_t *cond,
 const pthread_condattr_t *attr);

int pthread_cond_destroy (pthread_cond_t *cond);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int pthread_cond_signal (pthread_cond_t *cond);

int pthread_cond_broadcast (pthread_cond_t *cond);

int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex);

int pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mutex,
const struct timespec *abstime);
```

IMPLEMENTATION

Cray PVP systems systems

STANDARDS

PThreads

DESCRIPTION

The function pthread_condattr_init initializes *attr*, a *condition variable attributes object* (CVAO), using the default value for all attributes. Currently there are no supported attributes. Attempting to initialize an already initialized CVAO results in undefined behavior.

The pthread_condattr_destroy function destroys the specified CVAO. This allows the system to reclaim any resources used by the CVAO.

The pthread_cond_init function initializes the condition variable referenced by *cond* with attributes referenced by *attr*. If *attr* is a null value, the default condition variable attributes are used; this is the same as passing the address of a default CVAO. Upon successful initialization, the state of the condition variable becomes initialized. Attempting to initialize an already initialized condition variable results in undefined behavior.

After a CVAO initializes one or more condition variables, any function affecting the CVAO (including destruction) has no effect on any previously initialized condition variables.

The `pthread_cond_destroy` function destroys the given condition variable specified by *cond*; the object becomes, in effect, uninitialized. A destroyed condition variable object can be reinitialized using `pthread_cond_init`; the results of otherwise referencing the object after it has been destroyed are undefined.

Attempting to destroy a condition variable upon which other threads are currently blocked results in undefined behavior.

In cases for which default condition variable attributes are appropriate, the macro `PTHREAD_COND_INITIALIZER` can initialize condition variables that are statically allocated. This is equivalent to dynamic initialization by a call to `pthread_cond_init` with parameter *attr* specified as a null value, except that no error checks are performed.

The `pthread_cond_signal` call unblocks at least one of the threads that are blocked on the specified condition variable *cond* (if any threads are blocked on *cond*).

The `pthread_cond_broadcast` call unblocks all threads currently blocked on the specified condition variable *cond*.

If more than one thread is blocked on a condition variable, the scheduling policy determines the order in which threads are unblocked. When each thread unblocked as a result of a `pthread_cond_signal` or the `pthread_cond_broadcast` call returns from its call to `pthread_cond_wait` or `pthread_cond_timedwait`, the thread owns the mutex with which it called `pthread_cond_wait` or `pthread_cond_timedwait`. The unblocked thread(s) contend for the mutex as if each had called `pthread_mutex_lock`.

The `pthread_cond_signal` or `pthread_cond_broadcast` functions may be called by a thread, whether or not it currently owns the mutex that threads calling `pthread_cond_wait` or `pthread_cond_timedwait` have associated with the condition variable during their waits; however, if predictable scheduling behavior is required, that mutex is locked by the thread calling `pthread_cond_signal` or `pthread_cond_broadcast`.

The `pthread_cond_signal` and `pthread_cond_broadcast` functions have no effect if there are no threads currently blocked on *cond*.

The `pthread_cond_wait` and `pthread_cond_timedwait` functions are used to block on a condition variable. They are called with *mutex* locked by the thread, or undefined behavior will result.

These functions atomically release *mutex* and cause the calling thread to block on the condition variable *cond*. Atomically here means "atomically with respect to access by another thread to the mutex and then the condition variable." That is, if another thread can acquire the mutex after it is released by an about-to-block thread, a subsequent call to `pthread_cond_signal` or `pthread_cond_broadcast` in that thread behaves as if it were issued after the other thread has blocked. Upon successful return, the mutex is locked and is owned by the calling thread.

When condition variables are used, there is always a boolean predicate involving a shared variable for each condition wait, which is true if the `pthread_cond_timedwait` functions may occur. Since the return from `pthread_cond_wait` or `pthread_cond_timedwait` implies nothing about this predicate's value, the predicate should be reevaluated upon such return.

More than one mutex should not be used for concurrent `pthread_cond_wait` or `pthread_cond_timedwait` operations on the same condition variable, as this can result in improper behavior from these functions.

The `pthread_cond_timedwait` function is the same as `pthread_cond_wait`, except that an error is returned if the absolute time specified by *abstime* passes (that is, if system time equals or exceeds *abstime*) before the condition *cond* is signaled or broadcast, or if the absolute time specified by *abstime* has already been passed at the time of the call. When such time-outs occur, `pthread_cond_timedwait` nonetheless releases and reacquires the mutex referenced by *mutex*.

RETURN VALUES

If successful, all of these functions return 0. Otherwise, an error number is returned to indicate the error.

MESSAGES

If any of the following conditions occur, the `pthread_condattr_init` function returns the corresponding error number:

`ENOMEM` Insufficient memory exists to initialize the condition variable attributes object.

If any of the following conditions occur, the `pthread_cond_init` function returns the corresponding error number:

`EAGAIN` The system lacked the necessary resources (other than memory) to initialize another condition variable.

`ENOMEM` Insufficient memory exists to initialize the condition variable.

If any of the following conditions occur, the `pthread_cond_timedwait` function returns the corresponding error number.

`ETIMEDOUT` The time specified by *abstime* to `pthread_cond_timedwait` has passed.

SEE ALSO

`pthread`(3C), `pthread_spec`(3C), `pthread_mutex`(3C)

NAME

pthread_mutexattr_init, pthread_mutexattr_destroy, pthread_mutexattr_setkind_np, pthread_mutexattr_getkind_np, pthread_mutex_init, pthread_mutex_destroy, pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock – Mutual exclusion

SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_init (pthread_mutexattr_t*attr);

int pthread_mutexattr_destroy (pthread_mutexattr_t*attr);

int pthread_mutexattr_setkind_np (pthread_mutexattr_t*attr, int kind);

int pthread_mutexattr_getkind_np (const pthread_mutexattr_t *attr);

int pthread_mutex_init (pthread_mutex_t*mutex,
const pthread_mutexattr_t*attr);

int pthread_mutex_destroy (pthread_mutex_t*mutex);

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_lock (pthread_mutex_t*mutex);

int pthread_mutex_trylock (pthread_mutex_t*mutex);

int pthread_mutex_unlock (pthread_mutex_t*mutex);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

PThreads

DESCRIPTION

The pthread_mutexattr_init function initializes the specified mutex attributes object to the default values.

The pthread_mutexattr_destroy function should be called when the attributes object is no longer needed in order to release any system resources associated with the object. The attributes object can be destroyed even when there are active mutexes created with that attributes object. Once destroyed, the attributes object cannot be used.

The pthread_mutexattr_setkind_np and pthread_mutexattr_getkind_np functions set or get the mutex type. The mutex type can be MUTEX_FAST_NP, MUTEX_NONRECURSIVE_NP, or MUTEX_RECURSIVE_NP. A mutex with type MUTEX_FAST_NP is a simple mutex lock which does no error checking. A mutex with type MUTEX_NONRECURSIVE_NP provides additional error checking. Both of these mutexes will block if the owner of the mutex attempts to lock the mutex a second time. If, instead, attempts to lock the mutex should be nested, then the mutex can be initialized as a MUTEX_RECURSIVE_NP mutex. The default mute type is MUTEX_FAST_NP.

The pthread_mutex_init function initializes the mutex referenced by *mutex* with attributes specified by *attr*. If *attr* is a null value, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object. On successful initialization, the state of the mutex becomes initialized and unlocked. Attempting to initialize an already initialized mutex results in undefined behavior.

The pthread_mutex_destroy function destroys the mutex object referenced by *mutex*; the mutex object becomes effectively uninitialized. A destroyed mutex object can be reinitialized using pthread_mutex_init; the results of otherwise referencing the object after it has been destroyed are undefined.

It is invalid to destroy a locked mutex.

For cases in which default mutex attributes are appropriate, the macro PTHREAD_MUTEX_INITIALIZER can be used to initialize mutexes that are statically allocated. This is equivalent to dynamic initialization by a call to pthread_mutex_init, with parameter *attr* specified as a null value, except that no error checks are performed.

The *mutex* object is locked by a call to pthread_mutex_lock. If the mutex is already locked, the calling thread blocks until the mutex becomes available. The operation returns with *mutex* in the locked state, with the calling thread as its owner.

The function pthread_mutex_trylock is identical to pthread_mutex_lock except that, if *mutex* is currently locked (by any thread including the current thread), the call returns immediately.

The pthread_mutex_unlock function is called by the owner of *mutex* to release it. It is invalid to call pthread_mutex_unlock from a thread that is not the owner of the mutex. Calling pthread_mutex_unlock when *mutex* is unlocked is also invalid. If there are threads blocked on *mutex* when pthread_mutex_unlock is called, the mutex becomes available, and the scheduling policy is used to determine which thread shall acquire the mutex.

RETURN VALUES

If successful, the pthread_mutexattr_init, pthread_mutexattr_destroy, and pthread_mutexattr_setkind_np functions return 0. Otherwise, an error number indicates the error.

The pthread_mutexattr_getkind_np function returns the mutex type of the specified attributes object.

If successful, the `pthread_mutex_init` and `pthread_mutex_destroy` functions return 0. Otherwise, an error number indicates the error.

If successful, the `pthread_mutex_lock` and `pthread_mutex_unlock` functions return 0. Otherwise, an error number is returned to indicate the error.

The function `pthread_mutex_trylock` returns 0 if a lock on *mutex* is acquired. Otherwise, an error number indicates the error.

MESSAGES

If any of the following conditions occur, the `pthread_mutexattr_init` function returns the corresponding error number:

`ENOMEM` The system lacked the necessary resources (other than memory) to initialize the mutex attributes object.

If any of the following conditions occur, the `pthread_mutexattr_setkind_np` function returns the corresponding error number:

`EINVAL` The mutex type specified by *kind* is invalid.

If any of the following conditions occur, the `pthread_mutex_init` function returns the corresponding error number:

`EAGAIN` The system lacked the necessary resources (other than memory) to initialize another mutex.

`ENOMEM` Insufficient memory exists to initialize the mutex.

If the following condition occurs, the `pthread_mutex_trylock` function returns the corresponding error number:

`EINVAL` The mutex could not be acquired because it was already locked.

SEE ALSO

`pthread`(3C), `pthread_cond`(3C), `pthread_spec`(3C)

NAME

pthread_key_create, pthread_key_delete, pthread_setspecific, pthread_getspecific – Thread-specific data

SYNOPSIS

```
#include <pthread.h>
int pthread_key_create (pthread_key_t *key, void (*destructor) (void *));
int pthread_key_delete (pthread_key_t key);
int pthread_setspecific (pthread_key_t key, const void *value);
void *pthread_getspecific (pthread_key_t key);
```

IMPLEMENTATION

Cray PVP systems systems

STANDARDS

PThreads

DESCRIPTION

The pthread_key_create function creates a thread-specific data key visible to all threads in the process. Key values provided by pthread_key_create are opaque objects used to locate thread-specific data. Although the same key value may be used by different threads, the values bound to the key by pthread_setspecific are maintained on a per-thread basis and persist for the life of the calling thread.

When a key is created, it is given a null value in all active threads. When a thread is created, all defined keys in the new thread are given null values.

An optional destructor function may be associated with each key value. At thread exit, if a key value has a non-null destructor pointer, and the thread has a non-null value associated with that key, the function pointed to is called with the current associated value as its sole argument. The order of destructor calls is unspecified if more than one destructor exists for a thread when it exits.

The pthread_key_delete function deletes a thread-specific data key previously returned by pthread_key_create. The thread-specific data values associated with *key* need not be null when pthread_key_delete is called. The application must free storage or perform cleanup actions for data structures related to the deleted key or associated thread-specific data in any threads; this cleanup can be done either before or after pthread_key_delete is called. The specified *key* cannot be used following the call to pthread_key delete. No destructor functions are invoked by pthread_key_delete.

The pthread_setspecific function associates a thread-specific value with a *key* obtained via a previous call to pthread_key_create. Different threads may bind different values to the same key. The pthread_setspecific function cannot be called from a destructor function, as this may result in lost storage or infinite loops.

The pthread_getspecific function returns the value currently bound to the specified *key* on behalf of the calling thread.

RETURN VALUES

The function pthread_getspecific returns the thread-specific data value for *key*. If *key* has no thread-specific data value, a null value is returned.

If successful, the pthread_setspecific, pthread_key_create, and pthread_key_delete functions return 0. Otherwise, an error number indicates the error.

MESSAGES

If any of the following conditions occur, the pthread_key_create function returns the corresponding error number:

EAGAIN The system lacked the necessary resources to create another thread-specific data key, or the system-imposed limit on the total number of keys per process, PTHREAD_KEYS_MAX, has been exceeded.

ENOMEM Insufficient memory exists to create the key.

If any of the following conditions occur, the pthread_key_delete function returns the corresponding error number:

EINVAL The *key* value is invalid.

If any of the following conditions occur, the pthread_setspecific function returns the corresponding error number:

ENOMEM Insufficient memory exists to associate the value with the *key*.

EINVAL The *key* value is invalid.

SEE ALSO

pthread(3C), pthread_cond(3C), pthread_mutex(3C)

NAME

putc, putchar, fputc, putc_unlocked, putchar_unlocked, putw, fputwc, putwc, putwchar – Puts a character or word on a stream

SYNOPSIS

```
#include <stdio.h>
int fputc (int c, FILE *stream);
int putc (int c, FILE *stream);
int putchar (int c);
int putc_unlocked (int c, FILE *stream);
int putchar_unlocked (int c);
int putw (int w, FILE*stream);
#include <wchar.h>
wint_t fputwc (wint_t wc, FILE *stream);
wint_t putwc (wint_t wc, FILE *stream);
wint_t putwchar (wint_t wc);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (fputc, putc, and putchar only)
POSIX (putc_unlocked and putchar_unlocked only)
XPG4 (putw, fputwc, ungetwc, and putwchar only)

DESCRIPTION

The fputc function writes the character specified by c (converted to an unsigned char) to the output stream pointed to by stream, at the position indicated by the associated file position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream.

The fputc function behaves like putc, but it runs more slowly than putc, and it takes less space per invocation.

The putc function is equivalent to fputc, except that if it is implemented as a macro, it may evaluate stream more than once, so the argument should never be an expression with side effects. In particular,

```
putc(c,*f++)
```

does not work as expected; use fputc instead.

The putchar function is equivalent to putc with the second argument stdout.

The putc_unlocked and putchar_unlocked functions provide functionality equivalent to the putc and putchar functions, respectively. However, these interfaces are not guaranteed to be locked with respect to concurrent standard I/O operations in a multitasked application. Thus you should use these functions only within a scope protected by the flockfile(3C) or ftrylockfile(3C) functions.

The putw function writes the word (that is, type int) *w* to the output *stream* (at the position at which the file pointer, if defined, is pointing). The size of a word is the size of a type int and varies from machine to machine. The putw function neither assumes nor causes special alignment in the file.

The fputwc function writes the character corresponding to the wide-character code *wc* to the output stream to which *stream* points, at the position indicated by the associated file-position indicator for the stream (if defined), and it advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened while writing the character, the shift state of the output file is left in an undefined state. The st_ctime and st_mtime fields of the file are marked for update between the successful execution of fputwc and the next successful completion of a call to fflush(2) or fclose(2) on the same stream or a call to exit or abort.

The putwc function is equivalent to fputwc, except that if it is implemented as a macro it may evaluate *stream* more than once, so the argument should never be an expression with side effects. Therefore, you should not use this function; use the fputwc() function instead.

The function call putwchar(*wc*) is equivalent to putwc(*wc, stdout*).

CAUTIONS

Because of possible differences in word length and byte ordering, files written using putw are machine-dependent, and they may not be read using getw on a different processor; therefore, avoid using putw.

NOTES

The macro version of the putc function is not multitask protected. To obtain a multitask protected version, compile your code by using -D_MULTIP_ and link by using /lib/libcm.a.

RETURN VALUES

If successful, these functions each return the value they have written. If a write error occurs, the error indicator for the stream is set and the functions return EOF (WEOF for fputwc). The latter occurs if the file *stream* is not open for writing, or if the output file cannot be created. Because EOF is a valid integer, use the ferror(3C) function to detect putw errors.

SEE ALSO

fclose(3C), ferror(3C), fopen(3C), fread(3C), printf(3C), puts(3C), setbuf(3C)

NAME

putenv – Changes or adds value to the environment

SYNOPSIS

```
#include <stdlib.h>
int putenv (const char *string);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

The putenv function makes the value of the environment variable name equal to *value* by altering an existing variable or creating a new one. In either case, the string to which *string* points becomes part of the environment, so altering the string changes the environment. The *string* argument points to a string of the form "name=value" that contains no embedded blanks. The space that *string* uses is no longer used after a new string-defining name is passed to putenv.

NOTES

On Cray MPP systems, each processing element (PE) gets a separate copy of the environment; therefore, alterations to the environment by using putenv on a single PE will not be reflected on other PEs.

The putenv function manipulates the environment to which sh(1)points, and it can be used in conjunction with the getenv(3C) function; however, *envp* (the third argument to main) is not changed.

This function uses malloc(3C) to enlarge the environment.

After putenv is called, environmental variables are not necessarily in alphabetical order.

Calling putenv with an automatic variable as the argument, then exiting the calling function while *string* is still part of the environment is a potential error.

RETURN VALUES

If putenv could not obtain enough space (using malloc(3C)) for an expanded environment, it returns a nonzero value; otherwise, it returns 0.

FORTRAN EXTENSIONS

You also may call the putenv function from Fortran programs, as follows:

```
INTEGER*8 PUTENV, I
CHARACTER *n string
I = PUTENV(string)
```

Argument *string* can be either a Fortran character variable or an integer variable of the form *name=value*. If you use an integer variable, the data must be packed 8 characters per word and terminated with a null (0) byte.

Fortran function PUTENV allocates space and copies *string* to that space. Therefore, altering *string* after calling PUTENV does not change the environment.

SEE ALSO

getenv(3C), malloc(3C), setenv(3C)

sh(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

exec(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

putpwent – Writes password file entry

SYNOPSIS

```
#include <pwd.h>
int putpwent (struct passwd *p, FILE *f);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

AT&T extension

DESCRIPTION

The putpwent function is the inverse of getpwent. Given a pointer to a passwd structure created by getpwent (or getpwuid or getpwnam), putpwent writes a line on the stream *f*, which must match the format of /etc/passwd (see passwd(5)).

NOTES

This function is included only for compatibility with previous systems. Writing something in /etc/passwd does not make an entry in the user information database and so is ineffective. The passwd file is automatically maintained by udbgen(8).

WARNINGS

The preceding function uses the header stdio.h, which causes it to increase the size of programs more than otherwise might be expected.

RETURN VALUES

If an error is detected during its operation, putpwent returns nonzero; otherwise, it returns 0.

SEE ALSO

getpwent(3C)

udbgen(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

passwd(5), udb(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

puts, fputs, fputws – Puts a string on a stream

SYNOPSIS

```
#include <stdio.h>
int puts (const char *s);
int fputs (const char *s, FILE *stream);
#include <wchar.h>
int fputws(const wchar_t *ws, FILE *stream);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (puts and fputs only)
XPG4 (fputws only)

DESCRIPTION

The puts function writes the null-terminated string to which *s* points, followed by a newline character, to the standard output stream stdout. The fputs function writes the null-terminated string to which *s* points to the specified output *stream*. Neither function writes the terminating null character.

The fputws function writes a character string that corresponds to the (null-terminated) wide-character string to which *ws* points to the stream to which *stream* points. No character that corresponds to the terminating null, wide-character code is written. The st_ctime and st_mtime fields of the file are marked for update between execution of fputws and completion of the next call to fflush(2) or fclose(2) on the same stream or a call to exit or abort.

NOTES

The puts function appends a newline character; fputs does not.

RETURN VALUES

These functions return a nonnegative value. The puts and fputs functions return an end of file (EOF) on error (if they try to write on a file that has not been opened for writing). The fputws function, on an error, returns – 1, sets an error indicator for the stream, and sets errno to indicate the error.

FORTRAN EXTENSIONS

You also can call the `fputs` function from Fortran programs, as follows:

```
INTEGER*8 FPUTS, stream, I
I = FPUTS(s, stream)
```

Argument *s* must be left-justified, word-aligned, and terminated by a null byte.

SEE ALSO

`ferror`(3C), `fopen`(3C), `fread`(3C), `printf`(3C), `putc`(3C)

NAME

qsort – Performs sort

SYNOPSIS

```
#include <stdlib.h>

void qsort (void *base, size_t nmemb, size_t size, int (*compar)(const void *,
const void *));
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The qsort function sorts an array of *nmemb* objects, the initial element of which is pointed to by *base*. The *size* argument specifies the size of each object.

The contents of the array are sorted into ascending order according to a comparison function to which *compar* points, which is called with two arguments that point to the objects being compared. The function returns an integer that is less than, equal to, or greater than 0 if the first argument is considered to be respectively less than, equal to, or greater than the second.

If two elements compare as equal, their order in the sorted array is unspecified.

NOTES

The comparison function's arguments should be of type void* and should be cast back to type pointer-to-element within the function. The comparison function need not compare every byte; therefore, arbitrary data may be contained in the elements in addition to the values being compared.

The output order of two items that compare as equal is unpredictable.

RETURN VALUES

The qsort function returns no value.

EXAMPLES

The following example shows how the qsort function executes:

```
#include <stdlib.h>

struct element {        /* array of elements to be sorted */
        int key;        /* key to sort on */
        .
```

```
        .
        .
} q[nel];

struct element *base = &q[0];

int compar(const void *a, const void *b)
                        /* comparsion function for qsort() */
{
        return (((struct element *)a)->key - ((struct element *)b)->key);
}

main() {
         .
         .
         .
         qsort(base, nel, sizeof(*base), compar);
         .
         .
         .
}
```

SEE ALSO

bsearch(3C), lsearch(3C)

NAME

raise – Sends a signal to the executing program

SYNOPSIS

```
#include <signal.h>
int raise (int sig);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The raise function sends the signal *sig* to the executing program.

RETURN VALUES

The raise function returns 0 if successful, nonzero if unsuccessful.

NAME

rand, srand, rand_r – Generates pseudo-random integers

SYNOPSIS

```
#include <stdlib.h>
int rand (void);
void srand (unsigned int seed);
int rand_r (unsigned int *seedptr);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI (rand and srand)
PThreads (rand_r)

DESCRIPTION

The rand function computes a sequence of pseudo-random integers in the range 0 to RAND_MAX, which is defined in the header file stdlib.h.

The srand function uses the *seed* argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to rand. If srand is then called with the same seed value, the sequence of pseudo-random numbers is repeated. If rand is called before any calls to srand have been made, the same sequence is generated as when srand is first called with a seed value of 1.

The rand_r function provides functionality equivalent to the rand function but with an interface that is safe for multitasked applications. It takes a pointer to the seed value (*seedptr*) as an argument. This function allows for easy maintenance of separate random number generators and safe management of random number sequences for multitasked applications.

RETURN VALUES

The rand and rand_r functions return a pseudo-random integer.

The srand function returns no value.

SEE ALSO

drand48(3C)

NAME

rcmd, rresvport, ruserok – Returns a stream to a remote command

SYNOPSIS

```
#include <unistd.h>

int rcmd (char **ahost, unsigned short inport, char *locuser, char *remuser, char
*cmd, int fd2p);

int rresvport (int *port);

int ruserok (char *rhost, int superuser, char *ruser, char *luser);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The rcmd function allows the super user to execute a remote command *cmd* on a remote machine, using an authentication scheme based on reserved port numbers. The rresvport function returns a descriptor to a socket with an address in the privileged port space. Servers on the local host use the ruserok function to authenticate users on a remote host who request service by means of the rcmd function. All three functions are present in the same file and are used by the rshd(8) server (among others).

The rcmd function uses the gethostbyname function (see gethost(3C)), to look up the host *ahost*, returning –1 if the host does not exist. Otherwise, *ahost* is set to the official name of the remote host, and a connection is established to a server residing at the Internet port *inport*.

If the rcmd function succeeds, a socket of type SOCK_STREAM is returned to the caller and given to the remote command *cmd* as the file descriptors stdin (for reading from the socket) and stdout (for writing to the socket). If *fd2p* is nonzero, an auxiliary channel to a control process is set up, and a descriptor for it is placed in *fd2p*. The control process returns diagnostic output from the command (stderr) on this channel and also accepts bytes on this channel as being UNICOS signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, stderr is made the same as stdout. In this case, no provision is made for sending arbitrary signals to the remote process, although you may be able to establish contact by using out-of-band data.

The service request protocol and user validation are described in detail in rshd(8).

The rresvport function obtains a socket with a privileged address bound to it. This socket is suitable for use by rcmd and several other functions. Privileged addresses consist of a port in the range 0 through 1023. Only the super user is allowed to bind an address of this sort to a socket.

The ruserok function takes a remote host's name (*rhost*), as returned by the gethostent function (see gethost(3C)), the name of the remote user (*ruser*), the name of the local user (*luser*) whose account will be accessed by the remote user, and a flag (either 0 or 1) indicating if the local user's name is that of the super user. It then checks the local host's /etc/hosts.equiv file and the .rhosts file, if it exists, in the current directory (normally the local user's home directory) for authorization for the person requesting service.

NOTES

There is no way to specify options to the socket(2) call that rcmd makes.

RETURN VALUES

A 0 is returned if the name of the remote host is listed in the /etc/hosts.equiv file, or if the remote host name and remote user name are listed in the .rhosts file. The system configuration can require the /etc/hosts.equiv and .rhosts files each to contain a match for the remote host, and also require the remote user and local user names to match; otherwise, ruserok returns a –1. If the super user flag is 1, indicating that the specified local user is the super user, the checking of /etc/host.equiv is bypassed. (See hosts.equiv(5) and rhosts(5).) Also, if the rhosts file is writable by group or other, ruserok returns a –1.

FILES

/etc/hosts.equiv

$HOME/.rhosts

SEE ALSO

gethost(3C), rexec(3C)

remsh(1B) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

socket(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

hosts.equiv(5), rhosts(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

rexecd(8), rshd(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

TCP/IP Network User's Guide, Cray Research publication SG–2009

NAME

rcmdexec – Returns a stream to a remote command

SYNOPSIS

```
int rcmdexec (char **ahost, unsigned short rshellp, unsigned short rexecp, char
*locuser, char *remuser, char *passwd, char *cmd, int fd2p);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The rcmdexec function is a combination of the rcmd function and the rexec function. (See rcmd(3C) and rexec(3C) for additional argument information.)

Argument *rshellp* is the *inport* value for function rcmd, and *rexecp* is the *inport* value for function rexec.

The rcmdexec function first attempts a rcmd function call. If the internal connect(2) call fails with error ECONNREFUSED, rcmdexec immediately tries an rexec function call. If the internal connect(2) call again fails with error ECONNREFUSED, rcmdexec sleeps for a period of time, and goes back and tries both calls again. The total time out period is about 30 seconds.

SEE ALSO

rcmd(3C), rexec(3C)

connect(2), intro(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

rexecd(8), rshd(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

re_comp, re_exec – Matches regular expressions

SYNOPSIS

```
#include <unistd.h>
char *re_comp (char *s);
int re_exec (char *s);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The re_comp function compiles a string into an internal form suitable for pattern matching. The re_exec function checks the argument string against the last string passed to re_comp.

If string *s* was compiled successfully, function re_comp returns 0; otherwise, a string that contains an error message is returned. If re_comp is passed 0 or a null string, it returns without changing the currently compiled regular expression.

If string *s* matches the last compiled regular expression, function re_exec returns 1; if string *s* failed to match the last compiled regular expression, it returns 0; if the compiled regular expression was not valid (indicating an internal error), it returns –1.

The strings passed to both re_comp and re_exec can have trailing or embedded newline characters; they are terminated by a null character. Taking account of these differences, the regular expressions recognized are described in the ed(1) man page.

RETURN VALUES

Function re_exec returns –1 for an internal error.

If an error occurs, function re_comp returns one of the following strings:

```
No previous regular expression
Regular expression too long
unmatched \(
missing ]
too many \(\) pairs
unmatched \)
```

SEE ALSO

ed(1), egrep(1), ex(1), fgrep(1), grep(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

regcmp, regex – Compiles and executes a regular expression

SYNOPSIS

```
#include <stdlib.h>
char *regcmp (char *string, ...);
char *regex (char *re, char *subject, ...);
char *__loc1;
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

AT&T extension

DESCRIPTION

The regcmp function compiles a regular expression and returns a pointer to the compiled form. The malloc(3C) function creates space for the compiled regular expression; to free the allocated space, you must call the free (see malloc(3C)) function; regcmp returns NULL if if an incorrect argument is passed. regcmp returns NULL. The regcmp(1) command has been written to generally preclude the need for this function at execution time.

The regex function executes a compiled pattern against the subject string. Additional arguments are passed to receive values back. On failure, regex returns NULL; on success, it returns a pointer to the next unmatched character.

Functions regcmp and regex take a variable number of char * arguments following *string* and *subject*, respectively. The last argument to regcmp must be (char *)0. A global character pointer, __loc1, points to where the match began. Both regcmp and regex were mostly borrowed from the editor, ed(1); however, the syntax and semantics have been changed slightly. The following are the valid symbols and their associated meanings:

Symbol	Description
[ ] * . ^	These symbols retain their current meaning (as in ed(1)).
$	Matches the end of the string; \n matches a newline character.
-	Within brackets, the minus means *through* (for example, [a-z] is equivalent to [abcd. . .xyz]). The - can appear as itself only if used as the first or last character (for example, the character class expression []- matches the characters] and -).
+	A regular expression followed by + means *one or more times*; for example, [0-9]+ is equivalent to [0-9][0-9]*.

{*m*} {*m*,} {*m*,*u*} Integer values enclosed in { } indicate the number of times the preceding regular expression will be applied. The value *m* is the minimum number, and *u* is a number, less than 256, which is the maximum. If only *m* is present (for example, {*m*}), it indicates the exact number of times the regular expression will be applied. The value {*m*,} is analogous to {*m*,infinity}. The plus (+) and star (*) operations are equivalent to {1,} and {0,}, respectively.

(. . .)$*n* The value of the enclosed regular expression is to be returned. The value is stored in the (*n*+1)th argument following the subject argument. At most, 10 enclosed regular expressions are allowed. The regex function makes its assignments unconditionally.

(. . .) Parentheses are used for grouping. An operator, such as "*", "+", "-", or "/", can work on a single character or a regular expression enclosed in parentheses; for example, (a*(cb+)*)$0.

By necessity, all of the preceding symbols are special; to be used as themselves, you must escape them by using a .

CAUTIONS

The user program may run out of memory if regcmp is called iteratively without freeing the compiled regular expressions no longer required.

EXAMPLES

Example 1: The following example matches a leading newline character in the subject string at which cursor points.

```
#include <stdlib.h>

char *cursor, *newcursor, *ptr;
      .
      .
      .
newcursor = regex((ptr = regcmp("^\n", (char *)0)), cursor);
free(ptr);
```

Example 2: The following example matches through the string "Testing3" and returns the address of the character after the last matched character (cursor+11). The string "Testing3" is copied to the character array ret0.

```
#include <stdlib.h>

char ret0[9];
char *newcursor, *name;
      .
      .
      .
name = regcmp("([A-Za-z][A-za-z0-9_]{0,7})$0", (char *)0);
newcursor = regex(name, "123Testing321", ret0);
```

Example 3: The following example applies a precompiled regular expression `name` in `file.i` (see `regcmp`(1)) against `string`.

```
#include <stdlib.h>
#include "file.i"

char *string, *newcursor;
      .
      .
      .
newcursor = regex(name, string);
```

SEE ALSO

`malloc`(3C)

`ed`(1), `regcmp`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

regcomp, regexec, regerror, regfree – Regular-expression library

SYNOPSIS

```
#include <sys/types.h>
#include <regex.h>

int regcomp(regex_t *preg, const char *pattern, int cflags);

int regexec(const regex_t *preg, const char *string, size_t nmatch,
   regmatch_t pmatch[], int eflags);

size_t regerror(int errcode, const regex_t *preg, char *errbuf,
   size_t errbuf_size);

void regfree(regex_t *preg);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

DESCRIPTION

These functions implement regular expressions (REs); see regex(3C). Functions are used as follows:

Function	Description
regcomp	Compiles an RE written as a string into an internal form
regexec	Matches the regcomp value against a string and reports results
regerror	Transforms error codes from either regcomp or regexec into human-readable messages
regfree	Frees any dynamically allocated storage used by the internal form of an RE

The header file regex.h declares two structure types, regex_t and regmatch_t, the former for compiled internal forms and the latter for match reporting. It also declares the four functions, a type regoff_t, and a number of constants with names that start with REG_.

The regcomp function compiles the regular expression contained in the *pattern* string, subject to the flags in *cflags*, and places the results in the regex_t structure to which *preg* points. The cflags variable is the bitwise OR of 0 or more of the following flags:

Function	Description
REG_EXTENDED	Compile modern (extended) REs, rather than the obsolete (basic) REs that are the default.
REG_ICASE	Compile for matching that ignores upper/lower case distinctions. See regex(7).
REG_NOSUB	Compile for matching that must report only success or failure, not what was matched.

REG_NEWLINE	Compile for newline-sensitive matching. By default, newline is a completely ordinary character with no special meaning in either REs or strings. With this flag, [^ bracket expressions and . never match newline, a ^ anchor matches the null string after any newline in the string in addition to its normal function, and the $ anchor matches the null string before any newline in the string in addition to its normal function.
REG_WORDS	Compile for matching that treats < as the beginning of a word and > as the end of a word.

The regcomp function returns 0 and fills in the structure to which *preg* points. One member of that structure is publicized: re_nsub, of type size_t, contains the number of parenthesized subexpressions within the RE (except this member's value is undefined if the REG_NOSUB flag was used). If regcomp fails, it returns a nonzero error code; see ERROR CODES.

The regexec function matches the compiled RE to which *preg* points against the *string*, subject to the flags in *eflags*, and reports results by using *nmatch*, *pmatch*, and the returned value. The RE must have been compiled by a previous invocation of regcomp. The compiled form is unchanged during execution of regexec; therefore, one compiled RE can be used simultaneously by multiple threads.

By default, the null-terminated string to which *string* points is considered to be the text of an entire line, minus any terminating newline. The *eflags* argument is the bitwise OR of 0 or more of the following flags:

Flag	**Description**
REG_NOTBOL	The first character of the string is not the beginning of a line, so the ^ anchor should not match before it. This condition does not affect the behavior of newlines under REG_NEWLINE.
REG_NOTEOL	The null terminating the string does not end a line, so the $ anchor should not match before it. This condition does not affect the behavior of newlines under REG_NEWLINE.
REG_STARTEND	The string is considered to start at *string* + *pmatch*[0].*rm_so* and to have a terminating NUL located at *string* + *pmatch*[0].*rm_eo* (NUL is not required at that location), regardless of the value of *nmatch*. The *pmatch* and *nmatch* variables are defined below. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be ported to other systems.

For a discussion of what is matched in cases in which an RE or a portion thereof could match any of several substrings of *string*, see regex(3C).

Usually, regexec returns 0 for success and the nonzero code REG_NOMATCH for failure. Other nonzero error codes may be returned in exceptional situations; see ERROR CODES.

If REG_NOSUB was specified in the compilation of the RE, or if *nmatch* is 0, regexec ignores the *pmatch* argument. Otherwise, *pmatch* points to an array of *nmatch* structures of type *regmatch_t*. Such a structure has at least the members *rm_so* and *rm_eo*, both of type *regoff_t* (a signed arithmetic type at least as large as an *off_t* and a *ssize_t*), containing respectively the offset of the first character of a substring and the offset of the first character after the end of the substring. Offsets are measured from the beginning of the *string* argument given to regexec. An empty substring is denoted by equal offsets, both indicating the character following the empty substring.

The 0th member of the *pmatch* array is filled in to indicate the substring of *string* that was matched by the entire RE. Remaining members report the substring that was matched by parenthesized subexpressions within the RE; member *i* reports subexpression *i*, with subexpressions counted (starting at 1) by the order of their opening parentheses in the RE, left to right. The *rm_so* and *rm_eo* arguments are set to –1 for unused entries in the array (entries that correspond either to subexpressions not used in the match, or to subexpressions that are not in the RE (that is, *i* > *preg* –> *re_nsub*). If a subexpression is used in the match several times, the reported substring is the last one it matched. When the RE `(b*)+` matches `bbb`, the parenthesized subexpression matches each of the three `b`s and then an infinite number of empty strings following the last `b`, so the reported substring is one of the empty strings.)

The `regerror` function maps a nonzero *errcode* from either `regcomp` or `regexec` to a printable message. If *preg* is nonnull, the error code should have arisen from use of the `regex_t` to which *preg* points, and if the error code came from `regcomp`, it should have been the result from the most recent `regcomp` using that `regex_t`. The `regerror` function may be able to supply a more detailed message by using information from the `regex_t`. The `regerror` function places the null-terminated message into the buffer to which *errbuf* points, limiting the length (including the null) to at most *errbuf_size* bytes. If the whole message does not fit, as much of it as will fit before the terminating null is supplied. In any case, the returned value is the size of buffer needed to hold the whole message (including terminating null). If *errbuf_size* is 0, *errbuf* is ignored, but the return value is still correct.

The `regfree` function frees any dynamically allocated storage associated with the compiled RE to which *preg* points. The remaining `regex_t` is no longer a valid compiled RE, and the effect of supplying it to `regexec` or *regerror* is undefined.

Implementation Choices

Many details are optional under POSIX, either explicitly undefined or forbidden by the RE grammar. The Cray Research implementation treats them as follows. For a discussion of the definition of case-independent matching, see `regex`(3C).

- No particular limit exists on the length of REs, except insofar as memory is limited. Memory usage is approximately linear in RE size, and largely insensitive to RE complexity, except for bounded repetitions. See BUGS for one short RE using them that will run almost any system out of memory.
- Any backslashed character other than the ones specifically legitimized by POSIX produces a `REG_EESCAPE` error.
- Any unmatched `[` is a `REG_EBRACK` error.
- Equivalence classes cannot begin or end bracket-expression ranges. The endpoint of one range cannot begin another.
- `RE_DUP_MAX`, the limit on repetition counts in bounded repetitions, is 255.
- A repetition operator (`?`, `*`, `+`, or bounds) cannot follow another repetition operator. A repetition operator cannot begin an expression, or subexpression, or follow `^` or `|`.
- A `|` cannot appear first or last in a (sub)expression or after another `|` (for example, an operand of `|` cannot be an empty subexpression). An empty parenthesized subexpression, `()`, is legal and matches an empty (sub)string. An empty string is not a legal RE.

- A { followed by a digit is considered the beginning of bounds for a bounded repetition, which must then follow the syntax for bounds. A { **not** followed by a digit is considered an ordinary character.
- A ^ and $ beginning and ending subexpressions in obsolete (basic) REs are anchors, not ordinary characters.

NOTES

One known functionality bug exists. The implementation of internationalization is incomplete; the locale is always assumed to be the POSIX default, and only the collating elements and other such elements of that locale are available.

ERROR CODES

Nonzero error codes from `regcomp` and `regexec` include the following:

Error Code	Description
REG_BADBR	Repetition count(s) in { } not valid
REG_BADPAT	Regular expression not valid
REG_BADRPT	?, *, or + operand not valid
REG_EBRACE	Braces { } not balanced
REG_EBRACK	Brackets [ ] not balanced
REG_ECOLLATE	Collating element not valid
REG_ECTYPE	Character class not valid
REG_EESCAPE	A \ applied to unescapable character
REG_EFATAL	Internal error
REG_ENEWLINE	A \n found before end of pattern
REG_ENOSYS	Function not supported
REG_ENSUB	More than nine ( ) pairs
REG_EPAREN	Parentheses ( ) not balanced
REG_ERANGE	Character range in [ ] not valid
REG_ESPACE	Ran out of memory
REG_STACK	Backtrack stack overflow
REG_ESUBREG	Backreference number not valid
REG_NOMATCH	regexec() failed to match

SEE ALSO

regex(3C)

grep(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

regexp.h – Library header for regular expression compile and match functions

IMPLEMENTATION

All Cray Research systems

STANDARDS

XPG4

MACROS

None

TYPES

None

FUNCTION DECLARATIONS

The following functions are declared in regexp.h:

```
char *compile (char*instring, char*expbuf, const char *endbuf, int eof);
int advance (const char *string, const char *expbuf);
int step (const char *string, char *expbuf);
```

and the following are declared as external variables:

```
extern char *loc1, *loc2, *locs;
```

DESCRIPTION

These general-purpose regular expression matching functions in the form of ed(1), are defined in the header file regexp.h. Programs such as ed(1), sed(1), grep(1), expr(1), and so on, which perform regular expression matching, use this source file. In this way, only this file need be changed to maintain regular expression compatibility.

The interface to the header file regexp.h is excessively complex. Programs that include this file must have the following five macros declared before the #include <regexp.h> statement. These macros are used by the compile function.

Macro	Description
GETC( )	Returns the value of the next character in the regular expression pattern. Successive calls to GETC( ) should return successive characters of the regular expression.
PEEKC( )	Returns the next character in the regular expression. Successive calls to PEEKC( ) should return the same character (which should also be the next character returned by GETC( )).

`UNGETC(`*c*`)` Causes the argument *c* to be returned by the next call to `GETC( )` (and `PEEKC( )`). No more than one character of pushback is ever needed and this character is guaranteed to be the last character read by `GETC( )`. The value of the macro `UNGETC(`*c*`)` is always ignored.

`RETURN(`*pointer*`)` Used on normal exit of the `compile` function. The value of the argument *pointer* is a pointer to the character after the last character of the compiled regular expression. This macro is useful to programs that have to manage memory allocation.

`ERROR(`*val*`)` Used on abnormal return from the `compile` function. The argument *val* is an error number (see the following table for meanings). This call should never return.

Error	**Meaning**
11	Range endpoint too large
16	Bad number
25	''\ digit'' out-of-range
36	Illegal or missing delimiter
41	No remembered search string
42	`\( \)` imbalance
43	Too many `\(`
44	More than two numbers given in `\{ \}`
45	`}` expected after `\`
46	First number exceeds second in `\{ \}`
49	`[ ]` imbalance
50	Regular expression overflow

Arguments to the `compile` function are as follows:

Argument	**Description**
instring	Never used explicitly by the `compile` function but is useful for programs that pass down different pointers to input characters. It is sometimes used in the `INIT` declaration (see following explanation). Programs that call functions to input characters or have characters in an external array can pass down a value of `((char *) 0)` for this parameter.
expbuf	Character pointer to the place where the compiled regular expression will be placed.
endbuf	One more than the highest address where the compiled regular expression may be placed. If the compiled expression cannot fit in (*endbuf-expbuf*) bytes, a call to `ERROR(50)` is made.
eof	Character that marks the end of the regular expression; for example, in `ed`(1), this character is usually a `/`.

Each program that includes this file must have a `#define` statement for `INIT`. This definition is placed right after the declaration for the function `compile` and the opening brace ({). It is used for dependent declarations and initializations. Most often it is used to set a register variable to point to the beginning of the regular expression so that this register variable can be used in the declarations for `GETC( )`, `PEEKC( )`, and `UNGETC( )`. Otherwise, it can be used to declare external variables that might be used by `GETC( )`, `PEEKC( )`, and `UNGETC( )`. See the following example of the declarations taken from `grep`(1).

There are other functions in this file that perform actual regular expression matching, one of which is the function `step`.

Arguments to the `step` function are as follows:

Argument	Description
string	Pointer to a string of characters to be checked for a match. This string should be null terminated.
expbuf	Compiled regular expression that was obtained by a call of the function `compile`.

The `step` function returns 1, if the given string matches the regular expression, and 0 if the expressions do not match. If there is a match, two external character pointers are set as a side effect to the call to `step`. The variable set in `step` is *loc1*. This is a pointer to the first character that matched the regular expression.

The variable *loc2*, which is set by the function `advance`, points to the character after the last character that matches the regular expression. Thus, if the regular expression matches the entire line, *loc1* points to the first character of *string* and *loc2* points to the null at the end of *string*.

The `step` function uses the external variable *circf*, which is set by `compile` if the regular expression begins with `^`. If this is set, `step` only tries to match the regular expression to the beginning of the string. If more than one regular expression is to be compiled before the first is executed, the value of *circf* should be saved for each compiled expression and *circf* should be set to that saved value before each call to `step`.

The `advance` function is called from `step` with the same arguments as `step`. The purpose of `step` is to step through the *string* argument and call `advance` until `advance` returns 1 indicating a match or until the end of *string* is reached. If you want to constrain *string* to the beginning of the line in all cases, `step` need not be called, simply call `advance`.

When `advance` encounters an `*` or `\{ \}` sequence in the regular expression, it advances its pointer to the string to be matched as far as possible and recursively calls itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, `advance` backs up along the string until it finds a match or reaches the point in the string that initially matched the `*` or `\{ \}`. It is sometimes desirable to stop this backing up before the initial point in the string is reached. If the external character pointer *locs* is equal to the point in the string at sometime during the backing up process, `advance` breaks out of the loop that backs up and returns 0. This is used by `ed`(1) and `sed`(1) for substitutions done globally (not just the first occurrence, but the whole line); so, for example, expressions such as `s/y*//g` do not loop forever.

EXAMPLES

The following example shows how the regular expression macros and calls look from `grep`(1):

```
#define INIT        register char *sp = instring;
#define GETC( )     (*sp++)
#define PEEKC( )    (*sp)
#define UNGETC(c)   (--sp)
#define RETURN(c)   return;
#define ERROR(c)    regerr( )

#include <regexp.h>
 ...
          compile(*argv, expbuf, &expbuf[ESIZE], (int) '\0');
 ...
          if (step(linebuf, expbuf))
                   succeed( );
```

NOTES

Because `regexp.h` is a header file, no actual `libc` modules exist. This source code is provided for users wanting to use functions for regular expression work.

SEE ALSO

`ed`(1), `expr`(1), `grep`(1), `sed`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

remainder, remainderf, remainderl – Divides its arguments and returns the remainder

SYNOPSIS

```
#include <fp.h>

double remainder (double x, double y);
float remainderf (float x, float y);
long double remainderl (long double x, long double y);
```

IMPLEMENTATION

CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

ANSI/IEEE Std 754-1985
X3/TR-17:199x

DESCRIPTION

The remainder, remainderf, and remainderl functions compute the remainder *r* = *x* REM *y*. If *y* is not equal to 0, according to the IEEE floating-point standard, the remainder "is defined regardless of the rounding mode by the mathematical relation *r* = *x* - *y* * *n*, where *n* is the integer nearest the exact value of *x*/*y*; whenever |*n* - *x*/*y* | = *1/2*, then *n* is even. Thus, the remainder is always exact. If *r* = 0, its sign shall be that of *x*."

RETURN VALUES

These functions return the remainder of the first argument divided by the second argument.

SEE ALSO

Migrating to the CRAY T90 Series IEEE Floating Point, Cray Research publication SN–2194

NAME

remove – Removes files

SYNOPSIS

```
#include <stdio.h>
int remove (const char *file);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The remove function causes the file whose name is the string to which *file* points to become inaccessible by that name. A subsequent attempt to open the file by using that name fails, unless you create the file from scratch. If the file is open, the behavior of remove is implementation-defined. On Cray Research systems, the file remains accessible to the file descriptor (or stream).

If *file* does not specify a directory, remove(*file*) is equivalent to unlink(*file*). If *file* specifies a directory, remove(*file*) is equivalent to rmdir(*file*).

RETURN VALUES

If the operation succeeds, the remove function returns 0; if it fails, it returns a nonzero value.

SEE ALSO

rename(2), rmdir(2), unlink(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

`rename` – Renames a file

SYNOPSIS

```
#include <stdio.h>
int rename(const char *old, const char *new);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

ISO/ANSI

DESCRIPTION

The `rename` function changes the name of a file. The *old* argument points to the path name of the file to be renamed. The *new* argument points to the new path name of the file.

If the *old* argument and the *new* argument both refer to links to the same existing file, the `rename` function returns successfully and performs no other action.

If the *old* argument points to the path name of a file that is not a directory, the *new* argument does not point to the path name of a directory. If the link specified by the *new* argument exists, it is removed and *old* is renamed *new*. In this case, a link named *new* exists throughout the renaming operation and refers either to the file referred to by *new* or *old* before the operation began. Write access permission is required for both the directory that contains *old* and the directory that contains *new*.

If the *old* argument points to the path name of a directory, the *new* argument points to the path name of a file that is a directory. If the directory specified by the *new* argument exists, it is removed and *old* renamed *new*. In this case, a link named *new* exists throughout the renaming operation and refers either to the file referred to by *new* or *old* before the operation began. If *new* specifies an existing directory, it must be an empty directory.

The *new* path name does not contain a path prefix that names *old*. Write access permission is required for the directory that contains *old* and the directory that contains *new*. If the *old* argument points to the path name of a directory, write access permission is required for the directory named by *old*, and, if it exists, the directory named by *new*.

If the link specified by the *new* argument exists and the file's link count becomes 0 when it is removed and no process has the file open, the space occupied by the file is freed and the file is no longer accessible. If one or more processes have the file open when the last link is removed, the link is removed before `rename` returns, but the removal of the file contents is postponed until all references to the file are closed.

NOTES

Under UNICOS, rename(2) is implemented as a system call, but the rename function also is defined to be a part of the ANSI Standard C library. For this reason, this documentation appears both here and in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012.

RETURN VALUES

The rename function returns 0 if the operation succeeds and marks for update the st_ctime and st_mtime fields of the parent directory of each file. rename returns a nonzero if it fails; in which case, if the file existed previously, it is still known by its original name.

SEE ALSO

remove(3C)

rename(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

dn_comp, dn_expand, dn_skipname, hostalias, res_init, res_mkquery, res_query, res_querydomain, res_search, res_send – Provides domain name service resolver functions

SYNOPSIS

```
#include <sys/types.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_init (void);

int res_mkquery (int op, char *name, int class, int type, char *data, int datalen,
u_char *newrr, u_char *buf, int buflen);

int res_query (char *name, int class, int type, u_char *answer, int anslen);

int res_search (char *name, int class, int type, u_char *answer, int anslen);

int res_querydomain (char *name, char *domain, int class, int type, u_char
*answer, int anslen);

char *hostalias (char *name);

int res_send (u_char *buf, int buflen, u_char *answer, int anslen);

int dn_expand (u_char *msg, u_char *eomorig, u_char *comp_dn, char *exp_dn, int
length);

int dn_comp (char *exp_dn, char *comp_dn, int length, u_char **dnptrs, u_char
**lastdnptr);

int dn_skipname (unsigned char *comp_dn, unsigned char *eom);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The res_init function initializes the resolver functions, consulting /etc/resolv.conf (if it exists) for configuration information. The res_send(), res_query(), and res_search() routines automatically call res_init() if it has not yet been called.

The `res_query` function creates a standard query for the fully qualified domain name pointed to by *name*, of class *class* and type *type*, sends the query to the configured name server, and awaits an answer. The answer is placed in the buffer, of length *anslen*, pointed to by *answer*.

The `res_search` function calls `res_query` and `res_querydomain` to perform a search for the domain name *name* among the domains specified in the resolver configuration.

The `res_mkquery` function creates a query in the buffer, of length *buflen*, pointed to by *buf*. The query is formulated with the following characteristics: *op* is the opcode of the query; *name* points to the domain name to be queried; *class* is the class of the query; *type* is the type of query; *data* points to *datalen* bytes of associated data for the query; *newrr* points to an existing resource record associated with the query. (Legal values for *op*, *class*, and *type* can be found in the include file `arpa/nameser.h`.) This routine is typically called only by `res_query()`.

The `res_querydomain` function calls `res_query` to perform a lookup of the concatenation of the domain names pointed to by *name* and *domain*, and returns the value returned by `res_query`. This routine is typically called only by `res_search()`.

The `hostalias` function consults the environment variable `HOSTALIASES` for the name of a file that contains a list of domain names and aliases, and returns a pointer to the first alias found in such a file for the domain name pointed to by *name*.

The `res_send` function sends the query pointed to by *buf*, of length *buflen*, to the configured server or servers. It retrieves the answer in the buffer, of length *anslen*, pointed to by *answer*.

The `dn_expand` function expands the compressed domain name pointed to by *comp_dn* into the buffer of length *length*, pointed to by *exp_dn*.

The `dn_comp` function compresses the domain name pointed to by *exp_dn* into the buffer of length *length* pointed to by *comp_dn*. *dnptrs* points to a null-terminated list of pointers to previous compressed names; *lastdnptr* points to the actual end of the array pointed to by *dnptrs*.

The `dn_skipname` function skips over a compressed domain name.

RETURN VALUES

The `res_init` function returns 0 on successful initialization, or –1 on error.

The `res_query` function returns the length of answer received, or –1 on error, in which case it sets the external variable `h_errno` to reflect the type of error.

The `res_mkquery` function returns the size of the resulting query, or –1 on error.

The `hostalias` function returns `NULL` if the environment variable `HOSTALIASES` does not exist, the file referred to by `HOSTALIASES` cannot be opened, or no alias is found for *name*.

The `res_send` function returns the length of the answer received, or –1 on error.

The dn_expand function returns the length of the compressed name, or –1 on error.

The dn_comp function returns the length of the compressed name, or –1 on error.

The dn_skipname functions returns the size of the compressed name skipped, or –1 on error.

FILES

arpa/nameser.h

/etc/resolv.conf

SEE ALSO

gethost(3C), herror(3C)

resolv.conf(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

rexec – Returns a stream to a remote command

SYNOPSIS

```
#include <unistd.h>

int rexec (char **ahost, unsigned short inport, char *user, char *passwd, char
*cmd, int *fd2p);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The rexec function uses gethostbyname (see gethost(3C)) to look up the remote host *ahost*; rexec returns –1 if the remote host does not exist. Otherwise, *ahost* is set to the official name of the host. If a user name and password are both specified, these are used to determine whether authorization exists for the remote host; otherwise, the .netrc file in the user's home directory is searched for the appropriate information. If all searches fail, the user is prompted for a login name and password.

The port *inport* specifies which TCP port to use for the connection; it is normally the value returned from the function getservbyport (see getserv(3C)).

The protocol for the connection is described in detail in rexecd(8).

If the rexec function succeeds, a socket of type SOCK_STREAM is returned to the caller and given to the remote command *cmd* as the file descriptors stdin (for reading to the socket) and stdout (for writing to the socket). If *fd2p* is nonzero, an auxiliary channel to a control process is set up, and a descriptor for it is placed in *fd2p*. The control process returns diagnostic output from the command (stderr) on this channel and also accepts bytes on this channel as being UNICOS signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, stderr is made the same as stdout. In this case, no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

NOTES

There is no way to specify options to the socket(2) call that rexec makes.

FILES

`$HOME/.netrc`

SEE ALSO

`gethost`(3C), `getserv`(3C), `rcmd`(3C), `stdio.h`(3C)

`socket`(2) in *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`hosts`(5), `netrc`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`rexecd`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`rint`, `rintf`, `rintl` – Rounds arguments to an integral value in floating-point format

SYNOPSIS

```
#include <fp.h>

double rint (double x);
float rintf (float x);
long double rintl (long double x);
```

IMPLEMENTATION

CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

ANSI/IEEE Std 754-1985
X3/TR-17:199x

DESCRIPTION

The `rint`, `rintf`, and `rintl` functions round their arguments to an integral value in floating-point format, using the current rounding direction.

RETURN VALUES

These functions return the rounded integral value of their arguments.

SEE ALSO

`rinttol`(3C)

Migrating to the CRAY T90 Series IEEE Floating Point, Cray Research publication SN–2194

NAME

`rinttol` – Rounds a floating-point number to a long integer value

SYNOPSIS

```
#include <fp.h>
long int rinttol (long double x);
```

IMPLEMENTATION

CRAY T90 systems with IEEE floating-point arithmetic

STANDARDS

ANSI/IEEE Std 754-1985
X3/TR-17:199x

DESCRIPTION

The `rinttol` function rounds its `long double` argument to `long int`, using the current rounding direction. If the rounded value is outside the range of `long int`, the numeric result is unspecified.

RETURN VALUES

The `rinttol` function returns the rounded `long int` value, using the current rounding direction.

SEE ALSO

`rint`(3C)

Migrating to the CRAY T90 Series IEEE Floating Point, Cray Research publication SN–2194

NAME

rpc – Makes a remote procedure call

IMPLEMENTATION

All Cray Research systems

STANDARDS

BSD extension

DESCRIPTION

The remote procedure call (RPC) functions allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a data packet to the server. On receipt of the packet, the server calls a dispatch function to perform the requested service and sends back a reply. Finally, the procedure call returns to the client.

The RPC library functions follow:

Function	Description
authdes_create	Returns the RPC authentication handle with DES authentication
auth_destroy	Destroys the authentication information handle
authnone_create	Returns an RPC null authentication handle
authunix_create	Returns an RPC UNIX authentication handle
authunix_create_default	Returns the default UNIX authentication handle
callrpc	Calls a remote procedure, given [*prognum*,*versnum*,*procnum*]
clnt_broadcast	Broadcasts the remote procedure call
clnt_call	Calls the remote procedure associated with the client handle
clnt_create	Creates an RPC client when passed a remote host and transport type
clnt_destroy	Destroys the client's RPC handle
clnt_freeres	Frees data allocated by the RPC/XDR system when decoding results
clnt_geterr	Copies error information from the client handle to an error structure
clnt_pcreateerror	Prints a message to stderr that indicates why client handle creation failed
clnt_perrno	Prints a message that corresponds to the condition given to stderr
clnt_perror	Prints a message to stderr that indicates why the RPC call failed
clntraw_create	Creates a simple RPC client for simulation
clnttcp_create	Creates an RPC client by using TCP transport
clntudp_create	Creates an RPC client by using UDP transport
get_myaddress	Gets the machine's IP address
pmap_getmaps	Returns a list of RPC program-to-port mappings
pmap_getport	Returns the port number on which the supporting service waits
pmap_rmtcall	Instructs the portmapper to make an RPC call
pmap_set	Establishes mapping between [*prognum*,*versnum*,*procnum*] and *port*

`pmap_unset`	Destroys mapping between [*prognum*,*versnum*,*procnum*] and *port*
`registerrpc`	Registers a procedure with RPC as the service package
`svc_destroy`	Destroys the RPC service transport handle
`svc_freeargs`	Frees data allocated by the RPC/XDR system when decoding arguments
`svc_getargs`	Decodes the arguments of an RPC request
`svc_getcaller`	Gets the network address of the caller of a procedure
`svc_getreq`	Returns when all associated sockets have been serviced
`svc_getreqset`	Returns when all associated sockets have been serviced
`svc_register`	Associates *prognum* and *versnum* with a service dispatch procedure
`svc_run`	Waits for RPC requests to arrive and calls the appropriate service
`svc_sendreply`	Sends back results of a remote procedure call
`svc_unregister`	Removes mapping of [*prognum*,*versnum*] to dispatch functions
`svcerr_auth`	Refuses service because of an authentication error
`svcerr_decode`	Indicates that a service cannot decode its parameters
`svcerr_noproc`	Indicates that a service has not implemented the desired procedure
`svcerr_noprog`	Shows that a program is not registered with the RPC package
`svcerr_progvers`	Shows that a version is not registered with the RPC package
`svcerr_systemerr`	Indicates that a service detects a system error
`svcerr_weakauth`	Refuses service because of insufficient authentication
`svcraw_create`	Creates a simple RPC service transport for testing
`svctcp_create`	Creates an RPC service based on TCP transport
`svcudp_create`	Creates an RPC service based on UDP transport
`xdr_accepted_reply`	Generates RPC-style replies without using the RPC package
`xdr_authdes_cred`	Sends or receives DES credentials without using the RPC package
`xdr_authdes_verf`	Sends or receives DES verifier without using the RPC package
`xdr_authunix_parms`	Generates UNIX credentials without using the RPC package
`xdr_callhdr`	Generates RPC-style headers without using the RPC package
`xdr_callmsg`	Generates RPC-style messages without using the RPC package
`xdr_opaque_auth`	Describes RPC authenticators, externally
`xdr_pmap`	Describes parameters for portmap procedures, externally
`xdr_pmaplist`	Describes a list of port mappings, externally
`xdr_rejected_reply`	Generates RPC-style rejections without using the RPC package
`xdr_replymsg`	Generates RPC-style replies without using the RPC package
`xprt_register`	Registers RPC service transport with the RPC package
`xprt_unregister`	Unregisters RPC service transport from the RPC package

NOTES

Users access these library functions from `libc`.

FILES

`/lib/libc.a`	File that contains the RPC functions

SEE ALSO

`xdr`(3C)

Remote Procedure Call (RPC) Reference Manual, Cray Research publication SR–2089

"Remote Procedure Call Programming Guide," "Remote Procedure Call Protocol Specification," and "External Data Representation Specification" in Networking on the Sun Workstation, part #800–1324–03, Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94043.

RPC: Remote Procedure Call Version 2 Standard, RFC 1057

NAME

`rtclock` – Gets current real-time clock (RTC) reading

SYNOPSIS

```
#include <time.h>
long rtclock (void);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The `rtclock` function returns the current reading of the RTC.

SEE ALSO

`cpused`(3C)

NAME

SAMEQU – Specifies equivalent character in Sort/Merge session

SYNOPSIS

```
INTEGER init_array(2**N)
CALL SAMEQU(colseq, chr, init_array)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SAMEQU is used to make the characters specified in an array equivalent to one specific character in a specified collating sequence. The following is a list of valid arguments for this routine.

colseq Hollerith constant of 8 characters or less containing the name of the collating sequence.

chr A right-justified Hollerith character specifying the character that is equivalent to the characters in *init_array*.

init_array An integer array containing a set of characters that are considered equivalent to *chr* in the collating sequence *colseq* after the execution of SAMEQU. Each element of the array holds 1 Hollerith character that is right-justified and padded with zeros on the left. The maximum number of characters in *init_array* is 2^n, where *n* is the size in bits of a character. The default character size is 8. The list of characters in the array is terminated by an array element that contains the value –1.

After execution, the character values listed in *init_array* all compare equally to the specified character *chr* in the collating sequence *colseq*.

EXAMPLES

In the following example, the ASCII collating sequence is modified to make digits 1 through 9 equivalent to digit 0. After the execution of the call, digits 1 through 9 are treated in the same manner as digit 0 in the ASCII collating sequence by the Sort/Merge session.

```
INTEGER SAMPLE(10)
DATA SAMPLE/'1'R,'2'R,'3'R,'4'R,'5'R,'6'R,'7'R,'8'R,'9'R,-1/
CALL SAMEQU('ASCII'H,'0'R,SAMPLE)
```

SEE ALSO

SAMSIZE(3F)

NAME

SAMFILE – Defines subroutines for Sort/Merge operations

SYNOPSIS

CALL SAMFILE(*ftype*[[,*option*,*subname*] [,*option*,*subname*]])

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SAMFILE is used to specify subroutines to be used by the Sort/Merge routine. SAMFILE is used when the following options are specified:

ftype A Hollerith constant that is left-justified and padded with blanks, or an integer variable containing a Hollerith constant that specifies the file access type. Allowable values are:

- `'IN´H` Specifies an input subroutine; use the option ´READ=´ followed by a subroutine name.
- `'IN/M´H` Specifies an input subroutine that generates records that are already sorted; use the option `'READ='` followed by a subroutine name.
- `'OUT´H` Specifies an output subroutine; use the option ´WRITE=´ followed by a subroutine name.

option A Hollerith constant that is left-justified and filled with blanks, or an integer variable containing a Hollerith constant that specifies an action to be performed by a user-supplied subroutine for the Sort/Merge sessions. Depending on *ftype*, *option* can be one of the following:

ftype=`'IN'H` or `'IN/M'H`	***ftype***=`'OUT'H`
`'AFTER='H`	`'AFTER='H`
`'EOF='H`	`'=FIRST='H`
`'READ='H`	`'=EQUALS='H`
`'OPEN='H`	`'=LAST='H`
`'CLOSE='H`	`'EOF='H`
`'ERROR='H`	`'WRITE='H`
	`'OPEN='H`
	`'CLOSE='H`
	`'ERROR='H`

subname The name of the user-supplied subroutine to be used by the Sort/Merge session for the action specified by *option*.

User-supplied Subroutines

The user-supplied subroutines all have a common structure:

```
SUBROUTINE MYROUTINE (INBUF,INSZ,RETURN_CODE,OUTBUF,OUTSZ)
INTEGER INSZ,OUTSZ,RETURN_CODE
INTEGER INBUF(INSZ),OUTBUF(OUTSZ)
```

For an OPEN routine, parameters INBUF, INSZ, OUTBUF, and OUTSZ are dummies and must not be referenced by the subroutine. The RETURN_CODE is undefined on entry; it must be set before return to either ´RETURN´ or ´ABORT´. ´RETURN´ indicates successful initialization and ´ABORT´ indicates the opposite and terminates the sorting process.

For a READ routine, the user's program may fill INBUF(1) through INBUF(INSZ) with data for the input record. It should set INSZ to the number of words actually filled. RETURN_CODE may be set to ´ABORT´ in case of error; this terminates the sort process. If there is no more information, RETURN_CODE should be set to ´EOF´. If the routine successfully fills INBUF with information, RETURN_CODE should be set to ´INCLUDE´ (it is initialized by the Sort/Merge routine to this).

The CLOSE routine parameters INBUF, INSZ, OUTBUF, and OUTSZ are all dummies and must not be referenced by the user subroutine. The RETURN_CODE is undefined on entry; it must be set before return to either ´RETURN´ or ´ABORT´. ´RETURN´ indicates successful initialization and ´ABORT´ indicates the opposite and terminates the sorting process.

The WRITE routine words INBUF(1) through INBUF(INSZ) contain the current sorted output record. The write routine can do whatever it chooses with the contents of this record. The return code defaults to ´RETURN´, which allows the process to proceed. You may also specify ´ABORT´ in the event of error, which terminates the sort. If you specify ´EOF´, the Sort/Merge routine closes the current output sink and advances to the next sink, if any.

An error routine is called in a different manner:

```
SUBROUTINE MYERROR(NUMBER,MESSAGE)
INTEGER NUMBER
INTEGER MESSAGE(8)
```

You can supply values for the error number and the message; if a different error handler is needed for each file, you can embed knowledge about the source (file or internally generated) in the error handler. When the error handler returns, the sort terminates. The return codes are Hollerith constants that are left-justified and blank-filled.

EXAMPLES

In the following example, MYOPEN is called before the first read and MYCLOSE is called after the read routine declares end-of-file. MYERROR is called in the event of an error.

```
 EXTERNAL MYREAD, MYOPEN, MYCLOSE, MYERR
 CALL SAMFILE('IN'H,'READ='H,MYREAD,'OPEN='H,MYOPEN,
$             'CLOSE='H,MYCLOSE,'ERROR='H,MYERR)
```

In the following example, whenever Sort/Merge would write an output record it calls MYWRITE instead.

```
 EXTERNAL MYWRITE,MY_W_OPEN,MY_W_CLOSE,MY_W_ERR
 CALL SAMFILE('OUT'H,'WRITE='H,MY_WRITE,'OPEN='H,MY_W_OPEN,
$             'CLOSE='H,MY_W_CLOSE,'ERROR='H,MY_W_ERR)
```

SEE ALSO

SAMPATH(3F), SAMSORT(3F)

NAME

SAMGO – Initiate a Sort/Merge session

SYNOPSIS

```
CALL SAMGO
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The SAMGO routine starts a Sort/Merge session. This call must be last chronologically in a series of calls that start with either SAMSORT or SAMMERGE. There must be one or more intervening calls to SAMKEY, one or more calls to SAMFILE or SAMPATH specifying input sources, and one or more calls to SAMFILE or SAMPATH specifying output sinks. Optional calls can be made to SAMEQU, SAMOPT, SAMSEQ, or SAMSIZE.

SEE ALSO

SAMEQU(3F), SAMFILE(3F), SAMKEY(3F), SAMOPT(3F), SAMPATH(3F), SAMSEQ(3F), SAMSIZE(3F), SAMSORT(3F)

NAME

SAMKEY – Defines sort keys for a Sort/Merge session

SYNOPSIS

CALL SAMKEY(*type*[,*order*],arg_1, ... arg_n [,*colseq*])

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SAMKEY is one of a sequence of calls that specify the sort or merge operation to be performed. Unless you provide your own comparison routine (using the ´COMPARE=´ argument to SAMSORT or SAMMERGE) there must be a least one call to SAMKEY between a call to SAMSORT or SAMMERGE and the call to SAMGO, which initiates the Sort/Merge operation.

A call to SAMKEY must be preceded by a call to either SAMSORT or SAMMERGE.

There is no limit on the number of calls to SAMKEY.

The order in which you make the calls to SAMKEY determines the significance of the key in the sort. The first key specified has the most weight in comparisons.

The following is a list of valid arguments to this routine:

type A Hollerith constant or an integer variable containing a Hollerith constant specifying a Sort/Merge operation. *type* is a Hollerith constant that is left-justified, blank-filled and can contain the following values:

- ´BIN´H Sorts a field of arbitrary length as an unsigned integer. The field can be as small as 1 bit; there is no limit on length.
- ´INT´H Sorts a signed (2's complement) integer field. This is usually applied to a 64-bit Cray word. The maximum length is 64 bits; the field can be shorter. Such a field will usually start on a word boundary.
- ´FLT´H Sorts a floating-point number in Cray internal format. The maximum field length is 64 bits. The field can be shorter, but results will not be meaningful with a length less than 18 bits. The numbers should be normalized.
- ´CHR´H Sorts fields containing characters. The length of the string should be a multiple of the character size, which defaults to 8 bits (but which can be set with the SAMSIZE subroutine call). The default collating sequence is ASCII, but the collating options can be used or you can provide your own. There is no practical limit on the size of the field.

´DEC´H Compares a character field that contains decimal numbers. Results are meaningful only if the collating sequence is equivalent to ASCII or ASCIIUP sequences (an error results in other cases). The field must contain only decimal digits and at most one decimal point and at most one leading sign (positive or negative). Exponential notation (for example, 1.0E+2) cannot be used. Leading and trailing blanks are ignored. Embedded blanks are treated as zeros.

The *type* specified determines the file format used by the sorting routines. A *type* of 'BIN'H, 'INT'H, or 'FLT'H indicates UNFORMATTED input/output files. A *type* of 'CHR'H or 'DEC'H indicates FORMATTED input/output files.

order A Hollerith constant or an integer variable containing a Hollerith constant specifying the order in which records should be sorted. Allowable values are ´DESCEND´H or ´ASCEND´H (the default will be sorted in ascending field value). The value must be left-justified and filled with blanks.

arg Argument(s) specifying the starting location of the field. If there is no default length or if the default length is inappropriate, *arg* also specifies the length or ending position of the field. All the arguments associated with the starting location must precede any argument associated with the ending location or length. Arguments are provided in pairs (one of the key words followed by a decimal value). The key words are:

```
'START'H
'LENGTH'H
'END'H
'WORD='H
'CHR='H
'BIT='H
```

colseq An integer variable or Hollerith constant specifying the collating sequence to be used if the key type is ´CHR´. This may be a user-specified collating sequence, or it can be one of the following built-in collating sequences:

´ASCII´H	Sorts in ASCII sequence
´ASCIIUP´H	Sorts in ASCII sequence except lowercase letters are treated as uppercase letters
´EBCDIC´H	Sorts in ASCII sequence according to the EBCDIC sequence
´EBCDICUP´H	Sorts in ASCII sequence according to the EBCDIC sequence except that lowercase letters are treated as uppercase letters

Values for *arg*

Hollerith constants are used to specify information about location. The starting location of a field is tracked as a bit offset within the record. The first (high-order) bit is bit number 1. You can also specify a word offset or a character offset. The word offset is transformed into a bit offset at the rate of 64 bits per word (or the value specified in a SAMSIZE subroutine call). A character offset is transformed into a bit offset at a rate of 8 bits per character (or the size specified in a SAMSIZE call).

You can specify one, two, or all three of the parameters for specifying a bit offset. If you are doing a ´BIN´ comparison, you might want to start a field at the 5th bit of the 6th character of the 7th word. Any of the following sets of arguments would work:

```
,'WORD='H,7,'CHR='H, 6,'BIT='H,5,
,'WORD='H,6,'CHR='H,14,'BIT='H,5,
,'WORD='H,5,'CHR='H,22,'BIT='H,5,
,'WORD='H,4,'CHR='H,30,'BIT='H,5,
,'WORD='H,3,'CHR='H,38,'BIT='H,5,
,'WORD='H,2,'CHR='H,46,'BIT='H,5,
,'WORD='H,1,'CHR='H,54,'BIT='H,5,
            'CHR='H,53,'BIT='H,13,
                       'BIT='H,429,
```

If unspecified, the word, character, or bit parameter value is assumed to be 1. The word, character, and bit parameters values are transformed into a bit offset in the obvious way.

You must first specify the starting position. The *arg* parameter should be the value ´START´. It should be followed by 2, 4, or 6 parameters giving the actual starting position. Following this, you must specify either an ending bit position or a length. In either case, you combine the keywords in exactly the same way as for the starting bit position. Use the 'END'H or ´LENGTH'H keywords.

EXAMPLES

In the following example, the key is a character string that starts with the fourth character of the record and consists of 10 characters. The example does not specify a ranking order or collating sequence; therefore, the defaults apply.

```
CALL SAMKEY('CHR'H,'START'H,'CHR='H,4,'LENGTH'H,'CHR='H,10)
```

In the following example, the key is an integer that starts with the seventh bit position and is 12 bits. The keys are ranked in ascending order.

```
CALL SAMKEY('INT'H,'START'H,'BIT='H,7,'LENGTH'H,'BIT='H,12)
```

SEE ALSO

SAMGO(3F), SAMKEY(3F), SAMPATH(3F), SAMSORT(3F)

NAME

SAMOPT – Specifies sort options used in a Sort/Merge session

SYNOPSIS

CALL SAMOPT(*arg*[,*arg*])

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SAMOPT is used to specify the sort options available during a Sort/Merge session.

arg A Hollerith constant, or an integer variable containing a Hollerith constant, specifying sort options. *arg* is a Hollerith constant that is left-justified and padded with blanks. Specify one or both of the following values for *arg*:

- ´NOVERIFY´H Disables Sort/Merge verification of file order. The Sort/Merge routine will verify that files exist in the order in which they have been declared. Specify the order by starting the sort with the SAMMERGE subroutine which affects all input files, or by calling SAMFILE or SAMPATH with an *ftype* of ´IN/M´H for a specific file.
- ´RETAIN´H Maintains the original input order of a sequence of records with equivalent keys. If this option is not specified, the Sort/Merge routine does not maintain the original order.

SEE ALSO

SAMFILE(3F), SAMPATH(3F), SAMSORT(3F)

NAME

SAMPATH – Defines input and output files and characteristics for a Sort/Merge session

SYNOPSIS

CALL SAMPATH(*ftype*,*pathname*,[,*option*, *subname*]...)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SAMPATH specifies Sort/Merge input and output files to be used during a Sort/Merge session. User-supplied routines can be added at various stages of processing during the session.

The SAMFILE routine should be used when an output file is not required during a Sort/Merge session.

The following is a list of valid arguments to this routine:

ftype A Hollerith constant, or an integer variable containing a Hollerith constant that specifies a file access type. *ftype* is a Hollerith constant that must be left-justified and padded with blanks. Allowable values are:

- ´IN´H Specifies an input file
- ´IN/M´H Specifies an input file that generates records that are sorted
- ´OUT´H Specifies an output file

pathname An integer or Hollerith constant specifying the absolute or relative path name of the input or ouptut file to be used during a Sort/Merge session.

option An integer or Hollerith constant specifying when to use a subroutine supplied through *subname*. *option* is a Hollerith constant that must be left-justified and padded with blanks. The allowable values are:

- ´AFTER´H Specifies that a subroutine will be called after a record has been selected for either input or output
- ´=FIRST=´H, ´=EQUALS=´H, ´=LAST=´H Specifies that a subroutine be called to process the first member of a class of records with equal keys or to process the last member of the class, or to process the middle records

subname The name of a subroutine to be called as specified in *option*.

User-supplied Routines

The user may supply subroutines to be used by the Sort/Merge routine during a Sort/Merge operation. See the description of user-supplied subroutines in SAMFILE. The third parameter in a call to one of these subroutines may be a STATUS or an ACTION variable. The following Hollerith constants can be specified as STATUS and ACTION variables:

STATUS Used to process records in routines. STATUS can have the following values:

- ´INCLUDE´ Includes the new record during input processing. You must copy the contents of INREC(1:INWDS) to OUTREC and set OUTWDS to INWDS. You can modify the record as you copy it, and revise the length of the record.
- ´REPLACE´ Creates a new record during input processing. The new record replaces the input record. A replacement counter is incremented (this counter appears in the statistical summary if requested).
- ´INSERT´ Inserts a new record during input processing or end-of-file (EOF) processing. Store the new record in OUTREC and set OUTWDS appropriately. An insertion counter is incremented (this counter appears in the statistical summary if requested).
- ´OMIT´ Omits the record during input processing. The Sort/Merge package ignores the current record and retrieves the next record. An omission counter is incremented (this counter appears in the statisical summary if requested).
- ´EOF´ States that the current input file is finished. The Sort/Merge package proceeds as if an EOF had been detected. If one has been specified, the next input file is used.
- ´ABORT´ States that an error occurred during input processing or EOF processing. If you provide an error-processing routine, it is called and the Sort/Merge session terminates.
- ´RETURN´ Used only in EOF processing. The EOF condition is propagated back to the Sort/Merge package.

ACTION Used in output processing. The following values are available for the ACTION variable:

- ´INCLUDE´ Includes the record presented as INREC as the output record.
- ´OMIT´ Discards the current record. An omission counter is incremented.
- ´REPLACE´ Writes the record stored to OUTREC (the length of which is stored in OUTWDS). A replacement record counter is incremented. The record offered in INREC is discarded.
- ´INSERT´ Writes the record stored to OUTREC (the length of which is stored in OUTWDS). A replacement record counter is incremented. Sort/Merge immediately calls the routine presenting the same record.

´EOF´	Calls a user-supplied EOF routine (if available). If only one output file was specified, the sort terminates normally. If more than one output file was specified, the current file is closed.
´ABORT´	Terminates the sort/merge session.
´RETURN´	Used only in EOF processing. The EOF condition is propagated back to the sort/merge package.

Equivalence Class Processing

Equivalence class processing is similar to other output processing. The possible return codes are the same and operate in the same way. Any records that are inserted do not affect the determination of equivalence classes. If a user-supplied routine inserts a record, it will be recalled with the original record, even though it is not first. The Sort/Merge session will call the user-supplied subroutine with the original record until that record has been processed.

It is not necessary to write routines for all three equivalence class possibilities. Any user-supplied routine to handle the EOF condition will be called if the sort runs out of records or if another user-supplied routine has returned the 'EOF' for STATUS.

The only valid ACTION values are ´INSERT´ or ´RETURN´. Store the record to be inserted in OUTREC and set its length in OUTWDS. Sort/Merge writes the record and calls the EOF processing routine again. The inserted record is not considered in the equivalence processing. If a record is supplied in OUTREC, its length must be less than the maximum record length, which is 20 words (unless changed with the SAMTUNE call).

EXAMPLES

The following example gains control after each record is read:

```
      EXTERNAL MYAFTER
C     other parts of the program
      CALL SAMPATH('IN'H, 'the_data_file'H, 'AFTER='H, MYAFTER)
```

After the records are read from the_data_file, the Sort/Merge package calls the MYAFTER subroutine, which should have the following elements:

```
      SUBROUTINE MYAFTER(INREC,INWDS,STATUS,OUTREC,OUTWDS)
      INTEGER INWDS, STATUS, OUTWDS
      INTEGER INREC(INWDS), OUTRED(OUTWDS)
```

The size of the input and output records are in Cray 64-bit words. When reading character records, the record size is rounded to an integral number of Cray words. The output record is also an integral number of Cray words.

The following example gains control after an EOF condition has been detected on an input file:

```
      EXTERNAL MYEOF
C     other parts of the program
      CALL SAMPATH('IN'H,'the_data_file'H,'EOF='H, MYEOF)
```

The following example includes both an ´AFTER´ routine and an ´EOF´ routine:

```
      EXTERNAL MYAFTER, MYEOF
    C  other parts of the program
      CALL SAMPATH('IN'H,'the_data_file'H,'AFTER='H,MYAFTER,'EOF='H,MYEOF
```

When the EOF condition occurs in the file, the subroutine is called. It should have the following elements:

```
      SUBROUTINE MYEOF (INREC,INWDS,STATUS,OUTREC,OUTWDS)
      INTEGER INWDS, STATUS, OUTWDS
      INTEGER INREC(INWDS), OUTREC(OUTWDS)
```

The following example captures records before they are written during output:

```
      EXTERNAL OUTAFTER
     C  other program parts
      CALL SAMPATH('OUT'H,path_name,'AFTER='H,OUTAFTER)
```

Before the output record is written, the Sort/Merge package calls the subroutine, which should have the following elements:

```
      SUBROUTINE OUTAFTER(INREC,INWDS,ACTION,OUTREC,OUTWDS)
      INTEGER INWDS,ACTION,OUTWDS
      INTEGER INREC(INWDS), OUTREC(OUTWDS)
```

SEE ALSO

SAMSORT(3F), SAMFILE(3F)

NAME

SAMSEQ – Specifies and defines a collating sequence

SYNOPSIS

```
INTEGER init_array (2**N)
CALL SAMSEQ(colseq, init_array)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SAMSEQ is used to define a collating sequence used in Sort/Merge sessions.

The following is a valid list of arguments to this routine:

colseq A Hollerith constant or an integer variable containing a Hollerith constant specifying a collating sequence. *colseq* is a Hollerith constant with a maximum of 8 characters and is left-justified and filled with blanks. This can be specified in subsequent SAMKEY calls.

init_array An integer array containing the collating sequence. It can have a maximum size of 2**N, where N is the character size (by default 8 bits; this can be set using the SAMSIZE routine). Each element in the array holds 1 Hollerith character that is right-justified and padded with zeros on the left. The character used in the low-order bits of the first element of the array is used as the low value in the collating sequence. The value used for the second element becomes the next lowest value in the collating sequence, and so on. The value –1 terminates the array entries.

EXAMPLES

In the following example, a user-specified collating sequence contains the default character set in reverse order. A file is sorted in reverse order by an ASCII key; the ´DESCEND´H keyword is not used in the SAMKEY call. The following program fragment defines a collating sequence in which X´FF´ compares low with respect to other elements. X´42´ (the ´B´ character) compares low with respect to X´41´ (the ´A´ character). You can then use the ´IICSA´H collating sequence in a SAMKEY call, achieving a descending sort.

```
      INTEGER REVERSED(256)
      DO 100 I = 1,256
      REVERSED(I) = 256 - I
  100 CONTINUE
      CALL SAMSEQ('IICSA'H,REVERSED)
```

In the following example, the SAMPLE collating sequence will consider 999 to be smaller than 000. In addition, any unspecified characters compare larger than any specified character and are equal to each other. Therefore 99Z is larger than 999, but is equal to 99A. The nonstandard notation ´*´R indicates that the character value is to be right-justified in the word and that unused character positions are to be set to 0.

```
INTEGER SAMPLE(11)
DATA SAMPLE/'9'R, '8'R, '7'R, '6'R, '5'R, '4'R, '3'R, '2'R, '1'R, -1/
CALL SAMSEQ('SAMPLE'H,SAMPLE)
```

SEE ALSO

SAMKEY(3F), SAMSIZE(3F)

NAME

SAMSIZE – Specifies word and character sizes for Sort/Merge session

SYNOPSIS

```
CALL SAMSIZE(keyword, value)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SAMSIZE specifies a character size or word size for character comparisons.

keyword A Hollerith constant, or an integer variable containing a Hollerith constant, specifying either character or word size. *keyword* is a Hollerith constant, containing either ´CHR=´H or ´WORD=´H.

value An integer variable, expression, or constant specifying the number of keyword items.

To specify a character size for use in character comparisons, specify ´CHR=´H for *keyword*, followed by an integer between 1 and 12 for *value*.

To specify a word size for character comparison, specify ´WORD=´H for *keyword* and an integer for *value*. The default *value* is 64; there is no maximum. This value is used to interpret the ´WORD=´H keyword in the SAMKEY subroutines.

Both options can be specified in the same call.

SEE ALSO

SAMKEY(3F)

NAME

SAMSORT, SAMMERGE – Begins a Sort/Merge specification

SYNOPSIS

```
CALL SAMSORT(errflag [,´STAT´H] [,´STATF=´H,dsname] [,´ERRORF=´H,dsname]
[,´ERROR=´H,errsub] [,´COMPARE=´H,compsub])
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SAMSORT starts the specification of a Sort/Merge session. After this call, you must call SAMKEY one or more times, call either SAMFILE or SAMPATH one or more times, and call other subroutines as needed.

SAMSORT must not be called a second time until the sort you are specifying has been executed by a call to SAMGO. If no error is encountered by SAMSORT, *errflag* is set to 0. If an error is detected by SAMSORT, *errflag* is set to 1. If an error is detected by a user-supplied routine, *errflag* is set to 2.

The following is a list of valid arguments for this routine:

errflag	An integer variable that is a member of a common block.
´STAT´H	Used to generate statistics on the Sort/Merge operation. Use either this option or ´STATF´H.
´STATF´H	Used to generate statistics; use either this option or ´STAT´H.
dsname	An integer variable, expression, or constant containing the path name of the file (a total of 8 characters or less) to which statistics generated by ´STATF´H should be written or to which error messages should be saved. If more than 8 characters are needed for a path name, use the SAMPATH subroutine. The Hollerith constant is left-justified and blank-filled.
´ERRORF´H	Used to specify a file to receive error messages. If more than 8 characters are needed for a path name, use the SAMPATH subroutine instead.
´ERROR´H	Used to specify an error routine. This routine cannot assist in error recovery; it can, however, do final cleanup such as closing files, writing error messages, and so on.
errsub	The name of the error subroutine.
´COMPARE´H	Used to specify an optional user-supplied comparison routine. See the CAUTIONS section for information about including a user-supplied comparison routine.
compsub	The name of the comparison subroutine.

CAUTIONS

It is recommended that you do not supply your own comparison routine to be used during the sorting or merging process. This prevents the Sort/Merge routine from storing key information compactly and from using the ORDERS(3F) routine from the scientific libraries.

SEE ALSO

SAMEQU(3F), SAMFILE(3F), SAMGO(3F), SAMKEY(3F), SAMPATH(3F), SAMOPT(3F), SAMSEQ(3F), SAMSIZE(3F)

NAME

SAMTUNE – Modifies selected parameters used in a Sort/Merge session

SYNOPSIS

CALL SAMTUNE(*keyword, value, ...*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SAMTUNE is used to alter the parameters used in a Sort/Merge session.

Performance in a Sort/Merge session is determined by the amount of available central memory. Sort/Merge uses half of a megaword of memory by default; the majority of the memory is used for buffers for intermediate files used during the merge phase. During the first phase all of the memory is used to form runs of greatest length.

keyword A Hollerith constant or integer variable containing a Hollerith constant specifying tuning keywords. *keyword* is a Hollerith constant that is left-justified and blank-filled.

value An integer constant or variable, depending on the *keyword*.

Initial runs are formed using the ORDERS(3F) routine in libsci unless a user-supplied comparison routine is specified. The ORDERS routine is vectorized; the later merge stage sorts the records using a tournament sort method. The merge phase sorts are scalar.

The following are valid *keyword*s. Any keywords must be used as is, with equal signs and single quotation marks.

´AVRL=´H Specifies the average record length. The default is the maximum record length. This value is used to allocate internal tables. It is recommended that you use an average record length that is too small rather than one that is too large.

´MXRL=´H Specifies the maximum record length. The default is 20 Cray words (160 bytes). If the maximum record length is too short, the sort aborts during the input phase.

´NRECEST=´H An estimate of the number of records in the input files. The default is 1,000,000 records. This value is no longer used.

´DISK=´H No longer used in this implementation. Instead, you may specify a colon-separated list of directories in the CSORTDIR environment variable.

´DSLO=´H No longer used in this implementation.

´NAMEBM=´H No longer used in this implementation.

´NAMESSD=´H No longer used in this implementation.

´NDS=´H	Specifies the number of temporary datasets to be used during the merge phase. The default is 10; minimum is 4. Change this parameter only if you must run a Sort/Merge routine in minimum memory. A large value is recommended, but many temporary datasets are assigned smaller buffers and buffers should be large to maximize I/O efficiency. This makes it difficult to assign an efficient number for this keyword.
´NDSSD=´H	No longer used in this implementation.
´NBSSD=´H	No longer used in this implementation.
´NDBM=´H	No longer used in this implementation.
´NBBM=´H	No longer used in this implementation.
´NBDSK=´H	Specifies the number of sort buffers to be allocated to each temporary dataset. The size of the buffer is specified by the ´MNBL=´H and ´MXBL=´H parameters. The default value is 2.
´MNBL=´H	Specifies the number of word blocks in each sort buffer. The default is 42 (the track size of a DD-49). If you supply both a minimum and a maximum, the minimum must be the smaller of the two numbers.
´MXBL=´H	Specifies the number of word blocks in each sort buffer. The default is 42. If you supply both a minimum and a maximum, the minimum must be the smaller of the two numbers.

SEE ALSO

SAMSORT(3F)

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