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