April 1993
Order Number: 312487-001

PARAGON OSF/1
C SYSTEM CALLS
REFERENCE MANUAL

Intel® Corporation

REV.	REVISION HISTORY	DATE
-001	Original Issue	4/93

LIMITED RIGHTS

Preface

The Paragon™ OSF/1 C system calls are described in two manuals:

- The *OSF/1 Programmer's Reference* describes the standard OSF/1 system calls, library routines, file formats, and special files.
- The *Paragon™ OSF/1 C System Calls Reference Manual* (this manual) describes the system calls and library routines (referred to collectively as "system calls") that let you access the special capabilities of the Intel supercomputer. These calls let you:
  - Create and control parallel applications and partitions.
  - Exchange messages between processes.
  - Get information about the computing environment (such as, the number of nodes in the current application).
  - Perform global operations optimized for the Intel supercomputer's architecture.
  - Perform 64-bit integer arithmetic (used for manipulating file pointers that exceed 32 bits).
  - Read and write files.

This manual assumes that you are an application programmer proficient in using the C programming language and the OSF/1 operating system.

Organization

The manual contains a "manual page" for each Paragon™ OSF/1 system call, organized alphabetically. Each manual page provides the following information:

- Synopsis (including call syntax, parameter declarations, and include files)
- Description of any parameters.
- Description of the call (including programming hints)
- Return values (if applicable)
- Error messages (including causes and remedies)
- Related calls

Some of the manual pages in this manual discuss several related system calls. For example, the **csend()** manual page discusses both the **csend()** and **csendx()** system calls. The title of a manual page that discusses more than one call is the name of the first call discussed on the page. To find the discussion of any call, use the Index at the back of this manual.

Appendix A tells how to select message types and build message type selectors for the message-passing system calls.

Notational Conventions

This section describes the following notational conventions:

- Type style conventions
- System call syntax descriptions

Type Style Conventions

This manual uses the following type style conventions:

Bold Identifies command names and switches, system call names, reserved words, and other items that must be used exactly as shown.

Italic Identifies variables, filenames, directories, processes, user names, and writer annotations in examples. Italic type style is also occasionally used to emphasize a word or phrase.

`Plain-Monospace`
Identifies computer output (prompts and messages), examples, and values of variables. Some examples contain annotations that describe specific parts of the example. These annotations (which are not part of the example code or session) appear in *italic* type style and flush with the right margin.

`Bold-Italic-Monospace`
Identifies user input (what you enter in response to some prompt).

`Bold-Monospace`
Identifies the names of keyboard keys (which are also enclosed in angle brackets). A dash indicates that the key preceding the dash is to be held down *while* the key following the dash is pressed. For example:

`<Break>` **`<s>`** **`<Ctrl-Alt-Del>`**

[ ] (Brackets) Surround optional items.

. . . (Ellipsis dots) Indicate that the preceding item may be repeated.

| (Bar) Separates two or more items of which you may select only one.

{ } (Braces) Surround two or more items of which you must select one.

System Call Syntax Descriptions

In this manual, a prototype for each system call is described in the "Synopsis" section, which contains the following:

- Include file declarations needed by the system call.
- Syntax of the system call.
- Parameter declarations of each system call.

The following notational conventions apply to the "Synopsis" section:

Bold	Identifies system call names.
Italic	Identifies parameter names.
[ ]	(Brackets) Surround optional items.
|	(Bar) Separates two or more items of which you may select only one.
{ }	(Braces) Surround two or more items of which you must select one.
. . .	(Ellipsis dots) Indicate that the preceding item may be repeated.

For example:

- The synopsis for the **iprobe()** system call appears as follows:

```
#include <nx.h>

long iprobe(
    long typesel);
```

Applicable Documents

For more information, refer to the following documents:

- *OSF/1 Programmer's Reference*
- *Paragon™ OSF/1 User's Guide*
- *Paragon™ OSF/1 Fortran System Calls Reference Manual*
- *Paragon™ OSF/1 Commands Reference Manual*

How Errors are Handled

How the Paragon OSF/1 operating system handles errors depends on the system call involved:

- For Paragon OSF/1 system calls whose names begin with "nx_" or "nxr_" or "x", the calls either return -1 and set the variable *errno* to a value that describes the error, or it sends a signal to the calling process. You can use **nx_perror(3)** or **perror(3)** to print a message for the value of *errno*.
- For all other Paragon OSF/1 system calls (except those whose names begin with "nx", "nxr", or "x"), the system normally displays a message on the terminal and terminates the calling process.
- For all Paragon OSF/1 system calls (except those whose names begin with "nx", "nxr", or "x"), there is a corresponding underscore system call that returns -1 and sets the variable *errno* to a value that describes the error. The underscore system calls are identified by an underscore (_) as the first character of the name. For example, the **_crecv()** function is the underscore version of the **_cread()** function. The underscore calls allow you to write programs that take specific actions when an error occurs. These calls do not terminate a process when an error occurs. You can use **nx_perror(3)** or **perror(3)** to print a message for the value of *errno*. For a complete list of the *errno* values set by the underscore calls, see the **errno** manual page.

Comments and Assistance

Intel Supercomputer Systems Division is eager to hear of your experiences with our products. Please call us if you need assistance, have questions, or otherwise want to comment on your Paragon system.

U.S.A./Canada Intel Corporation
phone: 800-421-2823
Internet: support@ssd.intel.com

Intel Corporation Italia s.p.a.
Milanofiori Palazzo
20090 Assago
Milano
Italy
1678 77203 (toll free)

France Intel Corporation
1 Rue Edison-BP303
78054 St. Quentin-en-Yvelines Cedex
France
0590 8602 (toll free)

Japan Intel Corporation K.K.
Supercomputer Systems Division
5-6 Tokodai, Tsukuba City
Ibaraki-Ken 300-26
Japan
0298-47-8904

United Kingdom Intel Corporation (UK) Ltd.
Supercomputer System Division
Pipers Way
Swindon SN3 IRJ
England
0800 212665 (toll free)
(44) 793 491056 (*answered in French*)
(44) 793 431062 (*answered in Italian*)
(44) 793 480874 (*answered in German*)
(44) 793 495108 (*answered in English*)

Germany Intel Semiconductor GmbH
Dornacher Strasse 1
8016 Feldkirchen bel Muenchen
Germany
0130 813741 (toll free)

World Headquarters
Intel Corporation
Supercomputer Systems Division
15201 N.W. Greenbrier Parkway
Beaverton, Oregon 97006
U.S.A.
(503) 629-7600

If you have comments about the Paragon manuals, please fill out and mail the enclosed Comment Card. You can also send your comments electronically to the following address:

techpubs@ssd.intel.com (Internet)

Table of Contents

Appendix A
Message Types and Typesel Masks

List of Tables

CPROBE() CPROBE()

cprobe(), **cprobex()**: Waits (blocks) until a message is ready to be received. (Synchronous probe)

Synopsis

```
#include <nx.h>

void cprobe(
        long typesel );

void cprobex(
        long typesel,
        long nodesel,
        long ptypesel,
        long info[] );
```

Parameters

typesel	Message type(s) to receive. Setting this parameter to `-1` probes for a message of any type. Refer to Appendix A of the *Paragon™ OSF/1 C System Calls Reference Manual* for more information about message type selectors.
nodesel	Node number of the sender. Setting the *nodesel* parameter to `-1` probes for a message from any node.
ptypesel	Process type of the sender. Setting the *ptypesel* parameter to `-1` probes for a message from any process type.
info	Eight-element array of long integers in which to store message information. The first four elements contain the message's type, length, sending node, and sending process type. The last four elements are reserved for system use. If you do not need this information, you can specify the global array *msginfo*, which is the array used by the **info...()** calls. See the *nx.h* include file for information about the global array *msginfo*.

CPROBE() *(cont.)* CPROBE() *(cont.)*

Description

Use the appropriate synchronous probe system call to block the calling process until a specified message is ready to be received:

- Use the **cprobe()** function to wait for a message of a specified type. Use the **info...()** system calls to get more information about the message.
- Use the **cprobex()** function to wait for a message of a specified type from a specified sender and store information about the message in the *info* array.

When a synchronous probe system call successfully returns, the message of the specified type is available. Use the receive system calls (for example, **crecv()** or **irecv()**) to receive the message.

These are synchronous system calls. The calling process waits (blocks) until the specified message is ready to be received. To probe for a message of the specified type without blocking the calling process, use one of the asynchronous probe system calls (for example, **iprobe()**).

Return Values

Upon successful completion, the **cprobe()** and **cprobex()** functions return control to the calling process; no values are returned. Otherwise, these functions display an error message to standard error and cause the calling process to terminate.

Upon successful completion, the **_cprobe()** and **_cprobex()** functions return `0` (zero). Otherwise, these functions return `-1` and set *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

crecv(), **errno**, **infocount()**, **infonode()**, **infoptype()**, **infotype()**, **iprobe()**, **irecv()**

CREAD() CREAD()

Reads from a file and blocks the calling process until the read completes. (Synchronous read)

Synopsis

```
#include <nx.h>

void cread(
        int fildes,
        char *buffer,
        unsigned int nbytes );
```

Parameters

fildes	File descriptor identifying the file to be read.
buffer	Pointer to the buffer in which to store the data after it is read from the file.
nbytes	Number of bytes to read from the file associated with the *fildes* parameter.

Description

Except for error handling, the **cread()** function operates identically to the OSF/1 **read()** function. See the **read(2)** manual page in the *OSF/1 Programmer's Reference*.

This is a synchronous system call. The calling process waits (blocks) until the read completes. To read a file without blocking the calling process, use the **iread()** function.

Reading past the end of a file causes an error, so you must know how many bytes remain in a file before you read from it. If any error occurs, the **cread()** function prints an error message and terminates the calling process. Use the **iseof()** function to detect end-of-file after calling the **cread()** function. Use the **lseek()** function to determine the length of a file.

If you need to detect errors in reading and writing, use either the standard OSF/1 calls (the **read()** and **write()** functions, described in the *OSF/1 Programmer's Reference*) or the underscore versions of the parallel I/O calls (the **_cread()** and **_cwrite()** function).

CREAD() *(cont.)* CREAD() *(cont.)*

Return Values

Upon successful completion, the **cread()** function returns control to the calling process; no values are returned. Otherwise, the **cread()** function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_cread()** function returns the number of bytes read. Otherwise, the **_cread()** function returns -1 and sets *errno* to indicate the error.

Errors

If the **_cread()** function fails, *errno* may be set to one of the error code values described for the OSF/1 **read()** function or the following value:

EMIXIO In I/O mode **M_SYNC**, nodes are attempting different operations (reads and writes) to a shared file. In these modes, all nodes must perform the same operation.

See Also

cwrite(), **iread()**, **iseof()**, **iwrite()**, **setiomode()**

OSF/1 Programmer's Reference: **lseek(2)**, **open(2)**, **read(2)**

CRECV() CRECV()

crecv(), crecvx(): Posts a receive for a message and blocks the calling process until the receive completes. (Synchronous receive)

Synopsis

```
#include <nx.h>

void crecv(
        long typesel,
        char *buf,
        long count );

void crecvx(
        long typesel,
        char *buf,
        long count,
        long nodesel,
        long ptypesel,
        long info[] );
```

Parameters

typesel	Message type(s) to receive. Setting this parameter to `-1` receives a message of any type. Refer to Appendix A of the *Paragon™ OSF/1 C System Calls Reference Manual* for more information about message type selectors.
buf	Pointer to the buffer in which to store the received message. The buffer can be of any valid data type, but should match the data type of the buffer in the corresponding send operation.
count	Length (in bytes) of the *buf* parameter.
nodesel	Node number of the sender. Setting the *nodesel* parameter to `-1` receives a message from any node.
ptypesel	Process type of the sender. Setting the *ptypesel* parameter to `-1` receives a message from any process type.

CRECV() *(cont.)* CRECV() *(cont.)*

info Eight-element array of long integers in which to store message information. The first four elements contain the message's type, length, sending node, and sending process type. The last four elements are reserved for system use. If you do not need this information, you can specify the global array *msginfo*, which is the array used by the **info...()** system calls.

Description

Use the appropriate synchronous receive system call to post a receive for a message and wait until the receive completes:

- Use the **crecv()** function to receive a message of a specified type.
- Use the **crecvx()** function to receive a message of a specified type from a specified sender and place information about the message in an array.

When the receive completes, the message is stored in the specified buffer and the calling process resumes execution.

After the **crecv()** function completes, you can use the **info...()** system calls to get more information about the message after it is received. After the **crecvx()** function completes, the same message information is returned in the *info* array.

These are synchronous system calls. The calling process waits (blocks) until the receive completes. To post a receive for a message without blocking the calling process, use an asynchronous receive system call (for example, the **irecv()** function) or a handler receive system call (for example, the **hrecv()** function).

Return Values

Upon successful completion, the **crecv()** and **crecvx()** functions return control to the calling process; no values are returned. If an error occurs, these functions print an error message to standard error and cause the calling process to terminate.

The **_crecv()** and **_crecv()** functions return -1 when an error occurs and set *errno* to indicate the error. Otherwise, these functions return the number of bytes received.

Errors

Refer to the **errno** manual page for a complete list of error codes that occur in the C underscore system calls.

CRECV() *(cont.)*

CRECV() *(cont.)*

See Also

cprobe(), csend(), csendrecv(), errno, hrecv(), hsend(), hsendrecv(), infocount(), infonode(), infoptype(), infotype(), iprobe(), irecv(), isend(), isendrecv()

CSEND() CSEND()

Sends a message and blocks the calling process until the send completes. (Synchronous send)

Synopsis

```
#include <nx.h>

void csend(
        long type,
        char *buf,
        long count,
        long node,
        long ptype );
```

Parameters

type	Type of the message to send. Refer to Appendix A of the *Paragon™ OSF/1 C System Calls Reference Manual* for more information about message types.
buf	Pointer to the buffer containing the message to send. The buffer may be of any valid data type.
count	Number of bytes to send in the *buf* parameter.
node	Node number of the message destination (the receiving node). Setting the *node* parameter to `-1` sends the message to all nodes in the application (except the sending node when the *ptype* parameter is the sender's process type).
ptype	Process type of the message destination (the receiving process).

Description

This is a synchronous system call. The calling process waits (blocks) until the send completes. To send a message without blocking the calling process, use one of the asynchronous send system calls (for example, **isend**()) or one of the handler-send system calls (for example, **hsend**()) instead.

The completion of the send operation does not mean that the message was received, only that the message was sent and the send buffer (*buf*) can be reused.

CSEND() *(cont.)* CSEND() *(cont.)*

Return Values

Upon successful completion, the **csend**() function returns control to the calling process; no values are returned. Otherwise, this function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_csend**() function returns `0` (zero). Otherwise, this function returns `-1` and sets *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

csend(), **crecv**(), **csendrecv**(), **errno**, **hrecv**(), **hsend**(), **hsendrecv**(), **iprobe**(), **irecv**(), **isend**(), **isendrecv**()

CSENDRECV() CSENDRECV()

Sends a message, posts a receive for a reply, and blocks the calling process until the receive completes. (Synchronous send-receive)

Synopsis

```
#include <nx.h>

long csendrecv(
        long type,
        char *sbuf,
        long scount,
        long node,
        long ptype,
        long typesel,
        char *rbuf,
        long rcount );
```

Parameters

type	Type of the message to send. Refer to Appendix A of the *Paragon™ OSF/1 C System Calls Reference Manual* for information on message types.
sbuf	Pointer to the buffer containing the message to send. The buffer may be of any legal data type.
scount	Number of bytes to send in the *sbuf* parameter.
node	Node number of the message destination (the receiving node). Setting the *node* parameter to `-1` sends the message to all nodes in the application (except the sending node when the *ptype* parameter is set to the sender's process type).
ptype	Process type of the message destination (the receiving process).
typesel	Message type(s) to receive. Setting this parameter to `-1` receives a message of any type. Refer to Appendix A of the *Paragon™ OSF/1 C System Calls Reference Manual* for more information about message type selectors.
rbuf	Pointer to the buffer in which to store the reply. The buffer can be of any valid data type, but should match the data type of the buffer in the corresponding send operation.
rcount	Length (in bytes) of the *rbuf* parameter.

CSENDRECV() *(cont.)* CSENDRECV() *(cont.)*

Description

The **csendrecv**() function sends a message and waits for a reply. When a message whose type matches the type(s) specified by the *typesel* parameter arrives, the calling process receives the message, stores it in *rbuf*, and resumes execution.

This is a synchronous system call. The calling process waits (blocks) until the receive completes. To send a message and post a receive for the reply without blocking the calling process, use the **isendrecv**() function or the **hsendrecv**() function (asynchronous system calls) instead of the **csendrecv**() function.

If the reply is too long for the *rbuf* buffer, the receive completes with no error returned, and the content of *rbuf* is undefined.

The **csendrecv**() function does not affect the information returned by the **info...**() system calls.

Return Values

Upon successful completion, the **csendrecv**() function returns the length (in bytes) of the received message, and returns control to the calling process. Otherwise, this function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_csendrecv**() function returns length (in bytes) of the received message. Otherwise, this function returns `-1` and sets *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

cprobe(), **crecv**(), **csend**(), **errno**, **hrecv**(), **hsend**(), **hsendrecv**(), **iprobe**(), **irecv**(), **isend**(), **isendrecv**()

CWRITE() CWRITE()

Writes to a file and blocks the calling process until the write completes. (Synchronous write)

Synopsis

```
#include <nx.h>

void cwrite(
        int fildes,
        char *buffer,
        unsigned int nbytes );
```

Parameters

fildes	File descriptor identifying the open file to which the data is to be written.
buffer	Pointer to the buffer containing the data to be written.
nbytes	Number of bytes to write to the file associated with the *fildes* parameter.

Description

Other than the return values and the additional error discussed below, the **cwrite()** function is identical to the OSF/1 **write()** function. See **write(2)** in the *OSF/1 Programmer's Reference.*

This is a synchronous system call. The calling process waits (blocks) until the write completes. To write a file without blocking the calling process, use the **iwrite()** function.

To determine whether the write operation moved the file pointer to the end of the file, use the **iseof()** function.

Return Values

Upon successful completion, the **cwrite()** function returns control to the calling process; no values are returned. Otherwise, the **cwrite()** function displays an error message to standard output and causes the calling process to terminate.

Upon successful completion, the **_cwrite()** function returns the number of bytes written. Otherwise, the **_cwrite()** function returns -1 and sets *errno* to indicate the error.

CWRITE() *(cont.)* CWRITE() *(cont.)*

Errors

If the **_cwrite()** function fails, *errno* may be set to one of the values described for the OSF/1 **write(2)** function or the following value:

EMIXIO In I/O mode **M_SYNC**, nodes are attempting different operations (reads and writes) to a shared file. In these modes, all nodes must perform the same operation.

See Also

cread(), **iread()**, **iseof()**, **iwrite()**, **setiomode()**

OSF/1 Programmer's Reference: **open(2)**, **write(2)**

DCLOCK() DCLOCK()

Gets elapsed time in seconds since the node was booted.

Synopsis

```
#include <nx.h>

double dclock(void);
```

Description

The **dclock**() function measures time intervals in seconds. The **dclock**() value rolls over approximately every 14 years, maintaining an accuracy of 100 nanoseconds (the count rate for the node's counter) during that time.

NOTE

Each node has its own counter, which differs from the counters on other nodes. Do not use **dclock()** to synchronize processes.

Return Values

Upon successful completion, the **dclock**() function returns a double precision value for the elapsed time (in seconds) since booting the node, and returns control to the calling process. Otherwise, the **dclock**() function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_dclock**() function returns the elapsed time (in seconds) since booting the system. Otherwise, the **_dclock**() function returns -1 and sets *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

errno

EADD() EADD()

eadd(), **ecmp**(), **ediv**(), **emod**(), **emul**(), **esub**(): Perform mathematical operations on extended (64-bit) integers.

Synopsis

```
#include <nx.h>

esize_t eadd(
        esize_t e1,
        esize_t e2 );

long ecmp(
        esize_t e1,
        esize_t e2 );

long ediv(
        esize_t e,
        long n );

long emod(
        esize_t e,
        long n );

esize_t emul(
        esize_t e,
        long n );

esize_t esub(
        esize_t e1,
        esize_t e2 );
```

EADD() *(cont.)* EADD() *(cont.)*

Parameters

e, e1, e2	Extended integer values
n	Integer value used to multiply or divide an extended integer

Description

Extended integers are unsigned 64-bit integers with values from 0 to 2^{64} - 1 (approximately 1.8 x 10^{19}). Always use the extended-integer system calls to access extended integers.

Use these system calls to perform the following mathematical operations on extended integers:

eadd()	Add an extended integer to another extended integer.
ecmp()	Compare two extended integers.
ediv()	Divide an extended integer by an integer.
emod()	Get the remainder of an extended integer divided by an integer.
emul()	Multiply an extended integer with an integer.
esub()	Subtract an extended integer from another extended integer.

These system calls support the extended file sizes in the Parallel File System (PFS).

Return Values

Upon successful completion, the **eadd()**, **emul()**, and **esub()** functions return the computed value of type **esize_t** (see the *nx.h* include file). The type **esize_t** has the following structure:

```
struct s_size {
        long    slow;
        long    shigh;
};
typedef struct s_size esize_t;
```

EADD() *(cont.)* EADD() *(cont.)*

Upon successful completion, the **ecmp**() function returns the following values:

`-1`	if *e1* < *e2*
`0`	if *e1* = *e2*
`1`	if *e1* > *e2*

Upon successful completion, the **ediv**() and **emod**() functions return the computed value (of type **long**). Otherwise, the **eadd**(), **ecmp**(), **ediv**(), **emod**(), **emul**(), and **esub**() functions display an error message to standard error and cause the calling process to terminate.

Upon successful completion, the **_eadd**(), **_ecmp**(), **_ediv**(), **_emod**(), **_emul**(), and **_esub**() functions return the same value as their respective non-underscore version of the function. Otherwise, these functions return `-1` (the functions that return an **esize_t** structure return `-1` in all fields of the structure) and set *errno* to indicate the error.

Errors

If an error occurs during an **_eadd**(), **_ecmp**(), **_ediv**(), **_emod**(), **_emul**(), or **_esub**() function, *errno* may be set to the following error code value:

EQESIZE Arithmetic overflow of extended integer.

If an error occurs during an **_ediv**() or an **_emod**() function, *errno* may be set to the following error code value:

EQESIZE Quotient does not fit into a **long** integer.

See Also

eseek(), **esize**(), **estat**(), **etos**(), **stoe**()

ERRNO ERRNO

Error values returned by functions in the *errno* global variable.

Synopsis

```
#include <errno.h>
```

Description

There are two versions of the Paragon™ OSF/1 C system calls:

- The standard C system calls that send a message to standard error when an error occurs
- The underscore C system calls that return an error code (*errno)* when an error occurs

The standard C system calls terminate a process when an error occurs and send a message to standard error describing the error. For example, the **crecv**() function terminates when an error occurs and it sends a message to the standard error describing the error.

The underscore C system calls are identified by an underscore as the first character of the name. For example, the **_crecv**() function is the underscore version of the **crecv**() function. The underscore calls allow you to write programs that take specific actions when an error occurs. They return a non-negative value upon successful completion. When an error occurs in an underscore system call, the call does not terminate the process, but returns a `-1` value and sets the *errno* global variable with an error value.

The *errno* global variable is set with an error value that has an associated message that helps determine the problem in a program. This manual page provides a complete list of the error values for Paragon OSF/1 C system calls. You can also find the list of error codes in the file */usr/include/sys/errno.h*. See the *OSF/1 Programmer's Reference* for more information about error codes and error numbers.

There are two functions you can use to print out the error code for a program that terminates with an error: **perror**() and **nx_perror**(). The **perror**() function writes an error message on the standard error output that describes the last error encountered by a function, library function, or Paragon OSF/1 system call. The **nx_perror**() function is identical to the **perror**() function, except that it writes the current node number and process type in addition to the error message.

ERRNO *(cont.)*

ERRNO *(cont.)*

For example, the underscore C system call **_crecv**() call does not terminate when an error occurs. On a error, it returns a `-1` and sets *errno* to the error code for the error that occurred. You can use **perror**() or **nx_perror**() to print the error message.

The following table lists the *errno* values for Paragon OSF/1 system calls. The table lists the error code, the error code number, the message text, and notes on the error code. The message text appears in italic text.

Error Code	Value	Messages and Notes
E2BIG	7	*Arg list too long.* The number of bytes received by the argument is too big.
EACCES	13	*Permission denied.* The calling process does not have permission for the operation.
EADDRINUSE	48	*Address already in use.* The specified address is already in use.
EADDRNOTAVAIL	49	*Can't assign requested address.* The specified address is not available from the local machine.
EAEXIST	158	*Application exists for process group.*
EAFNOSUPPORT	47	*Address family not supported by protocol family.* The addresses in a specified address family cannot be used with the socket.
EAGAIN	35	*Resource temporarily unavailable.* A resource, such as a lock or process, is temporarily unavailable.
EAINVALGTH	156	*Give threshold invalid or out of range.* For information about the range of values for the give threshold, see the ***application*** manual page either online or in the *Paragon™ OSF/1 Commands Reference Manual.*
EAINVALMBF	151	*Memory buffer invalid or out of range.* For information about the range of values for the memory buffer size, see the ***application*** manual page either online or in the *Paragon™ OSF/1 Commands Reference Manual.*

ERRNO *(cont.)*

ERRNO *(cont.)*

EAINVALMEA	153	*Memory each invalid or out of range.* For information about the range of values for the memory each size, see the ***application*** manual page either online or in the *Paragon™ OSF/1 Commands Reference Manual.*
EAINVALMEX	152	*Memory Export invalid or out of range.* For information about the range of values for the memory export size, see the ***application*** manual page either online or in the *Paragon™ OSF/1 Commands Reference Manual.*
EAINVALPKT	150	*Packet size invalid or out of range.* For information about the range of values for the packet size, see the ***application*** manual page either online or in the *Paragon™ OSF/1 Commands Reference Manual.*
EAINVALSCT	155	*Send count invalid or out of range.* For information about the range of values for the send count size, see the ***application*** manual page either online or in the *Paragon™ OSF/1 Commands Reference Manual.*
EAINVALSTH	154	*Send threshold invalid or out of range.* For information about the range of values for the send count size, see the ***application*** manual page either online or in the *Paragon™ OSF/1 Commands Reference Manual.*
EALREADY	37	*Operation already in progress.*
EANOEXIST	164	*Application does not exist for process group.* The specified process group does not exist.
EANOTPGL	157	*Calling process not process group leader.*
EBADF	9	*Bad file number.* A socket or file descriptor parameter is invalid.
EBADMSG	84	*Next message has wrong type.*
EBADPORT	101	*Failed port to struct translation.*
EBADRPC	72	*RPC structure is bad.*
EBUSY	16	*Device busy.* The requested element is unavailable, or the associated system limit was exceeded.

ERRNO *(cont.)*

ERRNO *(cont.)*

ECFPS	199	*Seek to different file pointers.* Two or more application processes are calling **lseek()** with different shared I/O modes (**M_SYNCH** or **M_RECORD**).
ECHILD	10	*No child processes.* The child process does not exist, or the requested child process information is unavailable.
ECLONEME	88	*Tells open to clone the device.*
ECONNABORTED	53	*Software caused connection abort.* The software caused a connection to abort because there is no space on the socket's queue and the socket cannot receive further connections.
ECONNREFUSED	61	*Connection refused.*
ECONNRESET	54	*Connection reset by peer.* The attempt to connect was rejected.
EDEADLK	11	*Resource deadlock avoided.* There is a probable deadlock condition, or the requested lock is owned by someone else.
EDESTADDRREQ	39	*Destination address required.*
EDIRTY	89	*Mounting a dirty file system w/o force.* The file system is not clean and **M_FORCE** is not set.
EDOM	33	*Argument out of domain.* The value of the parameter is a Not a Number (NaN).
EDQUOT	69	*Disc quota exceeded.* The file system of the requested directory has exceeded the user's quota of disk blocks.
EDUPPKG	90	*Duplicate package name.* The loaded module exported a package which duplicated the package name of a module already loaded in the same process.
EEXIST	17	*File exists.* The requested file already exists.
EFAULT	14	*Bad address.* The requested address is in some way invalid.

ERRNO *(cont.)*

ERRNO *(cont.)*

EFBIG	27	*File too large.* The file size exceeds the process' file size limit, or the requested semaphore number is invalid.
EFSNOTSUPP	210	*Operation not supported by this file system.*
EHOSTDOWN	64	*Host is down.*
EHOSTUNREACH	65	*Host is unreachable.*
EIDRM	81	*Identifier removed.* The requested semaphore or message queue ID has been removed from the system.
EIMODE	202	*Bad io mode number.* Use the I/O mode **M_UNIX, M_LOG, M_SYNC,** or **M_RECORD.**
EINPROGRESS	36	*Operation now in progress.*
EINTR	4	*Interrupted system call.* The operation was interrupted by a signal.
EINVAL	22	*Invalid argument.* The argument or parameter is not valid for the system call.
EIO	5	*I/O error.* An I/O error occurred while reading or writing to the file system.
EISCONN	56	*Socket is already connected.* The socket is already connected.
EISDIR	21	*Is a directory.* The request is for a write to a file but the specified file name is a directory, or the function is trying to rename a file as a directory.
ELOCAL	103	*Handle operation locally.*
ELOOP	62	*Too many levels of symbolic links.* Too many symbolic links were encountered in translating a pathname.
EMFILE	24	*Too many open files.* Too many files descriptors are open, no space remains in the mount table, or the attempt to attach a shared memory region exceeded the maximum number of attached regions for a process.
EMIXIO	201	*Mixed file operations.* See the **setiomode**() function.

ERRNO *(cont.)*

ERRNO *(cont.)*

EMLINK	31	*Too many links.* The number of links would exceed **LINK_MAX**.
EMSGSIZE	40	*Message too long.* The message is too large to be sent all at once, as the socket requires.
ENAMETOOLONG	63	*File name too long.* The pathname argument exceeds **PATH_MAX** (1024 characters) or the pathname component exceeds **NAME_MAX** (255 characters).
ENETDOWN	50	*Network is down.*
ENETRESET	52	*Network dropped connection on reset.*
ENETUNREACH	51	*Network is unreachable.* No route to the network or host is present.
ENFILE	23	*File table overflow.* Too many files are currently open in the system.
ENFPS	200	*Different file pointers.*
ENOBUFS	55	*No buffer space available.* Insufficient resources, such as buffers, are available to complete the call.
ENOCFS	204	*No CFS available.* The concurrent file system (CFS) is not available.
ENODATA	86	*No message on stream head read q.*
ENODEV	19	*No such device.* The file descriptor refers to an object that cannot be mapped, the requested block special device file does not exist, or a file system is unmounted.
ENOENT	2	*No such file or directory.* A pathname component of the parameter does not exist.
ENOEXEC	8	*Exec format error.* The parameter specifies a file with a bad object file format.
ENOLCK	77	*No locks available.* The lock table is full because too many regions are already locked.

ERRNO *(cont.)*

ERRNO *(cont.)*

ENOMEM	12	*Not enough space.* Insufficient memory is available for the requested function.
ENOMSG	80	*No message of desired type.* A message of a requested type does not exist.
ENOPKG	92	*Unresolved package name.* One or more unresolved package names were found.
ENOPROTOOPT	42	*Option not supported by protocol.* The option is unknown.
ENOSPC	28	*No space left on device.* There is not enough memory space to extend the file system or device for file or directory writes.
ENOSR	82	*Out of STREAMS resources.*
ENOSTR	87	*fd not associated with a stream.*
ENOSYM	93	*Unresolved symbol name.* One or more unresolved external symbols were found.
ENOSYS	78	*Function not implemented.*
ENOTBLK	15	*Block device required.* The specified device is not a block device.
ENOTCONN	57	*Socket is not connected.* The socket is not connected.
ENOTDIR	20	*Not a directory.* A component of the pathname is not a directory.
ENOTEMPTY	66	*Directory not empty.*
ENOTSOCK	38	*Socket operation on non-socket.* The parameter refers to a file not a socket.
ENOTTY	25	*Not a typewriter.* The specified request does not apply to the kind of object that the descriptor references.
ENXIO	6	*No such device or address.* The device or address does not exist.

ERRNO *(cont.)*

ERRNO *(cont.)*

EOPNOTSUPP	45	*Operation not supported on socket.* The socket does not support the requested operation, or the socket does not accept the connection.
EPACCES	139	*Partition permission denied.*
EPALLOCERR	130	*Allocator internal error.*
EPBADNODE	132	*Bad node specification.*
EPERM	1	*Not owner.* The calling process does not have permissions for the operation.
EPFNOSUPPORT	46	*Protocol family not supported.*
EPINGRP	160	*Invalid group.*
EPINRN	161	*Invalid partition rename.* Use a simple name for a partition name.
EPINUSER	159	*Invalid user.*
EPINVALMOD	136	*Invalid mode.*
EPINVALPART	133	*Partition not found.*
EPINVALPRI	134	*Invalid priority.*
EPINVALSCHED	138	*Invalid Scheduling.*
EPIPE	32	*Broken pipe.* An attempt was made to write to a pipe or FIFO that is not open for reading by any process.
EPLOCK	162	*Partition lock denied.* You specified a partition that is currently in use and being updated by someone else. You cannot change the characteristics of a partition that is currently being used.
EPNOTEMPTY	135	*Partition not empty.*
EPPARTEXIST	137	*Partition exists.*
EPROCLIM	67	*Too many processes.*
EPROCUNAVAIL	76	*Bad procedure for program.*

ERRNO *(cont.)*

ERRNO *(cont.)*

EPROGMISMATCH	75	*Program version wrong.*
EPROGUNAVAIL	74	*RPC program not available.*
EPROTO	85	*Error in protocol.*
EPROTONOSUPPORT	43	*Protocol not supported.* The socket or protocol is not supported.
EPROTOTYPE	41	*Protocol wrong type for socket.*
EPXRS	131	*Exceeds partition resources.*
EQBADFIL	183	*Invalid object file.* Specify a loadable file.
EQBLEN	171	*Buffer length exceeds allocation.* Make sure the buffer length does not exceed the buffer size.
EQDIM	195	*Invalid dimension.*
EQESIZE	205	*Invalid size.*
EQHND	179	*Invalid handler type.* Select one of the handlers listed in the handler description.
EQLEN	172	*Invalid length.* Use a non-negative number or a length that is less than or equal to the maximum message length.
EQMEM	190	*Not enough memory.* Simplify the application program.
EQMID	178	*Invalid message id.* Use the message ID (MID) returned by the **irecv()** or **isend()** functions.
EQMODE	196	*Invalid diagnostic channel mode.*
EQMSGLONG	174	*Received message too long for buffer.* Make sure the buffer is large enough to hold the message.
EQMSGSHORT	198	*Received message too short for buffer.*
EQNOACT	182	*No active process.* Use the process ID (PID) of a loaded process.

ERRNO *(cont.)*

ERRNO *(cont.)*

EQNODE	176	*Invalid node.* Use the **numnodes**() function to determine the partition size and the **myhost**() function to determine the host node number.
EQNOMID	191	*Too many requests.* Use the **msgwait**() function for outstanding requests from the **irecv**() or **isend**() functions.
EQNOPROC	180	*Out of process slots.* Use fewer processes.
EQNOSET	193	*No ptype defined.*
EQPARAM	184	*Invalid parameter.*
EQPATH	207	*Path name too long.*
EQPBUF	170	*Invalid buffer pointer.* Specify a pointer that contains the address of a valid data buffer.
EQPCCODE	188	*Invalid ccode pointer.*
EQPCNODE	186	*Invalid cnode pointer.*
EQPCPID	187	*Invalid cpid pointer.* Do not call the **setpid**() function again.
EQPFIL	185	*Invalid file name pointer.*
EQPGRP	209	*Supplied processes group does not exist or is under control of another TAM.*
EQPID	175	*Invalid ptype.* The PID must not be negative.
EQPRIV	189	*Privileged operation.*
EQSET	192	*Ptype already set.*
EQSTATUS	197	*Invalid diagnostic channel status.*
EQTAM	208	*Max number of applications under debug was reached.*
EQTIME	173	*Time limit exceeded.*
EQTYPE	177	*Invalid type.* Use a non-negative number.

ERRNO *(cont.)*

ERRNO *(cont.)*

EQUSEPID	181	*Ptype already in use.* Select another PID.
EQUSM	194	*Invalid diagnostic channel usm id.*
ERANGE	34	*Result too large.* The symbol address could not be converted into an absolute value.
ERDEOF	206	*Attempt to read past end of file.*
EREMOTE	71	*Item is not local to host.*
EREMOTEPORT	102	*Returned port is remote.*
ERFORK	140	*Do an rfork instead of a fork.*
EROFS	30	*Read-only file system.* The directory in which the file is to be created is located on a read-only file system.
ERPCMISMATCH	73	*RPC version is wrong.*
ESETIO	203	*File is not synchronized.* In I/O modes **M_SYNC** and **M_RECORD**, all nodes must read or write synchronously.
ESHUTDOWN	58	*Can't send after socket shutdown.*
ESOCKTNOSUPPORT	44	*Socket type not supported.*
ESPIPE	29	*Illegal seek.* An invalid seek operation was requested for a pipe (FIFO), socket, or multiplexed special file.
ESRCH	3	*No such process.* The requested process or child process ID is invalid, no disk quota is found for the specified user, or the specified thread ID does not refer to an existing thread.
ESTALE	70	*Missing file or file system.* The process' root or current directory is located in a virtual file system that has been unmounted.
ETIME	83	*System call timed out.*
ETIMEDOUT	60	*Connection timed out.* The establishment of the connection timed out before the connection could be made.

ERRNO *(cont.)*

ERRNO *(cont.)*

ETOOMANYREFS	59	*Too many references: can't splice.*
ETXTBSY	26	*Text file busy.* The file is currently opened for writing by another process, or a write access is requested by a pure procedure (shared text) file that is being executed.
EUSERS	68	*Too many users.* There are too many users.
EVERSION	91	*Version mismatch.*
EWOULDBLOCK	35	*Operation would block.* The file is locked, but blocking is not set. The socket is marked nonblocking, so the connection cannot be completed.
EXDEV	18	*Cross-device link.* The link and the file are on different file systems.

See Also

***application*, nx_perror(), perror(3)**

ESEEK() ESEEK()

Moves the read-write file offset.

Synopsis

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

esize_t eseek(
        int fildes,
        esize_t offset,
        int whence );
```

Parameters

fildes File descriptor for an open extended file or standard OSF/1 file. A standard OSF/1 file cannot have a resulting size greater than 2G -1 bytes.

offset The value, in bytes, to be used in conjunction with the *whence* parameter to set the file pointer.

whence Specifies how to interpret the *offset* parameter in setting the file pointer associated with the *fildes* parameter. Values for the *whence* parameter are as follows (defined in *unistd.h*):

SEEK_SET Sets the file pointer to *offset* bytes from the beginning of the file.

SEEK_CUR Sets the file pointer to its current location plus *offset* bytes.

SEEK_END Sets the file pointer to the size of the file plus the value of *offset* bytes.

Description

Other than the return value and the additional errors discussed below, the **eseek()** function behavior is identical to the OSF/1 **lseek()** function. See **lseek(2)** in the *OSF/1 Programmer's Reference.*

Extended files are files that support the maximum file size of the Paragon™ OSF/1 operating system. Standard OSF/1 files cannot have a size greater than 2G - 1 bytes.

ESEEK() *(cont.)* ESEEK() *(cont.)*

Return Values

Upon successful completion, the **eseek()** function returns an extended integer (**esize_t**) that is the new position of the file pointer measured in bytes from the beginning of the file. Otherwise, the **eseek()** function displays an error message to standard error and causes the calling process to terminate. The **esize_t** structure has the following format (see the *nx.h* include file):

```
struct s_size {
        long    slow;
        long    shigh;
};
typedef struct s_size esize_t;
```

Upon successful completion, the **_eseek()** function returns the same value as the **eseek()** function. Otherwise, the **_eseek()** function returns -1 in all fields of the **esize_t** structure and sets *errno* to indicate the error.

Errors

If the **_eseek()** function fails, *errno* may be set to one of the error code values described for the OSF/1 **lseek(2)** function or to one of the following values:

ECFPS In I/O modes **M_SYNC**, **M_RECORD**, or **M_GLOBAL**, nodes are attempting to seek to different positions in a shared file. In these modes, any seeks must be performed by all nodes to the same file position.

EMIXIO In I/O modes **M_SYNC** or **M_GLOBAL**, nodes are attempting different operations to a shared file. In these modes, all nodes must perform the same operation.

EFBIG The file size specified by the *whence* and *offset* parameters exceeds the maximum file size.

ECFPS Two or more application processes are calling **eseek()** with different shared I/O modes (**M_SYNCH**, **M_RECORD**, or **M_GLOBAL**).

See Also

cread(), **cwrite()**, **esize()**, **iread()**, **iseof()**, **iwrite()**, **setiomode()**

OSF/1 Programmer's Reference: **fcntl(2)**, **lseek(2)**, **open(2)**

ESIZE() ESIZE()

Increases the size of a file.

Synopsis

```
#include <nx.h>

esize_t esize(
        int fildes,
        esize_t offset,
        int whence );
```

Parameters

fildes File descriptor for an extended file or standard OSF/1 files open for writing. A standard OSF/1 file cannot have a resulting size greater than 2G - 1 bytes.

offset Value, in bytes, to be used in conjunction with the *whence* parameter to set the file size.

whence Specifies how to interpret the *offset* parameter in increasing the size of the file associated with the *fildes* parameter. Values for the *whence* parameter are as follows (defined in *nx.h*):

SIZE_SET Sets the file size to the greater of the current size or to the value of the *offset* parameter.

SIZE_CUR Sets the file size to the greater of the current size or the current location of the file pointer plus the value of the *offset* parameter.

SIZE_END Sets the file size to the greater of the current size or the current size plus the value of the *offset* parameter.

ESIZE() *(cont.)* ESIZE() *(cont.)*

Description

The **esize**() function increases the size of a file. This function cannot decrease the size of a file.

The **esize**() function supports both extended files and standard OSF/1 files. Extended files are files that support the maximum file size of the Paragon™ OSF/1 operating system. Standard OSF/1 files cannot have a size greater than 2G - 1 bytes. You can also use the **lsize**() function to change the size of standard OSF/1 files.

Use the **esize**() function to allocate sufficient file space before starting performance-sensitive calculations or storage operations. This increases an application's throughput, because the I/O system does not have to allocate data blocks for every write that extends the file size.

The **esize**() function does not affect FIFO special files, directories, or the position of the file pointer. The contents of the new file space allocated by **esize**() is undefined.

The **esize**() function updates the modification time of the opened file. If the file is a regular file it clears the file's set-user ID and set-group ID attributes.

You cannot use the **esize**() function with a file that has enforced file locking enabled and file locks on the file.

Note

If the new size specified by *offset* and *whence* is greater than the available disk space, **esize()** allocates what disk space is available and returns the actual new size.

Return Values

Upon successful completion, the **esize**() function returns an extended integer (type **esize_t**) that indicates the new size of the file (in bytes). Otherwise, the **esize**() function displays an error message to standard error and causes the calling process to terminate. The type **esize_t** has the following structure (see the *nx.h* include file):

```
struct s_size {
        long    slow;
        long    shigh;
};
typedef struct s_size esize_t;
```

ESIZE() *(cont.)* ESIZE() *(cont.)*

Upon successful completion, the **_esize()** function returns an extended integer that indicates the new size of the file (in bytes). Otherwise, the **_esize()** function returns -1 in all fields of the **esize_t** structure and sets *errno* to indicate the error.

Errors

If the **_esize()** function fails, *errno* may be set to one of the following error code values:

EAGAIN The file has enforced mode file locking enabled and there are file locks on the file.

EACCES Write access permission to the file was denied.

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

EFBIG The file size specified by the *whence* and *offset* parameters exceeds the maximum file size.

EFSNOTSUPP The *fildes* parameter refers to a file that resides in a file system that does not support this operation.

EINVAL The file is not a regular file.

ENOSPC No space left on device.

EROFS The file resides on a read-only file system.

See Also

eseek(), lsize()

OSF/1 Programmer's Reference: **chmod(2), dup(2), fcntl(2), lseek(2), open(2)**

ESTAT() ESTAT()

estat(), **festat**(): Gets status of an file.

Synopsis

```
#include <nx.h>

long estat(
        char *path,
        struct estat *buffer );

long festat(
        int fildes,
        struct estat *buffer );
```

Parameters

path	Pointer to the pathname identifying a file.
buffer	Pointer to an *estat* structure in which the status information is placed. The *estat* structure is described in the *sys/estat.h* header file.
fildes	A file descriptor representing an open file.

Description

The **estat**() function semantics are identical to the OSF/1 **stat**() and **fstat**() functions. See the **stat**(2) manual page in the *OSF/1 Programmer's Reference*.

The **estat**() function gets information about the file named by the *path* parameter. Read, write, or execute permission for the named file is not required, but all directories listed in the pathname leading to the file must be searchable. The file information is written to the area specified by the *buffer* parameter, which is a pointer to an *estat* structure, defined in *sys/estat.h*.

The **festat**() function is identical to the **estat**() function except that the information obtained is about an open file referenced by the *fildes* parameter.

ESTAT() *(cont.)* ESTAT() *(cont.)*

Return Values

Upon successful completion, the **estat()** and **festat()** functions return a value of 0 (zero). Otherwise, these functions return a value of `-1`, display an error message to standard error, and cause the calling process to terminate.

Upon successful completion, the **_estat()** and **_festat()** functions return a value of 0 (zero). Otherwise, these functions return -1 and set *errno* to indicate the error.

Errors

If the **_estat()** or **_festat()** functions fail, *errno* may be set to one of the error code values described for the OSF/1 **stat()** function.

See Also

eseek(), **esize()**

OSF/1 Programmer's Reference: **dup(2)**, **open(2)**, **stat(2)**

ETOS() ETOS()

etos(), stoe(): Converts an extended integer to a string or a string to an extended integer.

Synopsis

```
#include <nx.h>

void etos(
        esize_t e,
        char *s );

esize_t stoe(
        char *s );
```

Parameters

e	An extended integer.
s	Pointer to a character string.

Description

Use these functions to perform the following operations:

etos()	Converts an extended integer to a character string.
stoe()	Converts a string of characters to an extended integer.

Return Values

On successful completion, the **etos()** function returns control to the calling process; no values are explicitly returned. On successful completion, the **stoe()** function returns control to the calling process and returns an extended integer (type **esize_t**). Otherwise, these functions display an error message to standard error and cause the calling process to terminate. If an error occurs, the **stoe()** function returns `-1` in all fields of the **esize_t** structure.

ETOS() *(cont.)*

ETOS() *(cont.)*

The **esize_t** structure has the following format (see the *nx.h* include file):

```
struct s_size {
        long    slow;
        long    shigh;
};
typedef struct s_size esize_t;
```

Upon successful completion, the **_etos()** function returns 0 (zero) and the **_stoe()** function returns an extended integer. Otherwise, the **_etos()** function returns -1 and sets *errno* to indicate the error. The **_stoe()** function returns - 1 in all fields of the **esize_t** return structure and sets *errno* to indicate the error.

Errors

If the **_etos()** or **_stoe()** functions fail, *errno* may be set to the following error code value:

EQESIZE Argument is too large. The size of the extended integer must be less than 2^{60} or overflow occurs.

See Also

eadd(), **ecmp()**, **ediv()**, **emod()**, **emul()**, **eseek()**, **esub()**

FLICK() FLICK()

Gives control of the node processor to the operating system for as long as 10 milliseconds.

Synopsis

```
#include <nx.h>

void flick(void);
```

Description

The **flick**() function temporarily releases control of the node processor to another process in the same application. If there are no other processes in the same application when a process calls the **flick**() function, control returns to the operating system. For example, if your application has several handler-receive operations set up and nothing else to do, it should call the **flick**() function. The operating system can then more efficiently respond to an incoming message and wake up your process.

The **flick**() function does not affect an application's rollin or rollout.

How the **flick**() function works depends on whether the calling process is the only process on the node or there are multiple processes on the node:

- If the calling process is the only process on the node, the **flick**() function suspends execution of the calling process and gives control of the node to the operating system until any interrupt occurs. The operating system handles the interrupt and returns control of the node to the calling process. This improves performance by eliminating interrupt overhead; the operating system does not have to take control of the node before handling the interrupt. The operating system never retains control of the node longer than 10 milliseconds; the internal clock generates an interrupt at 10 millisecond intervals.

- If there are multiple processes on the node, the **flick**() function suspends the calling process and gives control to the next scheduled process on the node. The calling process resumes executing when it is next scheduled to execute. This provides higher performance because control passes to the next scheduled process immediately and the scheduler does not intervene.

Return Values

Upon successful completion, the **flick**() function returns control to the calling process; no values are returned. Otherwise, this function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_flick**() function returns `0` (zero). Otherwise, this function returns `-1` and sets *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

errno

OSF/1 Programmer's Reference: **sleep**()

FLUSHMSG() FLUSHMSG()

Removes (flushes) messages from the system buffers.

Synopsis

```
#include <nx.h>

void flushmsg(
        long typesel,
        long nodesel,
        long ptypesel );
```

Parameters

typesel	Message type(s) to remove. Setting this parameter to -1 flushes messages of any type. Refer to Appendix A of the *Paragon™ OSF/1 C System Calls Reference Manual* for more information about message type selectors.
nodesel	Node number of the message destination (that is, the receiving node). Setting this parameter to -1 flushes a message from any node.
ptypesel	Process type of the receiver. Setting *ptypesel* to -1 flushes a message for any process type.

Description

The **flushmsg**() function removes from system buffers all messages of the specified type(s) sent to the specified node and process type.

The *ptypesel* parameter specifies the process type for the receiving process not the sending process. The **flushmsg**() function has no effect on messages sent to processes that do not have the same process type as the *ptypesel* parameter, even if the sending process has a matching process type. The **flushmsg**() function affects only messages that have arrived but not received.

To ensure that all messages are removed, first use the **msgcancel**() function to cancel any messages in transit, then use the **flushmsg**() function to flush any messages that have arrived but have not yet been received.

FLUSHMSG() *(cont.)*

FLUSHMSG() *(cont.)*

Return Values

Upon successful completion, the **flushmsg**() function returns control to the calling process; no values are returned. Otherwise, this function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_flushmsg**() function returns `0` (zero). Otherwise, this function returns `-1` and sets *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

errno, msgcancel()

FPGETROUND() FPGETROUND()

fpgetround(), **fpsetround**(), **fpgetmask**(), **fpsetmask**(), **fpgetsticky**(), **fpsetsticky**(): IEEE floating-point environment control.

Synopsis

```
#include <ieeefp.h>

fp_rnd fpgetround( void );

fp_rnd fpsetround(
        fp_rnd rnd_dir );

fp_except fpgetmask( void );

fp_except fpsetmask(
        fp_except mask );

fp_except fpgetsticky( void );

fp_except fpsetsticky(
        fp_except sticky );
```

Parameters

rnd_dir The new rounding mode for the calling process. Must be one of the following values:

FP_RN or 0	Round to nearest representable number (if two representable numbers are equidistant, round to the even one).
FP_RM or 1	Round toward minus infinity.
FP_RP or 2	Round toward plus infinity.
FP_RZ or 3	Round toward zero (truncate).

These are the only valid values for the **enum** type **fp_rnd**, which is declared in *<ieeefp.h>*.

mask The new exception mask for the calling process. You can create this mask value by **OR**-ing together the following constants, which are defined in *<ieeefp.h>*:

FP_X_INV	Invalid operation exception.
FP_X_DZ	Divide-by-zero exception.
FP_X_OFL	Overflow exception.
FP_X_UFL	Underflow exception.
FP_X_IMP	Imprecise (loss of precision) exception.

sticky The new exception sticky flags for the calling process. You can create this value by **OR**-ing together the same constants used for *mask*.

Description

The **fpget...**() and **fpset...**() functions get and set the i860™ microprocessor's floating-point rounding mode, exception flags, and exception sticky flags for the calling process.

The floating-point rounding mode determines what happens when a floating-point value generated in a calculation cannot be represented exactly. You can use **fpgetround**() to determine the current rounding mode and **fpsetround**() to set the rounding mode.

NOTE

When you convert a floating-point value to an integer type in C, it always truncates. The processor's rounding mode is ignored.

There are six floating-point exceptions: divide by zero, overflow, underflow, imprecise (inexact) result, denormalization, and invalid operation. When one of these exceptions occurs, the corresponding exception sticky flag is set to 1. If the corresponding exception mask bit is set to 1, the exception is trapped. You can use **fpgetsticky**() and **fpsetsticky**() to get and set the exception sticky flags, and **fpgetmask**() and **fpsetmask**() to get and set the exception mask.

NOTE

fpsetsticky() and **fpsetmask()** set the sticky flags and exception mask to the specified values. Any bits not set in the *mask* or *sticky* argument are cleared.

To set or clear a particular mask or sticky flag, get the current mask or sticky flags, modify it, and then call **fpsetsticky**() or **fpsetmask**() with the modified mask or sticky flags.

NOTE

After an exception, you must clear the corresponding sticky flag to recover from the trap and proceed.

If the sticky flag is not cleared before the next floating-point exception occurs, an incorrect exception type may be signaled. For the same reason, when you call **fpsetmask**(), you must be sure that the sticky flag corresponding to each exception being enabled is cleared.

Return Values

Upon successful completion, the **fpget...**() and **fpset...**() functions return the following values and return control to the calling process:

fpgetround() Returns the current rounding mode.

fpsetround() Returns the previous rounding mode.

fpgetmask() Returns the current exception mask.

fpsetmask() Returns the previous exception mask.

fpgetsticky() Returns the current exception sticky flags.

fpsetsticky() Returns the previous exception sticky flags.

Otherwise, these functions display an error message to standard error and cause the calling process to terminate.

Upon successful completion, the **_fptget...()** and **_fptset...()** functions return the following values:

_fpgetround() Returns the current rounding mode.

_fpsetround() Returns the previous rounding mode.

_fpgetmask() Returns the current exception mask.

_fpsetmask() Returns the previous exception mask.

_fpgetsticky() Returns the current exception sticky flags.

_fpsetsticky() Returns the previous exception sticky flags.

Otherwise, these functions return `-1` and set *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

errno, **isnan()**

GCOL() GCOL()

Collects contributions from all nodes. (Global concatenation operation)

Synopsis

```
#include <nx.h>

void gcol(
        char x[],
        long xlen,
        char y[],
        long ylen,
        long *ncnt );
```

Parameters

x	Pointer to the input buffer to be used in the operation. This parameter can be of any type.
xlen	Length (in bytes) of *x*.
y	Pointer to the output buffer to be used in the operation (*y* contains the desired result). This parameter must be of the same data type as *x*.
ylen	Length (in bytes) of *y*.
ncnt	Pointer to the number of bytes returned in *y*.

Description

The **gcol()** function collects and concatenates (in node number order) a contribution from each node in the current application. The *x* and *y* parameters can be of any data type, but they must be of the same data type. The result is returned in *y* to every node.

Problems that involve computing matrix vector products by allowing the nodes to compute partial answers can use **gcol()** to collect and concatenate the entire vector.

If the lengths of the contributions from all the nodes are known, use **gcolx()** instead of **gcol()**.

GCOL() *(cont.)*

GCOL() *(cont.)*

This is a "global" operation, which means that all nodes in the application must execute this operation before the process can continue on any node, and all participating processes must have the same process type.

Return Values

Upon successful completion, the **gcol()** function returns control to the calling process; no values are returned. Otherwise, this function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_gcol()** function returns `0` (zero). Otherwise, this function returns `-1` and sets *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

errno, **gcolx()**, **gdhigh()**, **gdlow()**, **gdprod()**, **gdsum()**, **giand()**, **gior()**, **gopf()**, **gsync()**

GCOLX() GCOLX()

Collects contributions of known length from all nodes. (Global concatenation operation for contributions of known length)

Synopsis

```
#include <nx.h>

void gcolx(
        char x[],
        long xlens[],
        char y[] );
```

Parameters

x	Pointer to the input buffer to be used in the operation. This parameter may be of any type.
xlens	Pointer to an array containing the length (in bytes) of the input buffer *x* expected on each node. The elements in *xlens* must be in increasing node number order.
y	Pointer to the output buffer to be used in the operation (*y* receives the desired result). This parameter must be of the same data type as *x*.

Description

The **gcolx**() function globally collects and concatenates (in node number order) a contribution of specified length from each node in the current application. The *x* and *y* parameters can be of any data type, but they must be of the same data type. The result is returned in *y* to every node. By providing the expected length of each contribution, **gcolx**() improves the speed of this operation compared to **gcol**() due to the reduced overhead of calculating where each contribution belongs in the output buffer.

If the lengths of the contributions from all the nodes are unknown, use **gcol**() instead of **gcolx**().

This is a "global" operation, which means that all nodes in the application must execute this operation before the process can continue on any node, and all participating processes must have the same process type.

GCOLX() *(cont.)*

GCOLX() *(cont.)*

Return Values

Upon successful completion, the **gcolx**() function returns control to the calling process; no values are returned. Otherwise, this function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_gcolx**() function returns `0` (zero). Otherwise, this function returns `-1` and sets *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

errno, **gcol**(), **gdhigh**(), **gdlow**(), **gdprod**(), **gdsum**(), **gopf**(), **giand**(), **gior**(), **gsync**()

GDHIGH() GDHIGH()

gdhigh(), **gihigh**(), **gshigh**(): Determines the maximum value across all nodes. (Global maximum operation)

Synopsis

```
#include <nx.h>

void gdhigh(
        double x[],
        long n,
        double work[] );


void gihigh(
        long x[],
        long n,
        long work[] );


void gshigh(
        float x[],
        long n,
        float work[] );
```

Parameters

x	Pointer to the buffer that contains the final result of the global maximum operation.
n	Number of elements in *x*.
work	Pointer to the buffer that receives the contributions from other nodes. The number of elements in *work* must be at least *n*.

GDHIGH() *(cont.)* GDHIGH() *(cont.)*

Description

Use the following functions to determine maximum values across nodes:

- Use **gdhigh**() to determine the double precision maximum value of *x* across all nodes.
- Use **gihigh**() to determine the integer maximum value of *x* across all nodes.
- Use **gshigh**() to determine the float maximum value of *x* across all nodes.

The result is returned in *x* to every node. When *x* is a vector, each element of the resulting vector represents the maximum of the corresponding vector elements of all nodes.

This is a "global" operation, which means that all nodes in the application must execute this operation before the process can continue on any node, and all participating processes must have the same process type.

Return Values

Upon successful completion, the **gdhigh**(), **gihigh**(), and **gshigh**() functions return control to the calling process; no values are returned. Otherwise, these functions display an error message to standard error, and cause the calling process to terminate.

Upon successful completion, the **_gdhigh**(), **_gihigh**(), and **_gshigh**() functions return `0` (zero). Otherwise, these functions return `-1` and set *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

errno, **gcol**(), **gcolx**(), **gdlow**(), **gdprod**(), **gdsum**(), **giand**(), **gior**(), **gopf**(), **gsync**()

GDLOW() GDLOW()

gdlow(), **gilow**(), **gslow**(): Determines the minimum value across all nodes. (Global minimum operation)

Synopsis

```
#include <nx.h>

void gdlow(
        double x[],
        long n,
        double work[] );

void gilow(
        long x[],
        long n,
        long work[] );

void gslow(
        float x[],
        long n,
        float work[] );
```

Parameters

x	Pointer to the buffer that contains the final result of the global minimum operation.
n	Number of elements in *x*.
work	Pointer to the buffer that receives the contributions from other nodes. The number of elements in *work* must be at least *n*.

GDLOW() *(cont.)* GDLOW() *(cont.)*

Description

Use the following functions to determine minimum values across nodes:

- Use **gdlow()** to determine the double precision minimum value of *x* across all nodes.
- Use **gilow()** to determine the integer minimum value of *x* across all nodes.
- Use **gslow()** to determine the float minimum value of *x* across all nodes.

The result is returned in *x* to every node. When *x* is a vector, each element of the resulting vector represents the minimum of the corresponding vector elements of all nodes.

This is a "global" operation, which means that all nodes in the application must execute this operation before the process can continue on any node, and all participating processes must have the same process type.

Return Values

Upon successful completion, the **gdlow()**, **gilow()**, and **gslow()** functions return control to the calling process; no values are returned. Otherwise, these functions display an error message to standard error, and cause the calling process to terminate.

Upon successful completion, the **_gdlow()**, **_gilow()**, and **_gslow()** functions return `0` (zero). Otherwise, these functions return `-1` and set *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

errno, **gcol()**, **gcolx()**, **gdhigh()**, **gdprod()**, **gdsum()**, **giand()**, **gior()**, **gopf()**, **gsync()**

GDPROD() GDPROD()

gdprod(), **giprod**(), **gsprod**(): Calculates a product across all nodes. (Global multiplication operation)

Synopsis

```
#include <nx.h>

void gdprod(
        double x[],
        long n,
        double work[] );

void giprod(
        long x[],
        long n,
        long work[] );

void gsprod(
        float x[],
        long n,
        float work[] );
```

Parameters

x	Pointer to the buffer that contains the result of the global multiplication operation.
n	Number of elements in *x*.
work	Pointer to the buffer that receives the contributions from other nodes. The number of elements in *work* must be at least *n*.

GDPROD() *(cont.)* GDPROD() *(cont.)*

Description

Use the following functions to calculate products across nodes:

- Use **gdprod**() to calculate the double precision product of *x* across all nodes.
- Use **giprod**() to calculate the integer product of *x* across all nodes.
- Use **gsprod**() to calculate the float product of *x* across all nodes.

The result is returned in *x* to every node. When *x* is a vector, each element of the resulting vector represents the product of the corresponding vector elements of all nodes.

This is a "global" operation, which means that all nodes in the application must execute this operation before the process can continue on any node, and all participating processes must have the same process type.

Return Values

Upon successful completion, the **gdprod**(), **giprod**(), and **gsprod**() functions return control to the calling process; no values are returned. Otherwise, these functions display an error message to standard error and cause the calling process to terminate.

Upon successful completion, the **_gdprod**(), **_giprod**(), and **_gsprod**() functions return `0` (zero). Otherwise, these functions return `-1` and set *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

errno, **gcol**(), **gcolx**(), **gdhigh**(), **gdlow**(), **gdsum**(), **giand**(), **gior**(), **gopf**(), **gsync**()

GDSUM() GDSUM()

gdsum(), **gisum**(), **gssum**(): Calculates a sum across all nodes. (Global addition operation)

Synopsis

```
#include <nx.h>

void gdsum(
        double x[],
        long n,
        double work[] );

void gisum(
        long x[],
        long n,
        long work[] );

void gssum(
        float x[],
        long n,
        float work[] );
```

Parameters

x	Pointer to the that contains the final result of the global addition operation.
n	Number of elements in *x*.
work	Pointer to the buffer that receives the contributions from other nodes. The number of elements in *work* must be at least *n*.

GDSUM() *(cont.)* GDSUM() *(cont.)*

Description

Use the following functions to calculate sums across nodes:

- Use **gdsum()** to calculate the double precision sum of *x* across all nodes.
- Use **gisum()** to calculate the integer sum of *x* across all nodes.
- Use **gssum()** to calculate the float sum of *x* across all nodes.

The result is returned in *x* to every node. When *x* is a vector, each element of the resulting vector represents the sum of the corresponding vector elements of all nodes.

This is a "global" operation, which means that all nodes in the application must execute this operation before the process can continue on any node, and all participating processes must have the same process type.

Return Values

Upon successful completion, the **gdsum()**, **gisum()**, and **gssum()** functions return control to the calling process; no values are returned. Otherwise, these functions display an error message to standard error, and cause the calling process to terminate.

Upon successful completion, the **_gdsum()**, **_gisum()**, and **_gssum()** functions return `0` (zero). Otherwise, these functions return `-1` and set *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

errno, **gcol()**, **gcolx()**, **gdhigh()**, **gdlow()**, **gdprod()**, **giand()**, **gior()**, **gopf()**, **gsync()**

GIAND() GIAND()

giand(), **gland**(): Performs an AND across all nodes. (Global AND operation)

Synopsis

```
#include <nx.h>

void giand(
        long x[],
        long n,
        long work[] );

void gland(
        long x[],
        long n,
        long work[] );
```

Parameters

x	Pointer to the buffer that contains the final result of the global AND operation.
n	Number of elements in *x*.
work	Pointer to the array that receives the contributions from other nodes. The number of elements in *work* must be at least *n*.

Description

Use the following functions to perform AND operations across all nodes:

- Use **giand**() to calculate the bitwise AND of *x* across all nodes.
- Use **gland**() to calculate the logical AND of *x* across all nodes.

The result is returned in *x* to every node. When *x* is a vector, each element of the resulting vector represents the AND of the corresponding vector elements of all nodes.

This is a "global" operation, which means that all nodes in the application must execute this operation before the process can continue on any node, and all participating processes must have the same process type.

Return Values

Upon successful completion, the **giand**(), and **gland**() functions return control to the calling process; no values are returned. Otherwise, these functions display an error message to standard error and cause the calling process to terminate.

Upon successful completion, the **_giand**(), and **_gland**() functions return `0` (zero). Otherwise, these functions return `-1` and set *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

errno, **gcol**(), **gcolx**(), **gdhigh**(), **gdlow**(), **gdprod**(), **gdsum**(), **gior**(), **gopf**(), **gsync**()

GIOR() GIOR()

gior(), **glor**(): Performs an OR across all nodes. (Global OR operation)

Synopsis

```
#include <nx.h>

void gior(
        long x[],
        long n,
        long work[] );

void glor(
        long x[],
        long n,
        long work[] );
```

Parameters

x	Pointer to the buffer that contains the final result of the global OR operation.
n	Number of elements in *x*.
work	Pointer to the buffer that receives the contributions from other nodes. The number of elements in *work* must be at least *n*.

Description

Use the following functions to perform OR operations across all nodes:

- Use **gior**() to calculate the bitwise OR of *x* across all nodes.
- Use **glor**() to calculate the logical OR of *x* across all nodes.

The result is returned in *x* to every node. When *x* is a vector, each element of the resulting vector represents the OR of the corresponding vector elements of all nodes.

This is a "global" operation, which means that all nodes in the application must execute this operation before the process can continue on any node, and all participating processes must have the same process type.

Return Values

Upon successful completion, the **gior**(), and **glor**() functions return control to the calling process; no values are returned. Otherwise, these functions display an error message to standard error, and cause the calling process to terminate.

Upon successful completion, the **_gior**(), and **_glor**() functions return `0` (zero). Otherwise, these functions return `-1` and set *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

errno, **gcol**(), **gcolx**(), **gdhigh**(), **gdlow**(), **gdprod**(), **gdsum**(), **giand**(), **gopf**(), **gsync**()

GOPF() GOPF()

Makes a global operation of a user-defined function.

Synopsis

```
#include <nx.h>

void gopf(
        char x[],
        long xlen,
        char work[],
        long (*function)() );
```

Parameters

x	Pointer to the buffer that contains the final result of the user-defined function.
xlen	Length (in bytes) of *x*.
work	Pointer to the buffer that receives the contributions from other nodes. The length of *work* must be at least *xlen*.
function	Pointer to the user-defined function to be called. The function is defined separately. The function must be an associative and commutative function of the two vectors *x* and *work* defined above.

Description

The **gopf()** function gives a user-defined function the same global properties as system-defined global communications functions (such as **gdsum()**). These properties are:

- All nodes must call the global routine (in this case, **gopf()**, which in turn calls the user-written function).
- All nodes in the application must complete the call before the process can continue on any node.
- All participating processes must have the same process type.
- Each node calculates the result and stores it in the *x* buffer.

GOPF() *(cont.)*

GOPF() *(cont.)*

- The *work* array receives contributions from other nodes.
- The result is returned in *x* to all nodes.
- The function must be associative and commutative.

Return Values

Upon successful completion, the **gopf()** function returns control to the calling process; no values are returned. Otherwise, this function displays an error message to standard error, and causes the calling process to terminate.

Upon successful completion, the **_gopf()** function returns `0` (zero). Otherwise, this function returns `-1` and sets *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

errno, **gcol()**, **gcolx()**, **gdhigh()**, **gdlow()**, **gdprod()**, **gdsum()**, **giand()**, **gior()**, **gsync()**

GSENDX() GSENDX()

Sends a message to a list of nodes.

Synopsis

```
#include <nx.h>

void gsendx(
        long type,
        char *buf,
        long count,
        long node[],
        long nodecount );
```

Parameters

type	Message type of the message being sent. Refer to Appendix A of the *Paragon™ OSF/1 C System Calls Reference Manual* for information on message types. The message type must be the same for all participating processes, and there must be no other messages of this type in the application.
buf	Pointer to the message buffer containing the message to be sent. The buffer may be any valid data type.
count	Length (in bytes) of the message being sent.
nodes	Pointer to a list of node numbers for the nodes receiving the message.
nodecount	Number of nodes in the *nodes* parameter.

Description

The **gsendx**() function sends a message to a set of nodes specified by the *nodes* parameter. The nodes that receive the message must call **crecv**(), **irecv**(), or **hrecv**() to receive the message. These receive calls must use the message type specified by **gsendx**(). In addition, all participating processes must have the same process type.

GSENDX() *(cont.)*

GSENDX() *(cont.)*

Return Values

Upon successful completion, the **gsendx**() function returns control to the calling process; no values are returned. Otherwise, this function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_gsendx**() function returns `0` (zero). Otherwise, this function returns `-1` and sets *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

errno, crecv(), **csend**(), **csendrecv**(), **irecv**(), **isend**(), **isendrecv**(), **hrecv**(), **hsend**(), **hsendrecv**()

GSYNC() GSYNC()

Synchronizes all node processes in an application. (Global synchronization operation)

Synopsis

```
#include <nx.h>

void gsync(void);
```

Description

When a node process calls the **gsync()** function, it waits until all other nodes in the application call **gsync()** before continuing. All nodes in the application must call **gsync()** before any node in the application can continue. All participating processes must have the same process type.

Return Values

Upon successful completion, the **gsync()** function returns control to the calling process; no values are returned. Otherwise, this function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_gsync()** function returns `0` (zero). Otherwise, this function returns `-1` and sets *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

errno, **gcol()**, **gcolx()**, **gdhigh()**, **gdlow()**, **gdprod()**, **gdsum()**, **giand()**, **gior()**, **gopf()**

HRECV() HRECV()

hrecv(), **hrecvx**(): Posts a receive for a message and returns immediately; invokes a specified handler when the receive completes. (Asynchronous receive with interrupt-driven handler)

Synopsis

```
#include <nx.h>

void hrecv(
        long typesel,
        char *buf,
        long count,
        void (*handler) () );

void hrecvx(
        long typesel,
        char *buf,
        long count,
        long nodesel,
        long ptypesel,
        void (*xhandler) (),
        long hparam );
```

Parameters

typesel Message type(s) to receive. Setting this parameter to `-1` receives a message of any type. Refer to Appendix A of the *Paragon™ OSF/1 C System Calls Reference Manual* for more information about message type selectors.

buf Pointer to the buffer for storing the received message. The buffer can be of any valid data type, but should match the data type of the buffer in the corresponding send operation.

count Length (in bytes) of the *buf* parameter.

handler Pointer to the handler to execute when the receive completes, after a call to the **hrecv**() function. This handler is user-written and must have four parameters only. See the "Description" section for a description of the handler for the **hrecv**() function.

HRECV() *(cont.)* HRECV() *(cont.)*

nodesel Node number of the sender. Setting *nodesel* to `-1` receives a message from any node.

ptypesel Process type of the sender. Setting *ptypesel* to `-1` receives a message from any process type.

xhandler Pointer to the handler to execute when the receive completes, after a call to the **hrecvx()** function. This handler is user-written and must have five parameters only. See the "Description" section for a description of the handler for the **hrecvx()** function.

hparam Integer that is passed directly to the handler specified by the *xhandler* parameter. Typically, the *hparam* value is used by the handler to identify the request that invoked the handler, making it possible to write shared handlers.

Description

The **hrecv()** and **hrecvx()** functions are asynchronous message-passing system calls. After calling a handler receive function, the function posts a receive for a message, specifies a handler to receive the message, and returns immediately. The calling process continues to run until the message arrives. When the message arrives, the calling process is interrupted, the message is stored in the buffer *buf*, and the specified handler is executed.

The handler is code you write to complete the receive and perform any clean-up after the receive. The handler receives information about the message including the message's type, length, sending node, and process type. When the handler returns, the calling process resumes where it left off.

Using the **hrecvx()** function, you can post multiple handler requests with the same shared handler. The **hrecvx()** function is identical to the **hrecv()** function except for an additional parameter, *hparam*. The *hparam* parameter is an integer value that is passed by the **hrecvx()** function to the handler. The handler uses this value to identify which handler request it is servicing.

A handler for the **hrecv()** and **hrecvx()** functions must have the following arguments:

type The message type (specified in the corresponding send operation).

count The message length (in bytes).

node The node that sent the message.

ptype The process type of the process that sent the message.

A handler for the **hrecvx()** function requires a fifth parameter, *hparam*. The *hparam* parameter is an integer passed to the handler that identifies the request invoking the handler.

An example handler for the **hrecv**() function has the following form:

```
void myhandler(
   long type,
   long count,
   long node,
   long ptype );
```

An example handler for the **hrecvx**() function has the following form:

```
void myhandler(
   long type,
   long count,
   long node,
   long ptype,
   long hparam );
```

If the received message is too long for the buffer *buf*, the receive completes with no error returned, and the content of *buf* is undefined. To detect this situation, the handler can examine the value of its *count* argument.

To post a receive and block the calling process until the receive completes, use one of the synchronous receive system calls (for example, **crecv**()). To receive a message and return a message ID (MID), use one of the other asynchronous send system calls (for example, **irecv**()).

To ensure a critical section of code is not interrupted by the execution of the handler, use the **masktrap**() function to protect that section of code.

Once a handler is invoked, no other handler will interrupt until the first handler returns. For this reason, do not use the **longjump**() function within a handler.

Return Values

Upon successful completion, the **hrecv**() and **hrecvx**() functions return control to the calling process; no values are returned. Otherwise, these functions display an error message to standard error and cause the calling process to terminate.

Upon successful completion, the **_hrecv**() and **_hrecvx**() functions returns `0` (zero). Otherwise, these functions return `-1` and set *errno* to indicate the error.

HRECV() *(cont.)* HRECV() *(cont.)*

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

errno, **cprobe()**, **csend()**, **crecv()**, **csendrecv()**, **hsend()**, **hsendrecv()**, **iprobe()**, **isend()**, **irecv()**, **isendrecv()**, **masktrap()**

HSEND() HSEND()

hsend(), **hsendx**(): Sends a message and returns immediately; invokes a specified handler when the send completes. (Asynchronous send with interrupt-driven handler)

Synopsis

```
#include <nx.h>

void hsend(
        long type,
        char *buf,
        long count,
        long node,
        long ptype,
        void (*handler) () );

void hsendx(
        long type,
        char *buf,
        long count,
        long node,
        long ptype,
        void (*xhandler) (),
        long hparam );
```

Parameters

type	Type of the message to send. Refer to Appendix A of the *Paragon™ OSF/1 C System Calls Reference Manual* for information on message types.
buf	Pointer to the buffer containing the message to send. The buffer may be of any valid data type.
count	Number of bytes to send in the *buf* parameter.
node	Node number of the message destination (the receiving node). Setting *node* to `-1` sends the message to all nodes in the application (except the sending node when the value of the *ptype* parameter is the sender's process type).

HSEND() *(cont.)* HSEND() *(cont.)*

ptype Process type of the message destination (the receiving process).

handler Pointer to the handler to execute when the send completes, after calling the **hsend**() function. This handler is user-written and must have five parameters only. See the "Description" section for a description of the handler for the **hsend**() function.

xhandler Pointer to the handler to execute when the send completes, after calling the **hsendx**() function. You must provide this handler and the handler must have five parameters only. See the "Description" section for a description of the handler for the **hsendx**() function.

hparam Integer that is passed directly to the handler specified by the *xhandler* parameter. Typically, the *hparam* value is used by the handler to identify the request that invoked the handler, making it possible to write shared handlers.

Description

The **hsend**() and **hsendx**() functions are asynchronous message-passing system calls. After calling one of these functions, the function starts a sending process and returns immediately. The sending process sends the message in the buffer *buf* to a destination specified by *node*. The calling process continues to run while the send is completing. When the send completes, the sending process interrupts the calling process and executes the specified handler.

The handler is code you write to complete the send and perform clean-up after the receive. The handler receives information about the message including the message's type, length, receiving node, and process type. When the handler returns, the calling process resumes where it left off.

Using the **hsendx**() function, you can post multiple handler requests with the same shared handler. The **hsendx**() function is identical to the **hsend**() function except for an additional parameter, *hparam*. The *hparam* parameter is an integer value that is passed by the **hsendx**() function to the handler. The handler uses this value to identify which request it is servicing.

These are asynchronous system calls. To send a message and block the calling process until the send completes, use one of the synchronous send system calls (for example, the **csend**() function). To send a message and return a message ID (MID), use one of the other asynchronous send system calls (for example, **isend**()).

HSEND() *(cont.)* HSEND() *(cont.)*

A handler for the **hsend**() and **hsendx**() functions must have the following arguments:

type	The message type.
count	The message length (in bytes).
node	The node number that is running the process that receives the message.
ptype	The process type of the node that receives the sent the message.

A handler for the **hsendx**() function requires a fifth parameter, *hparam*. The *hparam* parameter is an integer the handler uses to identify the request invoking the handler.

An example handler for the **hsend**() function has the following form:

```
void myhandler(
   long type,
   long count,
   long node,
   long ptype );
```

An example handler for the **hsendx**() function has the following form:

```
void myhandler(
   long type,
   long count,
   long node,
   long ptype,
   long hparam );
```

To ensure a critical section of code is not interrupted when the handler executes, use the **masktrap**() function to protect that section of code.

Once a handler is invoked, no other handler can interrupt the calling process until the first handler returns. For this reason, do not use the **longjump**() function within a handler.

HSEND() *(cont.)* HSEND() *(cont.)*

Return Values

Upon successful completion, the **hsend()** and **hsendx()** functions return control to the calling process; these functions do not return a value. Otherwise, these functions display an error message to standard error and cause the calling process to terminate.

Upon successful completion, the **_hsend()** and **_hsendx()** functions return `0` (zero). Otherwise, these functions return `-1` and set *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

cprobe(), **csend()**, **crecv()**, **csendrecv()**, **errno**, **hrecv()**, **hsendrecv()**, **iprobe()**, **isend()**, **irecv()**, **isendrecv()**, **masktrap()**

HSENDRECV() HSENDRECV()

Sends a message and posts a receive for a reply; invokes a user-written handler when the receive completes. (Asynchronous send-receive with interrupt-driven handler)

Synopsis

```
#include <nx.h>

void hsendrecv(
        long type,
        char *sbuf,
        long scount,
        long node,
        long ptype,
        long typesel,
        char *rbuf,
        long rcount,
        void (*handler) () );
```

Parameters

type	Type of the message to send. Refer to Appendix A of the *Paragon™ OSF/1 C System Calls Reference Manual* for information on message types.
sbuf	Pointer to the buffer containing the message to send. The buffer may be of any valid data type.
scount	Number of bytes to send in the *sbuf* parameter.
node	Node number of the message destination (the receiving node). Setting *node* to `-1` sends the message to all nodes in the application (except the sending node when *ptype* is the sender's process type).
ptype	Process type of the message destination (the receiving process).
typesel	Message type(s) to receive. Setting this parameter to `-1` receives a message of any type. Refer to Appendix A of the *Paragon™ OSF/1 C System Calls Reference Manual* for more information about message type selectors.
rbuf	Pointer to the buffer for storing the reply. The buffer can be of any valid data type, but should match the data type of the buffer in the corresponding send operation.

HSENDRECV() *(cont.)* HSENDRECV() *(cont.)*

rcount Length (in bytes) of the *rbuf* parameter.

handler Pointer to the handler to execute when the receive completes after a call to the **hrecv**() function. This handler is user-written and must have four parameters only. See the "Description" section for a description of the user-written handler for the **hrecv**() function.

Description

The **hsendrecv**() function is an asynchronous message-passing system call. After calling this function, the function starts a send-receive process, posts a receive for a reply, and returns immediately. The send-receive process sends the message in the buffer *sbuf* to a destination specified by *node*. The calling process continues to run while the send is completing. When the send-receive completes and the message is received in the buffer *rbuf*, the send-receive process interrupts the calling process and executes the specified handler.

The handler is code you write to complete the send-receive and perform any needed clean-up. The handler receives information about the message including the message's type, length, receiving node, and process type. When the handler returns, the calling process resumes where it left off.

When the message arrives, the **hsendrecv**() function passes information about the received message (its type, length, sending node, and process type) to the handler. The handler must have four parameters (which correspond to the message information passed by the receive function):

type The message type (specified in the corresponding send operation).

count The message length (in bytes).

node The node of the process that sent the message.

ptype The process type of the process that sent the message.

The handler must have the following form:

```
void myhandler(
   long type,
   long count,
   long node,
   long ptype );
```

If the received message is too long for the *rbuf* buffer, the receive completes with no error returned, and the content of the *rbuf* buffer is undefined. To detect this situation, the handler should look at the value of its *count* argument.

To ensure that a critical section of code is not interrupted by the execution of the handler, use the **masktrap**() function to protect that section of code.

Once a handler is invoked, no other handler can interrupt until the first handler returns. For this reason, do not use the **longjump**() function within a handler.

Return Values

Upon successful completion, the **hsendrecv**() function returns the length (in bytes) of the received message, and returns control to the calling process. Otherwise, this function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_hsendrecv**() function returns length (in bytes) of the received message. Otherwise, this function returns `-1` and sets *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

cprobe(), **crecv**(), **csend**(), **csendrecv**(), **errno**, **hrecv**(), **hsend**(), **iprobe**(), **irecv**(), **isend**(), **isendrecv**(), **masktrap**()

INFOCOUNT() INFOCOUNT()

infocount(), **infonode**(), **infoptype**(), **infotype**(): Gets information about a pending or received message.

Synopsis

```
#include <nx.h>

long infocount(void);

long infonode(void);

long infoptype(void);

long infotype(void);
```

Description

Use the **info...**() system calls to return information about a pending or received message. Return values are defined only when these system calls are used immediately after completion of one of the following (any of these conditions indicates that a message has arrived):

- A **cprobe**(), **crecv**(), or **msgwait**() system call.
- A **cprobex**() or **crecvx**() system call whose *info* parameter was set to the global array *msginfo*.
- An **iprobe**() or **msgdone**() system call that returns 1. If the *mid* referenced by **msgdone**() represents a "group" of message IDs (that is, it was returned by **msgmerge**()), then return values for the **info...**() system calls are undefined.

Return Values

Upon successful completion, the **info...**() functions return the following information about pending or received messages and return control to the calling process:

infocount()	Returns length in bytes (*count*) of message.
infonode()	Returns node ID (*node*) of sender.
infoptype()	Returns process type (*ptype*) of sender.
infotype()	Returns type (*type*) of message.

Otherwise, these functions display an error message to standard error and cause the calling process to terminate.

Upon successful completion, the **_infocount()**, **_infonode()**, **_infoptype()**, and **_infotype()** functions return the same values as the corresponding non-underscore function. Otherwise, these functions return `-1` and set *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

cprobe(), **crecv()**, **errno**, **iprobe()**, **msgdone()**, **msgmerge()**, **msgwait()**

IODONE() IODONE()

Determines whether an asynchronous read or write operation is complete.

Synopsis

```
#include <nx.h>

long iodone(
        long id );
```

Parameters

id	The non-negative I/O ID returned by an asynchronous read or write system call (for example, **iread**() or **iwrite**()).

Description

The **iodone**() function determines whether the asynchronous read or write operation (for example, **iread**() or **iwrite**()) identified by the *id* parameter is complete. If the operation is complete, this function releases the I/O ID for the operation.

If the **iodone**() function returns 1 (indicating that the I/O operation is complete):

- The buffer contains valid data (if *id* identifies a read operation), or the buffer is available for reuse (if *id* identifies a write operation).
- The I/O ID that identifies the asynchronous read or write *(id)* is released for use in a future asynchronous read or write.

Use the **iowait**() function if you need the blocking version of this function.

NOTE

You must call either the **iowait()** or **iodone()** function after an asynchronous read or write to ensure that the operation is complete and to release the I/O ID.

Return Values

Upon successful completion, the **iodone()** function returns control to the calling process and returns the following values:

`0` The read or write is not yet complete.

`1` The read or write is complete.

If an error occurs, the **iodone()** function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_iodone()** function returns the same values as the **iodone()** function. Otherwise, the **_iodone()** function returns -1 when an error occurs and sets *errno* to indicate the error.

Errors

If the **_iodone()** function fails, *errno* may be set to the following error code value:

EBADID The *id* parameter is not a valid I/O ID.

See Also

iowait(), **iread()**, **iwrite()**

IOMODE() IOMODE()

Gets the I/O mode of a file.

Synopsis

```
#include <nx.h>

long iomode(
        int fildes );
```

Parameters

fildes	A file descriptor representing an open file.

Description

The **iomode**() function determines the current I/O mode of the file identified by *fildes*. A file's I/O mode determines how a process may access the file.

Return Values

Upon successful completion, the **iomode**() function returns control to the calling process and returns the current I/O mode of the file descriptor identified by the *fildes* parameter. The I/O mode can be **M_UNIX**, **M_LOG**, **M_SYNC**, or **M_RECORD**. Refer to the **setiomode**() manual page for descriptions of each I/O mode.

If an error occurs, the **iomode**() function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_iomode**() function returns the same values as the **iomode**() function. Otherwise, the **_iomode**() function returns -1 and sets *errno* to indicate the error.

Errors

If the **_iomode**() function fails, *errno* may be set to the following error code value:

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

IOMODE() *(cont.)*

IOMODE() *(cont.)*

See Also

setiomode()

OSF/1 Programmer's Reference: **dup(2)**, **open(2)**

IOWAIT() IOWAIT()

Waits (blocks) until an asynchronous read or write operation completes.

Synopsis

```
#include <nx.h>

void iowait(
        long id );
```

Parameters

id	The non-negative I/O ID returned by an asynchronous read or write system call (for example, **iread**() or **iwrite**()).

Description

The **iowait**() function waits until an asynchronous read or write function (for example, the **iread**() or **iwrite**() function) identified by *id* completes. When the **iowait**() function returns:

- The buffer contains valid data (if *id* identifies a read operation), or the buffer is available for reuse (if *id* identifies a write operation).
- The I/O ID that identifies the asynchronous read or write (*id*) is released for use in a future asynchronous read or write.

Use the **iodone**() function for the non-blocking version of this function.

NOTE

You must call either the **iowait()** or **iodone()** function after an asynchronous read or write to ensure that the operation is complete and to release the I/O ID.

Return Values

Upon successful completion, the **iowait()** function returns control to the calling process; no values are returned. If an error occurs, the **iowait()** function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_iowait()** function returns the value 0 (zero). Otherwise, the **_iowait()** function returns -1 when an error occurs and sets *errno* to indicate the error.

Errors

If the **_iowait()** function fails, *errno* may be set to the following error code value:

EBADID The *id* parameter is not a valid I/O ID.

See Also

iodone(), **iread()**, **iwrite()**

IPROBE() IPROBE()

iprobe(), **iprobex()**: Determines whether a message is ready to be received. (Asynchronous probe)

Synopsis

```
#include <nx.h>

long iprobe(
        long typesel );

long iprobex(
        long typesel,
        long nodesel,
        long ptypesel,
        long info[] );
```

Parameters

typesel	Message type or set of message types for which to probe. Setting this parameter to `-1` probes for a message of any type. Refer to Appendix A of the *Paragon™ OSF/1 C System Calls Reference Manual* for more information about message type selectors.
nodesel	Node number of the sender. Setting *nodesel* to `-1` probes for a message from any node.
ptypesel	Process type of the sender. Setting *ptypesel* to `-1` probes for a message from any process type.
info	Eight-element array (four bytes per element) in which to store message information. The first four elements contain the message's type, length, sending node, and sending process type. The last four elements are reserved for system use. If you do not need this information, you can specify the global array *msginfo*, which is the array used by the **info...()** system calls.

IPROBE() *(cont.)* IPROBE() *(cont.)*

Description

Use the appropriate asynchronous probe function to determine if the specified message is ready to be received:

- Use the **iprobe**() function to probe for a message of a specified type.
- Use the **iprobex**() function to probe for a message of a specified type from a specified sender and place information about the message in an array.

If the **iprobe**() function returns `1` (indicating that the specified message is ready to be received), you can use the **info...**() system calls to get more information about the message. Otherwise, the **info...**() system calls are undefined.

Similarly, if the **iprobex**() function returns `1`, you can examine the *info* array to get more information about the message. Otherwise, the *info* array is undefined.

These are asynchronous system calls. To probe for a message and block the calling process until the message is ready to be received, use one of the synchronous probe system calls (for example, **cprobe**()).

Return Values

Upon successful completion, the **iprobe**() and **iprobex**() functions return the following values and return control to the calling process:

`0` If the specified message is not available.

`1` If the specified message is available.

Otherwise, these functions display an error message to standard error and cause the calling process to terminate.

Upon successful completion, the **_iprobe**() and **_iprobex**() functions return the following values:

`0` If the specified message is not available.

`1` If the specified message is available.

Otherwise, these functions return `-1` and set *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

cprobe(), errno, infocount(), infonode(), infoptype(), infotype()

IREAD() IREAD()

Reads from a file and returns immediately. (Asynchronous read)

Synopsis

```
#include <nx.h>

long iread(
        int fildes,
        char *buffer,
        unsigned int nbytes );
```

Parameters

fildes	File descriptor identifying the open file to be read.
buffer	Pointer to the buffer in which the data is stored after it is read.
nbytes	Number of bytes to read from the file associated with the *fildes* parameter.

Description

Other than the return values, the additional errors, and the asynchronous behavior (all discussed below), the **iread**() function is identical to the OSF/1 **read**() function. See **read(2)** in the *OSF/1 Programmer's Reference.*

The **iread**() function is an asynchronous version of the **cread**() function. The **iread**() function returns to the calling process immediately; the calling process continues to run while the read is being done. If the calling process needs the data for further processing, it must do one of the following:

- Use **cread**() (a synchronous system call) instead of **iread**()
- Use **iowait**() to wait until the read completes
- Loop until **iodone**() returns `1`, indicating that the read is complete

The asynchronous read system calls modify the file pointer immediately, so **lseek**(), **iseof**(), or a read or write system call can be used immediately without waiting for the read to finish. To determine whether the read operation moved the file pointer to the end of the file, use the **iseof**() system call.

Return Values

Upon successful completion, the **iread**() function returns control to the calling process and returns a non-negative I/O ID for use in **iodone**() and **iowait**() system calls. Otherwise, the **iread**() function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_iread**() function returns a non-negative I/O ID for use in **iodone**() and **iowait**() system calls. Otherwise, the **_iread**() function returns -1 when an error occurs and sets *errno* to indicate the error.

NOTE

The number of I/O IDs is limited, and an error occurs when no I/O IDs are available for a requested asynchronous read or write. Therefore, your program should release the I/O ID as soon as possible by calling **iodone()** or **iowait()**.

Errors

If the **_iread**() function fails, *errno* may be set to one of the error code values described for the OSF/1 **read(2)** function or one of the following values:

EMIXIO In I/O modes **M_SYNC**, nodes are attempting different operations (reads and writes) to a shared file. In these modes, all nodes must perform the same operation.

EMREQUEST An asynchronous system call has been attempted, but too many requests are already outstanding. Use either **iowait**() or **iodone**() to clear asynchronous read and write requests that are outstanding.

See Also

cread(), **cwrite**(), **iodone**(), **iowait**(), **iseof**(), **iwrite**(), **setiomode**()

OSF/1 Programmer's Reference: **dup(2)**, **open(2)**, **read(2)**

IRECV() IRECV()

irecv(), irecvx(): Posts a receive for a message and returns immediately. (Asynchronous receive)

Synopsis

```
#include <nx.h>

long irecv(
        long typesel,
        char *buf,
        long count );

long irecvx(
        long typesel,
        char *buf,
        long count,
        long nodesel,
        long ptypesel,
        long info[] );
```

Parameters

typesel	Message type(s) to receive. Setting this parameter to `-1` receives a message of any type. Refer to Appendix A of the *Paragon™ OSF/1 C System Calls Reference Manual* for more information about message type selectors.
buf	Pointer to the buffer in which to store the received message. The buffer can be of any valid data type, but should match the data type of the buffer in the corresponding send operation.
count	Length (in bytes) of the *buf* parameter.
nodesel	Node number of the sender. Setting the *nodesel* parameter to `-1` receives a message from any node.
ptypesel	Process type of the sender. Setting the *ptypesel* parameter to `-1` receives a message from any process type.

IRECV() *(cont.)* IRECV() *(cont.)*

info Eight-element array of long integers in which to store message information. The first four elements contain the message's type, length, sending node, and sending process type. The last four elements are reserved for system use. If you do not need this information, you can specify the global array *msginfo*, which is the array used by the **info...()** system calls.

Description

Use the appropriate asynchronous receive function to post a receive for a message and return immediately:

- Use the **irecv**() function to post a receive for a message of a specified type.
- Use the **irecvx**() function to post a receive for a message of a specified type from a specified sender and place information about the message in an array.

The asynchronous receive system calls return a message ID that you can use with the **msgdone**() and **msgwait**() system calls to determine when the receive completes (and the buffer contains valid data).

For the **irecv**() function, you can use the **info...**() system calls to get more information about the message after it is received. For the **irecvx**() function, the same message information is returned in the *info* array. Until the receive completes, neither the **info...**() system calls nor the *info* array contain valid information.

If the message is too long for the buffer, the receive completes with no error returned, and the content of the buffer is undefined. To detect this situation, check the value of the **infocount**() function or the second element of the *info* array.

These are asynchronous system calls. The calling process continues to run while the receive is being done. If your program needs the received message for further processing, it must do one of the following:

- Use the **msgwait**() function to wait until the receive completes.
- Loop until the **msgdone**() function returns `1`, indicating that the receive is complete.
- Use one of the synchronous system calls (for example, **crecv**()) instead.

NOTE

The number of message IDs is limited, and an error occurs when no message IDs are available for a requested asynchronous send or receive. Therefore, your program should release its message IDs as soon as possible by calling **msgcancel()**, **msgdone()**, **msgignore()**, or **msgwait()**.

Return Values

Upon successful completion, the **irecv**() and **irecvx**() functions return a message ID and return control to the calling process. If an error occurs, these functions print an error message to standard error and cause the calling process to terminate. The message ID is for use with the **msgcancel**(), **msgdone**(), **msgignore**(), **msgmerge**(), or **msgwait**() system calls.

Upon successful completion, the **_irecv**() and **_irecvx**() functions return a message ID. Otherwise, these functions return `-1` and set *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a complete list of error codes that occur in the C underscore system calls.

See Also

crecv(), **csend**(), **csendrecv**(), **errno**, **hrecv**(), **hsend**(), **hsendrecv**(), **infocount**(), **infonode**(), **infoptype**(), **infotype**(), **isend**(), **isendrecv**(), **msgcancel**(), **msgdone**(), **msgignore**(), **msgmerge**(), **msgwait**()

ISEND() ISEND()

Sends a message and returns immediately. (Asynchronous send)

Synopsis

```
#include <nx.h>

long isend(
        long type,
        char *buf,
        long count,
        long node,
        long ptype );
```

Parameters

type	Type of the message to send. Refer to Appendix A of the *Paragon™ OSF/1 C System Calls Reference Manual* for information on message types.
buf	Pointer to the buffer containing the message to send. The buffer may be of any valid data type.
count	Number of bytes to send in the *buf* parameter.
node	Node number of the message destination (that is, the receiving node). Setting *node* to `-1` sends the message to all nodes in the application (except the sending node when the *ptype* is the sender's process type).
ptype	Process type of the message destination (that is, the receiving process).

Description

The **isend()** function returns a message ID that you can use with the **msgdone()** and **msgwait()** functions to determine when the send completes. Completion of the send does not mean that the message was received, only that the message was sent and the send buffer (*buf*) can be reused.

In an asynchronous system call, the calling process continues to run while the send is being done. To send a message and block the calling process until the send completes, use an synchronous send call (for example, **csend()**).

Return Value

Upon successful completion, the **isend**() function returns a message ID and returns control to the calling process. If an error occurs, this function displays an error message to standard error and causes the calling process to terminate. The message ID is for use with the **msgcancel**(), **msgdone**(), **msgignore**(), **msgmerge**(), or **msgwait**() system calls.

Upon successful completion, the **_isend**() function returns a message ID. Otherwise, this function returns `-1` and sets *errno* to indicate the error.

NOTE

The number of message IDs is limited, and an error occurs when no message IDs are available for a requested asynchronous send or receive. Therefore, your program should release its message IDs as soon as possible by calling **msgcancel()**, **msgdone()**, **msgignore()**, or **msgwait()**.

Errors

Refer to the **errno** manual page for a complete list of error codes that occur in the C underscore system calls.

See Also

cprobe(), **crecv**(), **csend**(), **csendrecv**(), **errno**, **hrecv**(), **hsend**(), **hsendrecv**(), **iprobe**(), **irecv**(), **isendrecv**(), **msgcancel**(), **msgdone**(), **msgignore**(), **msgmerge**(), **msgwait**()

ISENDRECV() ISENDRECV()

Sends a message, posts a receive for a reply, and returns immediately. (Asynchronous send-receive)

Synopsis

```
#include <nx.h>

long isendrecv(
        long type,
        char *sbuf,
        long scount,
        long node,
        long ptype,
        long typesel,
        char *rbuf,
        long rcount );
```

Parameters

type	Type of the message to send. Refer to Appendix A of the *Paragon™ OSF/1 C System Calls Reference Manual* for more information about message types.
sbuf	Pointer to the buffer containing the message to send. The buffer may be of any valid data type.
scount	Number of bytes to send in the *sbuf* parameter.
node	Node number of the message destination (that is, the receiving node). Setting *node* to `-1` sends the message to all nodes in the application (except the sending node when *ptype* is the sender's process type).
ptype	Process type of the message destination (that is, the receiving process).
typesel	Message type(s) to receive. Setting this parameter to `-1` receives a message of any type. Refer to Appendix A of the *Paragon™ OSF/1 C System Calls Reference Manual* for more information about message type selectors.
rbuf	Pointer to the buffer in which to store the reply. The buffer can be of any valid data type, but should match the data type of the buffer in the corresponding send operation.
rcount	Length (in bytes) of the *rbuf* parameter.

ISENDRECV() *(cont.)* ISENDRECV() *(cont.)*

Description

The **isendrecv**() function sends a message and immediately posts a receive for a reply. The **isendrecv**() function immediately returns a message ID that you can use with **msgdone**() and **msgwait**() to determine when the send-receive completes (that is, the reply is received). When the reply arrives, the calling process receives the message and stores it in the *rbuf* buffer.

If the reply is too long for *rbuf*, the receive completes with no error returned, and the content of the *rbuf* buffer is undefined.

This is an asynchronous system call. The calling process continues to run while the send-receive operation is occurring. If your program needs the received data for further processing, it must do one of the following:

- Use the **msgwait**() function to wait until the receive completes.
- Loop until the **msgdone**() function returns `1`, indicating that the receive is complete.
- Use the **csendrecv**() function (a synchronous system call) instead of the **isendrecv**() function.

Return Values

Upon successful completion, the **isendrecv**() function returns a message ID and returns control to the calling process. If an error occurs, this function displays an error message to standard error and causes the calling process to terminate. The message ID is for use with the **msgcancel**(), **msgdone**(), **msgignore**(), **msgmerge**(), or **msgwait**() system calls.

Upon successful completion, the **_isendrecv**() function returns a message ID. Otherwise, this function returns `-1` and sets *errno* to indicate the error.

NOTE

The number of message IDs is limited, and an error occurs when no message IDs are available for a requested asynchronous send or receive. Therefore, your program should release its message IDs as soon as possible by calling **msgcancel()**, **msgdone()**, **msgignore()**, or **msgwait()**.

Errors

Refer to the **errno** manual page for a complete list of error codes that occur in the C underscore system calls.

See Also

cprobe(), **crecv()**, **csend()**, **csendrecv()**, **errno**, **hrecv()**, **hsend()**, **hsendrecv()**, **iprobe()**, **irecv()**, **isend()**, **isendrecv()**, **msgcancel()**, **msgdone()**, **msgignore()**, **msgmerge()**, **msgwait()**

ISEOF() ISEOF()

Determines whether the file pointer is at end-of-file.

Synopsis

```
#include <nx.h>

long iseof(
        int fildes );
```

Parameters

fildes	A file descriptor representing an open file.

Description

Use the **iseof**() function together with read or write operations to determine whether the file pointer in a file is at the end-of-file.

Return Values

Upon successful completion, the **iseof**() function returns control to the calling process and returns the following values:

0	If file pointer is not at end-of-file.
1	If file pointer is at end-of-file.

Otherwise, the **iseof**() function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_iseof**() function returns the same values as the **iseof**() function. Otherwise, the **_iseof**() function returns -1 and sets *errno* to indicate the error.

Errors

If the **_iodone**() function fails, *errno* may be set to the following error code value:

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

ISEOF() *(cont.)*

ISEOF() *(cont.)*

See Also

cread(), cwrite(), eseek(), iread(), iwrite(), lseek()

OSF/1 Programmer's Reference: **open(2), read(2), write(2)**

ISNAN() ISNAN()

isnan(), **isnand**(), **isnanf**(): Test for floating-point NaN (Not-a-Number).

Synopsis

```
#include <ieeefp.h>

int isnan(
        double dsrc );

int isnand(
        double dsrc );

int isnanf(
        float fsrc );
```

Parameters

dsrc	Any **double** value.
fsrc	Any **float** value.

Description

These functions determine whether or not their argument is an IEEE "Not-a-Number" (NaN). None of these functions ever generates an exception, even if the argument is a NaN.

Return Values

Upon successful completion, the **isnan**(), **isnand**(), and **isnanf**() functions return `1` if the argument is a NaN or `0` if the argument is not a NaN, and these functions return control to the calling process. If an error occurs, these functions print an error message to standard error and cause the calling process to terminate.

Upon successful completion, the **_isnan**(), **_isnand**(), and **_isnanf**() functions return `1` if the argument is a NaNor `0` if the argument is not a NaN. Otherwise, these functions return `-1` when an error occurs and set *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a complete list of error codes that occur in the C underscore system calls.

See Also

errno, **fpgetround**()

IWRITE() IWRITE()

Writes to a file and returns immediately. (Asynchronous write)

Synopsis

```
#include <nx.h>

long iwrite(
        int fildes,
        char *buffer,
        unsigned int nbytes );
```

Parameters

fildes	File descriptor identifying the file to which the data is to be written.
buffer	Pointer to the buffer containing the data to be written.
nbytes	Number of bytes to write.

Description

Other than return values, additional errors, and asynchronous behavior (all discussed in this manual page), the **iwrite()** function is identical to the OSF/1 **write()** function. See **write(2)** in the *OSF/1 Programmer's Reference.*

The **iwrite()** function is an asynchronous version of the **cwrite()** function. A call to the **iwrite()** function returns immediately to the calling process. The calling process continues to run while the write is being done. If the calling process needs the write buffer for further processing, it must do one of the following:

- Use **cwrite()** (a synchronous system call) instead of **iwrite()**.
- Use **iowait()** to wait until the write completes.
- Loop until **iodone()** returns a `1`, indicating that the write is complete.

The **iwrite()** function modifies the file pointer immediately, so **lseek()**, **iseof()**, or a read or write call can be used immediately without waiting for the write to finish. To determine whether the write operation moved the file pointer to the end of the file, use the **iseof()** function.

IWRITE() *(cont.)* IWRITE() *(cont.)*

Return Values

Upon successful completion, the **iwrite()** function returns control to the calling process and a non-negative I/O ID for use in **iodone()** and **iowait()** functions. Otherwise, the **iwrite()** function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_iwrite()** function returns a non-negative I/O ID for use in **iodone()** and **iowait()** calls. Otherwise, the **_iwrite()** function returns -1 and sets *errno* to indicate the error.

NOTE

The number of I/O IDs is limited, and an error occurs when no I/O IDs are available for a requested asynchronous read or write. Therefore, your program should release the I/O ID as soon as possible by calling **iodone()** or **iowait()**.

Errors

If the **_iwrite()** function fails, *errno* may be set to one of the error code values described in the OSF/1 **write(2)** function or one of the following values:

EMIXIO In I/O modes **M_SYNC** or **M_GLOBAL**, nodes are attempting different operations (reads and writes) to a shared file. In these modes, all nodes must perform the same operation.

EMREQUEST An asynchronous system call has been attempted, but too many requests are already outstanding. Use either **iowait()** or **iodone()** to clear asynchronous read and write requests that are outstanding.

See Also

cread(), **cwrite()**, **iodone()**, **iowait()**, **iread()**, **iseof()**, **setiomode()**

OSF/1 Programmer's Reference: **dup(2)**, **open(2)**, **write(2)**

LED() LED()

Turns the node board's green LED on or off.

Synopsis

```
#include <nx.h>

void led(
        long state );
```

Parameters

state Specifies the state of the node board's green LED:

`1`	Turns on the LED.
`0`	Turns off the LED.

Other values are not defined.

Description

The Intel supercomputer has a number of light-emitting diodes (LEDs) on its front panel that indicate the processor and message-passing status of all the nodes in the system. The **led**() function controls the node board's green LED allowing you to turn it on and off.

Return Values

Upon successful completion, the **led**() function returns control to the calling process; no values are returned. Otherwise, this function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_led**() function returns `0` (zero). Otherwise, this function returns `-1` and sets *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

LED() *(cont.)*

LED() *(cont.)*

See Also

errno

LSIZE() LSIZE()

Increases the size of a file.

Synopsis

```
#include <nx.h>

long lsize(
        int fildes,
        off_t offset,
        int whence );
```

Parameters

fildes A file descriptor representing a regular file opened for writing.

offset The value, in bytes, to be used together with the *whence* parameter to increase the file size.

whence Specifies how *offset* affects the file size. Values for the *whence* parameter are as follows (defined in *nx.h*):

SIZE_SET Sets the file size to the greater of the current size or *offset*.

SIZE_CUR Sets the file size to the greater of the current size or the current location of the file pointer plus *offset*.

SIZE_END Sets the file size to the greater of the current size or the current size plus *offset*.

Description

The **lsize()** function allocates sufficient file space before starting performance-sensitive applications or storage operations. This increases throughput for I/O operations on a file, because the I/O system does not have to allocate data blocks for every write that extends the file size.

This function cannot decrease the size of a file.

The **lsize()** function has no effect on FIFO special files or directories, nor does it affect the position of the file pointer. The contents of file space allocated by the **lsize()** function is undefined.

LSIZE() *(cont.)* LSIZE() *(cont.)*

If the file has enforced file locking enabled and there are file locks on the file, the **lsize**() function fails.

Note

If the requested size is greater than the available disk space, **lsize()** allocates the available disk space and returns the actual new size.

Return Values

Upon successful completion, the **lsize**() function returns the new size of the file, in bytes. Otherwise, the **lsize**() function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_lsize**() function returns the same value as the **lsize**() function. Otherwise, the **_lsize**() function returns `-1` and sets *errno* to indicate the error.

Errors

If the **_lsize**() function fails, *errno* may be set to one of the following error code values:

EGAIN The file has enforced mode file locking enabled and there are file locks on the file.

EACCES Write access permission to the file was denied.

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

EFBIG The file size specified by the *whence* and *offset* parameters exceeds the maximum file size.

EFSNOTSUPP The *fildes* parameter refers to a file that resides in a file system that does not support this operation.

EINVAL The file is not a regular file.

ENOSPC No space left on device.

EROFS The file resides on a read-only file system.

LSIZE() *(cont.)*

LSIZE() *(cont.)*

See Also

eseek(), esize()

OSF/1 Programmer's Reference: **fcntl(2), lseek(2), open(2)**

MASKTRAP() MASKTRAP()

Enables or disables send and receive traps.

Synopsis

```
#include <nx.h>

long masktrap(
        long state );
```

Parameters

state The state of send-receive traps:

`0`	Enables (allows) send and receive traps.
`1`	Disables (blocks) send and receive traps.

Other values are not defined.

Description

The **masktrap**() function enables and disables send and receive traps. Use this function to protect critical code from being interrupted by the execution of the handler procedure invoked when one of the handler send or receive system calls (for example, **hrecv**(), **hsend**(), or **hsendrecv**()) completes. If a send or receive operation completes after calling the **masktrap**() function with a *state* value of `1` to disable traps, its interrupt is delayed until the **masktrap**() function is called with a *state* value of `0` to enable traps again.

Return Values

Upon successful completion, the **masktrap**() function returns the previous value of *state* and returns control to the calling process. Otherwise, this function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_masktrap**() function returns the previous value of *state*. Otherwise, this function returns `-1` and sets *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

errno, **hrecv()**, **hsend()**, **hsendrecv()**

MSGCANCEL() MSGCANCEL()

Cancels an asynchronous send or receive operation.

Synopsis

```
#include <nx.h>

void msgcancel(
        long mid );
```

Parameters

mid	The message ID returned by one of the asynchronous send or receive system calls (for example, **isend**(), **irecv**(), or **isendrecv**()) or by the **msgmerge**() system call.

Description

The **msgcancel**() function cancels an asynchronous send or receive operation. When **msgcancel**() returns, you do not know whether the send or receive operation completed, but you do know the following:

- The asynchronous operation is no longer active.
- The message buffer may be reused.
- The message ID is released.

NOTE

The number of message IDs is limited, and an error occurs when no message IDs are available for a requested asynchronous send or receive. Therefore, your program should release its message IDs as soon as possible by calling **msgcancel()**, **msgdone()**, **msgignore()**, or **msgwait()**.

Return Values

Upon successful completion, the **msgcancel()** function returns control to the calling process; no values are returned. Otherwise, this function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_msgcancel()** function returns `0` (zero). Otherwise, this function returns `-1` and sets *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

errno, isend(), irecv(), isendrecv(), msgdone(), msgignore(), msgmerge(), msgwait()

MSGDONE() MSGDONE()

Determines whether an asynchronous send or receive operation is complete.

Synopsis

```
#include <nx.h>

long msgdone(
        long mid );
```

Parameters

mid	Message ID returned by one of the asynchronous send or receive system calls (for example, **isend()**, **irecv()**, or **isendrecv()**) or by the **msgmerge()** system call.

Description

If the **msgdone()** function returns 1, it means the asynchronous send or receive operation identified by *mid* is complete, and indicates the following:

- The buffer contains valid data (if *mid* identifies a receive operation), or the buffer is available for reuse (if *mid* identifies a send operation).
- The *info* array (used by the extended receive system calls) contains valid information.
- The **info...()** system calls return valid information.
- The message ID number that identifies the asynchronous send or receive *(mid)* is released for use in a future asynchronous send or receive.

NOTE

The number of message IDs is limited, and an error occurs when no message IDs are available for a requested asynchronous send or receive. Therefore, your program should release its message IDs as soon as possible by calling **msgcancel()**, **msgdone()**, **msgignore()**, or **msgwait()**.

MSGDONE() *(cont.)* MSGDONE() *(cont.)*

Return Values

Upon successful completion, the **msgdone()** function returns the following values and returns control to the calling process:

`0` If the send or receive is not yet complete.

`1` If the send or receive is complete.

Otherwise, this function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_msgdone()** function returns the following:

`0` If the send or receive is not yet complete.

`1` If the send or receive is complete.

Otherwise, this function returns `-1` and sets *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

errno, infocount(), infonode(), infoptype(), infotype(), irecv(), isend(), isendrecv(), msgcancel(), msgignore(), msgmerge(), msgwait()

MSGIGNORE() MSGIGNORE()

Releases a message ID as soon as its asynchronous send or receive operation completes.

Synopsis

```
#include <nx.h>

void msgignore(
        long mid );
```

Parameters

mid	The message ID returned by one of the asynchronous send or receive system calls (for example, **isend()**, **irecv()**, or **isendrecv()**) or by the **msgmerge()** system call.

Description

The **msgignore()** function releases a message ID as soon as its asynchronous send or receive operation completes. This is a non-blocking system call.

NOTE

The number of message IDs is limited, and an error occurs when no message IDs are available for a requested asynchronous send or receive. Therefore, your program should release its message IDs as soon as possible by calling **msgcancel()**, **msgdone()**, **msgignore()**, or **msgwait()**.

Note the following:

- An application must have some alternate means to determine when it can reuse a send or receive buffer.
- Do not use **msgignore()** as a substitute for **msgwait()**.
- The *mid* cannot be reused by **msgdone()** or **msgwait()**.

Return Values

Upon successful completion, the **msgignore()** function returns control to the calling process; no values are returned. Otherwise, this function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_msgignore()** function returns `0` (zero). Otherwise, this function returns `-1` and sets *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

errno, **irecv()**, **isend()**, **msgcancel()**, **msgdone()**, **msgmerge()**, **msgwait()**

MSGMERGE() MSGMERGE()

Groups two message IDs together so they can be treated as one.

Synopsis

```
#include <nx.h>

long msgmerge(
        long mid1,
        long mid2 );
```

Parameters

mid1, *mid2*	Message IDs returned by asynchronous send or receive system calls (for example, **isend**(), **irecv**(), or **isendrecv**()) or by the **msgmerge**() system call.

Description

The **msgmerge**() function groups *mid2* with *mid1* and returns a message ID to use for both. After calling **msgmerge**(), the original message IDs (*mid1* and *mid2*) become invalid (although they are not released until the new message ID is released). The operation associated with the new message ID (**msgdone**() or **msgwait**()) does not complete until *both* of the asynchronous send or receive operations associated with the original message IDs complete.

Normally, **msgmerge**() returns *mid1*, and only *mid2* becomes invalid. As a special case, one *mid* can be -1, in which case the other *mid* is returned with no other action.

Do not use the **info...**() system calls after a call to the **msgmerge**() function; the information returned is unpredictable.

Return Value

Upon successful completion, the **msgmerge()** function returns a message ID and returns control to the calling process. Otherwise, this function displays an error message to standard error and causes the calling process to terminate. The returned message ID is for use in **msgcancel()**, **msgdone()**, **msgignore()**, **msgmerge()**, or **msgwait()** system calls.

Upon successful completion, the **_msgmerge()** function returns a message ID. Otherwise, this function returns `-1` and sets *errno* to indicate the error.

NOTE

The number of message IDs is limited, and an error occurs when no message IDs are available for a requested asynchronous send or receive. Therefore, your program should release its message IDs as soon as possible by calling **msgcancel()**, **msgdone()**, **msgignore()**, or **msgwait()**.

Errors

Refer to the **errno** manual page for a complete list of error codes that occur in the C underscore system calls.

See Also

errno, **irecv()**, **isend()**, **isendrecv()**, **msgcancel()**, **msgdone()**, **msgignore()**, **msgwait()**

MSGWAIT() MSGWAIT()

Waits (blocks) until an asynchronous send or receive operation completes.

Synopsis

```
#include <nx.h>

void msgwait(
        long mid );
```

Parameters

mid	The message ID returned by one of the asynchronous send or receive system calls (for example, **isend**(), **irecv**(), or **isendrecv**()) or by the **msgmerge**() system call.

Description

The **msgwait**() function causes a node process to wait until an asynchronous send or receive operation (for example, **isend**() or **irecv**()) completes. When the **msgwait**() function returns:

- The buffer contains valid data (if *mid* identifies a receive operation), or the buffer is available for reuse (if *mid* identifies a send operation).
- The *info* array (used by the extended receive system calls) contains valid information.
- The **info...**() system calls return valid information.
- The message ID that identifies the asynchronous send or receive *(mid)* is released for use in a future asynchronous send or receive.

NOTE

The number of message IDs is limited, and an error occurs when no message IDs are available for a requested asynchronous send or receive. Therefore, your program should release its message IDs as soon as possible by calling **msgcancel()**, **msgdone()**, **msgignore()**, or **msgwait()**.

Return Values

Upon successful completion, the **msgwait**() function returns control to the calling process; no values are returned. Otherwise, this function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_msgwait**() function returns `0` (zero). Otherwise, this function returns `-1` and sets *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

errno, **infocount**(), **infonode**(), **infoptype**(), **infotype**(), **irecv**(), **isend**(), **isendrecv**(), **msgcancel**(), **msgdone**(), **msgignore**(), **msgmerge**()

MYHOST() MYHOST()

Gets the node number of the controlling process.

Synopsis

```
#include <nx.h>

long myhost(void);
```

Description

The **myhost()** function returns the node number of the caller's controlling process (the host process) for use in send and receive operations. For controlling processes, **myhost()** returns the same number as **mynode()**, which is the node number of the calling process.

Return Values

Upon successful completion, the **myhost()** function returns the node number of the controlling process and returns control to the calling process. Otherwise, this function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_myhost()** function returns the node number of the controlling process. Otherwise, this function returns `-1` and sets *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

csendrecv(), **errno**, **hsend()**, **hsendrecv()**, **isendrecv()**, **mynode()**, **myptype()**, **numnodes()**, **nx_loadve()**, **nx_nfork()**

MYNODE() MYNODE()

Gets the node number of the calling process.

Synopsis

```
#include <nx.h>

long mynode(void);
```

Description

The **mynode**() function returns the node number of the calling process (an integer between 0 and **numnodes**()).

Return Value

Upon successful completion, the **mynode**() function returns the node number of the calling process and returns control to the calling process. Otherwise, this function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_mynode**() function returns `0` (zero). Otherwise, this function returns `-1` and sets *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

errno, **myhost**(), **myptype**(), **numnodes**(), **nx_loadve**(), **nx_nfork**()

MYPTYPE() MYPTYPE()

Gets the process type of the calling process.

Synopsis

```
#include <nx.h>

long myptype(void);
```

Description

The **myptype**() function returns the process type of the calling process.

Return Values

Upon successful completion, the **myptype**() function returns the process type (*ptype*) of the calling process and returns control to the calling process. Otherwise, this function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_myptype**() function returns the process type (*ptype*) of the calling process. Otherwise, this function returns `-1` and sets *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

csend(), **csendrecv**(), **errno**, **hsend**(), **hsendrecv**(), **isend**(), **isendrecv**(), **myhost**(), **mynode**(), **numnodes**(), **nx_loadve**(), **nx_nfork**(), **setptype**()

NUMNODES() NUMNODES()

Gets the number of nodes in an application.

Synopsis

```
#include <nx.h>

long numnodes(void);
```

Description

The **numnodes**() function returns the number of nodes allocated to the application.

Return Values

Upon successful completion, the **numnodes**() function returns the number of nodes in an application and returns control to the calling process. Otherwise, this function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_numnodes**() function returns the number of nodes in an application. Otherwise, this function returns `-1` and sets *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

errno, **myhost**(), **mynode**(), **nx_initve**(), **nx_load**()

NX_CHPART() NX_CHPART()

nx_chpart_epl(), nx_chpart_mod(), nx_chpart_name(), nx_chpart_owner(), nx_chpart_rq(): Changes a partition's characteristics.

Synopsis

```
#include <nx.h>

long nx_chpart_epl(
        char *partition,
        long priority );

long nx_chpart_mod(
        char *partition,
        long mode );

long nx_chpart_name(
        char *partition,
        char *name );

long nx_chpart_owner(
        char *partition,
        long owner,
        long group );

long nx_chpart_rq(
        char *partition,
        long rollin_quantum );
```

Parameters

partition Pointer to a relative or absolute pathname of the partition for which you are changing the characteristics. The partition must exist and must give write permission to the calling process (except for the **nx_chpart_owner()** function).

priority New effective priority limit for the partition expressed as an integer with a range from 0 (lowest priority) to 10 (highest priority) inclusive.

NX_CHPART() *(cont.)* NX_CHPART() *(cont.)*

mode New protection modes for the partition expressed as an octal number. See the **chmod** command in the *OSF/1 Command Reference* for more information on specifying protection modes.

name New name for a partition.The *name* parameter must be a simple name (without dots).

owner New owner for the partition expressed as a numeric user ID (UID). If the *owner* parameter is `-1`, the owner name is not changed.

See the *OSF/1 Programmer's Reference* for information about using the **getpwnam()** function to convert a user name to a numeric user ID.

group New group for the partition expressed as a numeric group ID (GID). If the *group* parameter is -1, the group name is unchanged. See the *OSF/1 Programmer's Reference* for information about using the **getgrnam()** function to convert a group name to a numeric group ID.

rollin_quantum New rollin quantum for the partition expressed as an integer number of milliseconds, or `0` to specify infinite rollin quantum. The specified value must not be greater than 86,400,000 milliseconds (24 hours). If you specify a value that is not a multiple of 100, the value is silently rounded up to the next multiple of 100.

Description

The **nx_chpart...()** functions change specific characteristics of a partition. Each of these functions specifically changes a partition characteristic as follows:

nx_chpart_epl()
Changes the partition's effective priority limit.

nx_chpart_mod()
Changes the partition's permission modes.

nx_chpart_name()
Changes the partition's name.

nx_chpart_owner()
Changes the partition's owner and group.

nx_chpart_rq() Changes the partition's rollin quantum.

NX_CHPART() *(cont.)*

NX_CHPART() *(cont.)*

You initially set a partition's characteristics when you create the partition with the **mkpart** command or the **nx_mkpart...()** functions. Using the **mkpart** command, you can specifically set the partition's characteristics or use the default characteristics, which are the parent partition's characteristics. Using the **nx_mkpart...()** functions, the partition receives the characteristics of the parent partition. After you create a partition, you are the owner of the partition. You can use the **nx_chpart...()** functions or the **chpart** command to change the partition's characteristics.

The effective priority limit is the upper priority limit on a partition. The system uses the effective priority limit for gang scheduling in partitions. See the *Paragon™ OSF/1 User's Guide* for more information about effective priority limits and gang scheduling. The **nx_chpart_epl()** function changes the effective priority limit for a partition. The effective priority ranges from 0 to 10. This limit does not affect the priority of applications or partitions within a partition.

The **nx_chpart_name()** function changes the partition's name only. You cannot use this function to change the partition's parent partition or the partition's relationship in a partition hierarchy.

Each partition has an owner, a group, and protection modes that determine who can perform what operations on a partition. When you create a partition, you become the partition's owner and the partition's group is set to your current group. The **nx_chpart_owner()** function changes the owner and group of a partition. Use the OSF/1 **getpwnam()** function to convert an owner name to a user ID. Use the OSF/1 **getgrnam()** function to convert a group name to a numeric group ID. See the *OSF/1 Programmer's Reference* for more information about these functions.

A partition's protection mode consists of three groups of permission bits that indicate the read, write and execute permissions for the owner, group, and other users of the partition. A partition's protection mode is initially set when the partition is created. The **nx_chpart_mod()** function changes the protection mode for a partition. Set the *mode* parameter to the three-digit octal value that represents the protection mode you want for the partition. See the **chmod** command in the *OSF/1 Command Reference* for more information on specifying protection modes.

Return Values

0	Partition's characteristic was successfully changed.
-1	Error; *errno* is set.

Errors

When -1 is returned by this function, *errno* is set to one of the following values:

EPACCES The application has insufficient access permission on a partition.

EPALLOCERR
An internal error occurred in the node allocation server.

EPBADNODE The specified node is a bad node or is not present in the partition.

EPINVALPART
The specified partition (or its parent) does not exist.

EPLOCK The specified partition is currently being updated and is locked by someone else.

EPPARTEXIST
The specified partition already exists.

See Also

chpart, **lspart**, **mkpart**, **nx_mkpart()**, **nx_rmpart()**, **pspart**, **rmpart**

OSF/1 Command Reference: **chgrp(1)**, **chmod(1)**, **chown(1)**

OSF/1 Programmer's Reference: **getgrnam(3)**, **getpwnam(3)**

NX_INITVE() NX_INITVE()

Initializes a parallel application.

Synopsis

```
#include <nx.h>

long nx_initve(
        char *partition,
        long size,
        char *account,
        long *argc,
        char *argv[]);
```

Parameters

partition	Name of the partition in which to run the application in, or a null string (" ") to use the partition specified by the **-pn** switch on the command line. You can use a relative or an absolute partition pathname; the specified partition must exist and must give execute permission to the calling process. If you specify the null string for *partition*, and *argc* is 0 (zero) or the **-pn** switch is not used, the application runs in the partition specified by the environment variable *NX_DFLT_PART*. If *NX_DFLT_PART* is not set, the default is the *.compute* partition.
size	Number of nodes to run the application on, or 0 (zero) to use the size specified by the **-sz** switch on the command line. If *argc* is 0 (zero) or the **-sz** switch is not used, defaults to all nodes in the partition.
account	Reserved for future use. Set this parameter to NULL.
argc	Pointer to an integer that is the number of arguments on the command line (including the application name). If the value of the integer is 0 (zero), the command line is ignored. When **nx_initve()** returns, *argc* indicates the number of remaining command line arguments after all the recognized arguments are removed from *argv*.
argv	Array of character pointers to null-terminated strings containing the application's command line arguments. All recognized arguments are removed from *argv*.

NX_INITVE() *(cont.)* NX_INITVE() *(cont.)*

Description

The **nx_initve**() function initializes an application on a set of nodes in a specified partition. Use this call as follows:

- Call **nx_initve**() before any other Paragon™ OSF/1 system calls.
- Call **nx_initve**() only once.
- Do not use the **-nx** option to link a program that calls **nx_initve**(). Use the **-lnx** option.

The **nx_initve**() function just initializes a program. Use the **nx_loadve**(), **nx_load**(), or **nx_nfork**() calls to start a program's processes.

The **nx_initve**() function recognizes the following command line switches for an application: **-gth**, **-mbf**, **-mea**, **-mex**, **-on**, **-pkt**, **-plk**, **-pn**, **-pri**, **-pt**, **-sct**, **-sth**, and **-sz**. See the ***application*** manual page for a description of the recognized switches for applications. When a switch is recognized, the appropriate application characteristic is set, the switch and any associated argument are removed from *argv*, and *argc* is decremented appropriately. The remaining switches and arguments are moved to the beginning of *argv*.

The application's scheduling priority is specified by the **-pri** argument in *argv*. If the **-pri** switch is not specified or the *argc* parameter is 0 (zero), then the scheduling priority is set to 5.

When calling the **nx_initve**() function, the calling process becomes the controlling process of the application. If not already the process group leader, the **nx_initve**() function disassociates the calling process from its current process group, creates a new process group, and makes the calling process the process group leader of the new process group.

The **nx_initve**() function does not set the calling process's *ptype*.

Return Values

> 0	Number of nodes on which the application was created.
-1	An error occurred and *errno* is set.

NX_INITVE() *(cont.)* NX_INITVE() *(cont.)*

Errors

When `-1` is returned by this function, *errno* is set to one of the following values:

EAEXIST An application has already been established for the process group.

EPALLOCERR An internal error occurred in the node allocation server.

EPACCES The application has insufficient access rights to a partition for this operation.

EPXRS The request exceeds the partition resources.

See Also

application, **nx_load()**, **nx_nfork()**

NX_LOAD() NX_LOAD()

nx_load(), **nx_loadve**(): Loads and starts an executable image.

Synopsis

```
#include <nx.h>

long nx_load(
        long node_list[],
        long numnodes,
        long ptype,
        long pid_list[],
        char *pathname );

long nx_loadve(
        long node_list[],
        long numnodes,
        long ptype,
        long pid_list[],
        char *pathname,
        char *argv[],
        char *envp[] );
```

Parameters

node_list	Array of node numbers on which to load and start the executable image.
numnodes	Number of node numbers in the *node_list*. If *numnodes* is set to `-1`, the application is loaded onto all the application's nodes (the *node_list* parameter is ignored).
ptype	Process type of the new process(es).
pathname	Pathname of the executable image to load and start.

NX_LOAD() *(cont.)*

NX_LOAD() *(cont.)*

pid_list Array of OSF/1 process IDs (PID) of the new processes. Each element of the *pid_list* array identifies the process ID of the node identified by the corresponding element of *node_list*. An entry of 0 (zero) indicates that the process on the corresponding node was not started successfully. The *pid_list* array must be the size of the number of nodes used in the application.

If the *num_nodes* parameter equals -1, the first element of the *pid_list* array equals the PID of node 0, the second element of the *pid_list* array equals the PID of node 1, and so on for all the nodes in the system.

argv The argument vector pointer to pass to the executable image's new processes (corresponds to the *argv* parameter of the OSF/1 **execve(2)** system call).

envp The environment vector pointer to pass to the executable image's new processes (corresponds to the *env* parameter of the OSF/1 **execve(2)** system call).

Description

The **nx_load**() and **nx_loadve**() functions load and start an executable image on the nodes specified by the *node_list* parameter. The **nx_loadve**() function is just like the **nx_load**() function except it lets you specify the argument list and environment variables for the new process. These calls can only be made after the calling process makes an initial **nx_initve**() call.

The **nx_load**() and **nx_loadve**() functions return immediately to the calling process. Use **nx_waitall**() to wait for processes created by **nx_load**() and **nx_loadve**().

Return Values

> 0 Number of nodes on which the executable image was loaded and started successfully.

-1 Error; *errno* is set.

NOTE

It is possible that loading and starting the executable image could fail on more than one node, and that each failure could be for a different reason. In such a case, the value of *errno* reflects only one of the failures, and it is not possible to determine which one.

NX_LOAD() *(cont.)* NX_LOAD() *(cont.)*

Errors

When -1 is returned by this function, *errno* is set to one of the following values:

EPALLOCERR An internal error occurred in the node allocation server.

EPBADNODE The specified node is a bad node.

See Also

nx_initve(), nx_nfork(), nx_waitall(), setptype()

OSF/1 Programmer's Reference: **execve(2)**

NX_MKPART() NX_MKPART()

nx_mkpart(), **nx_mkpart_rect()**, **nx_mkpart_map()**: Creates a new partition.

Synopsis

```
#include <nx.h>

long nx_mkpart(
        char *partition,
        long size,
        long type );

long nx_mkpart_rect(
        char *partition,
        long rows,
        long cols,
        long type );

long nx_mkpart_map(
        char *partition,
        long numnodes,
        long node_list[],
        long type );
```

Parameters

partition	New partition's relative or absolute pathname. The new partition must not exist. The parent partition of the new partition must exist and must give the calling process write permission.
size	Number of nodes for the new partition, or `-1` to specify all nodes of the parent partition. If you specify a size smaller than the number of nodes in the parent partition, the system selects the nodes that make up the new partition and the nodes are not necessarily contiguous.
type	New partition's scheduling type: **NX_STD** specifies standard scheduling and **NX_GANG** specifies gang scheduling. The scheduling type names are specified in the *nx.h* include file. See the *Paragon™ OSF/1 User's Guide* for more information about partitions and scheduling.
rows	Number of rows in the new partition.

NX_MKPART() *(cont.)* NX_MKPART() *(cont.)*

cols	Number of columns in the new partition.
numnodes	Number nodes in the parent partition available to the new partition.
node_list	Array of node numbers that list the nodes in the parent partition available to the new partition.

Description

The **nx_mkpart**(), **nx_mkpart_rect**(), or **nx_mkpart_map**() functions create partitions for your application programs. The **nx_mkpart**() function creates a partition with a specified number of nodes. The system selects the shape of the partition and the nodes that make up the partition. The nodes are not necessarily contiguous.

The **nx_mkpart_rect**() function creates a partition with a rectangular shape and a specified number of rows and columns. The system allocates the rectangular partition where it can in the parent partition.

The **nx_mkpart_map**() function creates a partition with a specified list of nodes. You pass the *numnodes* and *nodelist* parameters to specify the number of nodes and the list of nodes to use for the new partition. The node numbers listed in the *nodelist* must exist and be available in the parent partition The system allocates the nodes for the new partition from the *nodelist* only.

When you create a partition with the **nx_mkpart...**() functions, the new partition gets default characteristics. The partition's owner and group are set to the owner and group of the calling process. All other characteristics including the effective priority limit, protection mode, and rollin quantum are set to the same values as the parent partition. If you want to change a partition's characteristics, use the **nx_chpart...**() functions.

Return Values

> 0	Number of nodes allocated for the partition.
-1	Error; *errno* is set.

NX_MKPART() *(cont.)* NX_MKPART() *(cont.)*

Errors

When `-1` is returned by this function, *errno* is set to one of the following values:

EPACCES The application has insufficient access permission on a partition.

EPALLOCERR
An internal error occurred in the node allocation server.

EPBADNODE The specified node is a bad node or is not present in the partition.

EPBXRS Partition request contains bad or missing nodes.

EPINVALPART
The specified partition (or its parent) does not exist.

EPLOCK Partition is currently in use or being updated.

EPPARTEXIST
The specified partition already exists.

EPXRS Request exceeds the partition's resources.

See Also

chpart, lspart, mkpart, nx_chpart(), nx_rmpart(), pspart, rmpart

NX_NFORK() NX_NFORK()

Forks the calling process and creates an application's processes.

Synopsis

```
#include <nx.h>

long nx_nfork(
        long node_list[],
        long numnodes,
        long ptype,
        long pid_list[] );
```

Parameters

node_list Array of node numbers on which to fork the calling process.

numnodes Length of the *node_list* array (that is, the number of nodes on which to fork the calling process). If you set the *numnodes* parameter to -1, the **nx_nfork**() uses all the nodes of the application and ignores the *node_list* parameter.

ptype Process type of the new process(es).

pid_list Array in which **nx_nfork**() records the OSF/1 process IDs of the new processes. Each element of the *pid_list* array contains the OSF/1 process ID of the process that was forked on the node identified by the corresponding element of the *node_list* array. An entry of 0 (zero) indicates that the process on the corresponding node was not forked successfully.

If the *numnodes* parameter equals -1, the first element of the *pid_list* array equals the PID of node 0, the second element of the *pid_list* array equals the PID of node 1, and so on for all the nodes in the system.

Description

The **nx_nfork**() function forks the calling process onto the nodes specified by the *node_list* parameter. The fork operation copies the calling process onto a specified set of nodes with a specified process type. It creates one *child process* for each specified node. The **nx_nfork**() function is similar to the OSF/1 **fork**() call, except that it can fork processes onto multiple nodes and specifies a process type for the child processes. This call can only be made after an initial **nx_initve**() call.

NX_NFORK() *(cont.)* NX_NFORK() *(cont.)*

Return Values

If the fork succeeds:

- The parent process receives a value that indicates the number of child processes that were created (that is, the number of nodes on which the process was forked).
- Each child process receives the value 0 (zero).

If the fork fails:

- The calling process receives the value -1.
- Each successfully created child process receives the value 0 (zero).

NOTE

It is possible that the fork could fail on more than one node, and that each failure could be for a different reason. In such a case, the value of *errno* reflects only one of the failures, and it is not possible to determine which one.

Errors

When `-1` is returned by this function, *errno* is set to one of the following values:

EPALLOCERR An internal error occurred in the node allocation server.

EPBADNODE The specified node is a bad node.

See Also

nx_initve(), nx_load(), setptype()

OSF/1 Programmer's Reference: **fork(2)**

NX_PERROR() NX_PERROR()

Print an error message corresponding to the current value of *errno*.

Synopsis

```
#include <nx.h>
#include <errno.h>

void nx_perror(
        char *string );
```

Parameters

string	String that contains the name of the program or function that caused the error.

Description

Other than additional errors and the error message format, **nx_perror()** is identical to the OSF/1 **perror()** call. See **perror(2)** in the *OSF/1 Programmer's Reference*.

There is a standard error message for each value of *errno*, which you can print out by calling **nx_perror()**. **nx_perror()** prints its argument (any string), the current node number and process type, and the error message associated with the current value of *errno* to the standard error output in the following format:

```
(node n, ptype p) string: error_message
```

The include file *errno.h* declares *errno* and defines constants for the possible *errno* values.

Errors

Refer to the **errno** manual page for a complete list of error codes that occur in the C underscore system calls.

See Also

errno

OSF/1 Programmer's Reference: **perror(2)**

NX_PRI() NX_PRI()

Sets the priority of an application.

Synopsis

```
#include <nx.h>

long nx_pri(
        long pgroup,
        long priority );
```

Parameters

pgroup	Process group ID for the application, or 0 (zero) to specify the application of the calling process. If the specified process group ID is not a process group ID of the calling process, the calling process's user ID must either be root or the same user ID as the specified application.
priority	New priority for the application, an integer from 0 (lowest priority) to 10 (highest priority) inclusive.

Description

An application runs in a partition with a priority. The priority determines how and when the application is scheduled to run in the partition. The **nx_pri()** function sets an application's priority. An application's priority can range from 0 (low priority) to 10 (high priority), inclusive; an application with the higher priority takes scheduling precedence over applications with lower priorities. See the *Paragon™ OSF/1 User's Guide* for more information on scheduling and an application's priority.

If you do not call **nx_pri()** and you do not use the **-pri** switch with your application, the default priority is 5.

Return Values

> 0	No errors; priority successfully set.
-1	Error; *errno* is set.

NX_PRI() *(cont.)* NX_PRI() *(cont.)*

Errors

When `-1` is returned by this function, *errno* is set to one of the following values:

EANOEXIST
The specified process group does not exist.

EPALLOCERR
An internal error occurred in the node allocation server.

EPINVALPRI
The specified priority is out of the range of priority values.

See Also

nx_chpart(), nx_initve(), nx_nfork(), nx_load()

Paragon™ OSF/1 User's Guide

NX_RMPART() NX_RMPART()

Removes a partition.

Synopsis

```
#include <nx.h>

long nx_rmpart(
        char *partition,
        long force,
        long recursive );
```

Parameters

partition Relative or absolute pathname of the partition to be removed. The parent partition must give write permission to the calling process.

force Removes partitions that contain running applications. If the value is 0 (zero), the partition will not be removed if any applications are running in the partition. Any other value specifies removing the partition even if applications are running in the partition.

recursive Recursively remove the partition. A value of 0 (zero) specifies that the partition will not be removed if the partition has any subpartitions.

A non-zero value specifies that the partition and all its subpartitions will be removed recursively. There cannot be any applications running in the partition or any of its subpartitions. If applications are running in the partition or any of its subpartitions, the **nx_rmpart()** function does not remove the partition or any of its subpartitions.

The *force* parameter set to a positive integer and used with the *recursive* parameter allows a partitions and subpartitions to be removed if they have applications running in them.

NX_RMPART() *(cont.)* NX_RMPART() *(cont.)*

Description

The **nx_rmpart**() function removes from the system a partition, its subpartitions, and applications running in the partition or its subpartitions. A calling process must have write permission on the parent partition to remove the partition.

The *force* parameter specifies whether to remove the partition if it contains applications. A 0 (zero) value specifies not to remove a partition if it contains applications. Any other value forces the partition to be removed. This is a safety mechanism so you do not accidently destroy an application or subpartition.

The *recursive* parameter specifies whether to remove the partition and all its subpartitions. A 0 (zero) value specifies not to remove a partition if it contains subpartitions. Any other value removes the partition and all its subpartitions.

If you provide non-zero values for both the *force* and *recursive* parameters, **nx_rmpart**() removes the partition and all its subpartitions, even if applications are running in the partition or its subpartitions.

Return Values

> 0	Partition was successfully removed.
-1	Error; *errno* is set.

Errors

When `-1` is returned by this function, *errno* is set to one of the following values:

EPACCESS Insufficient access permission for this operation on a permission.

EPALLOCERR
An internal error occurred in the node allocation server.

EPINVALPART
The specified partition does not exist.

EPLOCK The specified partition is currently being updated and is locked by someone else.

EPNOTEMPTY The specified partition contains one or more subpartitions or running applications.

NX_RMPART() *(cont.)* NX_RMPART() *(cont.)*

See Also

chpart, lspart, mkpart, nx_chpart(), nx_mkpart(), pspart, rmpart

NX_WAITALL() NX_WAITALL()

Waits for all the child processes of a calling process to stop or terminate

Synopsis

```
#include <nx.h>

long nx_waitall(void);
```

Description

The **nx_waitall()** function takes no parameters, waits for all the child processes of a calling process to stop or terminate, and returns `0` (zero) for successful termination of child processes or `-1` for unsuccessful termination of child processes. Otherwise, the **nx_waitall()** function is identical to the OSF/1 **wait()** function. See **wait(2)** in the *OSF/1 Programmer's Reference.*

The **nx_waitall()** function suspends the application's calling process until all the application's child process stop or terminate. An application can start child process with the **nx_nfork()**, **nx_load()**, or **nx_loadve()** functions.

Return Values

`0`	All the application's processes terminated successfully
`-1`	One or more of the application's processes terminated with an error

Errors

If the **nx_waitall()** function fails, *errno* may be set to one of the error code values described for the OSF/1 **wait(2)** function.

See Also

nx_nfork(), **nx_load()**

SETIOMODE() SETIOMODE()

Sets the I/O mode of a file and performs a global synchronization operation.

Synopsis

```
#include <nx.h>

void setiomode(
        int fildes,
        int iomode );
```

Parameters

fildes A file descriptor representing an open file.

iomode The I/O mode to be assigned to the file associated with *fildes*. Values for the *iomode* parameter are as follows:

M_UNIX	Each node has its own file pointer; access is unrestricted.
M_LOG	All nodes use the same file pointer; access is first come, first served; records may be of variable length.
M_SYNC	All nodes use the same file pointer; access is in node order; records are in node order but may be of variable length.
M_RECORD	Each node has its own file pointer; access is first come, first served; records are in node order and of fixed length.

Description

The **setiomode**() function changes the I/O mode of an open shared file. A shared file is a file that is opened for access by more than one node. Shared files opened by **open**() or **fopen**() have the I/O mode **M_UNIX**.

Each node calling **setiomode**() must specify a *fildes* that refers to the same file, and the file pointer must be in the same position in the file for each node at the time the call to **setiomode**() is made.

SETIOMODE() *(cont.)* SETIOMODE() *(cont.)*

In addition to setting the file's I/O mode, **setiomode**() performs a global synchronizing operation like that of the **gsync**() function. That is, all nodes must call the **setiomode**() function before any node can continue executing. In the **M_LOG**, **M_SYNC**, and **M_RECORD** modes, closing the file also performs a global synchronizing operation.

Use the **iomode**() function to return a file's current mode.

Using the **fork**() function, the child process does not inherit the I/O modes associated with the parent process's file descriptors; all I/O modes in the child process default to the mode **M_UNIX**.

M_UNIX (Mode 0)

In this mode, each node maintains its own file pointer and can access information anywhere in the file at any time. If two nodes write to the same place in the file, the latest data written by a node overwrites the data written previously by another node.

This mode is often used when all nodes are only reading a data file or when each node is responsible for data in a specific area of a file.

Because each node can access the file immediately, this mode generally has higher performance than the **M_LOG**, **M_SYNC**, or **M_RECORD** modes.

M_LOG (Mode 1)

In this mode, all nodes use the same file pointer. I/O requests from nodes are handled on a first-come, first-served basis. Because requests can be performed in any order, the order of the data in the file may vary from run to run.

This mode is often used for log files. The files *stdin*, *stdout*, and *stderr* are always opened in this mode.

Because only one node may access the file at a time, this mode has lower performance than the **M_UNIX** mode.

M_SYNC (Mode 2)

In this mode, all nodes use the same file pointer, but I/O requests are handled in node order. This mode treats file accesses as global operations in which all nodes must complete their access before any node can access the file again. The amount of data read or written may, however, vary from node to node.

In this mode, all nodes must perform the same file operations in the same order. The only valid use for the **lseek**() or **eseek**() function is for all nodes to seek to the same position in the file prior to an access.

Because nodes must access the file in node order, this mode has the lowest performance of all the modes.

M_RECORD (Mode 3)

In this mode, each node maintains its own file pointer and can access the file at any time. The data for each corresponding access (that is, the *n*th read or write) must be the same length for all nodes. Because the data from each node appears in the file in a predictable location, each node can access the file whenever it is ready.

Files created in this mode resemble files created in the **M_SYNC** mode. The data appear in node order. All nodes should perform the same file operations in the same order. However, in the **M_RECORD** mode most operations are not synchronized for performance reasons. The operations that are synchronized are the **lseek**() and **eseek**() system calls: the only valid use of one of these calls is for all nodes to seek to the same position in the file prior to an access.

Because nodes may access the file when they are ready, this mode offers better performance than mode **M_SYNC**.

Return Values

Upon successful completion, the **setiomode**() function returns control to the calling process; no values are returned. Otherwise, the **setiomode**() function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_setiomode**() function returns 0 (zero). Otherwise, the **_setiomode**() function returns -1 and sets *errno* to indicate the error.

Errors

If the **_setiomode**() function fails, *errno* may be set to one of the following error code values:

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

EINVAL The given value for *iomode* is not a valid I/O mode.

EMIXIO The given *fildes* is invalid because all nodes have not specified a *fildes* that represents the same file.

SETIOMODE() *(cont.)*

SETIOMODE() *(cont.)*

EMIXIO The given value for *iomode* is not valid because all nodes sharing the file represented by *fildes* have not used the same value.

EMIXIO In I/O modes **M_LOG**, **M_SYNC**, or **M_RECORD**, all nodes sharing the file have not set the file pointer to the same location.

See Also

cread(), **cwrite()**, **iomode()**, **iread()**, **iwrite()**

OSF/1 Programmer's Reference: **dup(2)**, **fork(2)**, **open(2)**

SETPTYPE() SETPTYPE()

Sets the process type of a process.

Synopsis

```
#include <nx.h>

void setptype(
        long ptype );
```

Parameters

ptype	Process type you are assigning to a process. The *ptype* must be a non-negative integer between 0 and 2^{31} - 1.

Description

The **setptype**() function sets the process type of the calling process.Call the **setptype**() function before using any message-passing calls.

Multiple processes running on the same node in the same application must have different process types (ptypes). However, processes on different nodes may (and usually do) have the same process type. Two processes running on a single node may have the same process type only if they belong to different applications.

When you run an application that is linked with the **-nx** switch, the system automatically sets, by default, the process type for all processes in an application to 0 (zero). You can override the default process type in the application's command line with the **-pt** switch.

The **nx_nfork**(), **nx_load**(), and **nx_loadve**() system calls have a *ptype* parameter that lets you specify the process type for newly created processes in an application.

If an application creates additional processes after it starts up, and no process type is specified for the new process, the new process's process type is set to the special value **INVALID_PTYPE** (a negative constant defined in the header file *nx.h*). A process whose process type is **INVALID_PTYPE** cannot send or receive messages. It must call the **setptype**() function to set its process type to a valid value before it can send or receive any messages.

SETPTYPE() *(cont.)* SETPTYPE() *(cont.)*

A process can call the **setptype**() function multiple times to change its process type to a new value or to a previously set value. Once a process has used a process type, it remains associated with the process for the life of the application. No other process in the same application on the same node can use that process type.

The process type in effect when making a send or receive system call determines the process type associated with the message. If a process changes its process type, messages that arrive for the previous process type cannot be received unless the process changes its process type back to the previous value.

Return Values

Upon successful completion, the **setptype**() function returns control to the calling process; no values are returned. Otherwise, this function displays an error message to standard error and causes the calling process to terminate.

Upon successful completion, the **_setptype**() function returns `0` (zero). Otherwise, this function returns `-1` and sets *errno* to indicate the error.

Errors

Refer to the **errno** manual page for a list of *errno* values that can return for errors in C underscore system calls.

See Also

application, **errno**, **myptype**(), **nx_load**(), **nx_nfork**()

Message Types and Typesel Masks

A

Types

The *type* parameter used in message passing calls is a user-defined integer value used to identify the kind of information contained in the message. Types `0` to `999,999,999` are normal types that can be used by any send or receive call.

NOTE

Types 1,000,000,000 to 1,073,741,823 and 2,000,000,000 and up are used by the system and should be avoided. Their use may produce unpredictable results.

Types 1,073,741,824 to 1,999,999,999 are special *force types* intended specifically for the **csendrecv()**, **hsendrecv()**, and **isendrecv()** calls. Force types have three special properties:

1. A message with a force type bypasses the normal flow control mechanisms and is not delayed by clogged message buffers on the sending or receiving node.

2. Force types do not match the `-1` wildcard type selector. This property can be used to guarantee that the message is received by the proper buffer, no matter what other messages are also received.

3. A message with a force type is discarded if no receive is posted (as when the receiving process has been killed). In general, bypassing the normal flow control mechanisms causes no problem because the send-receive calls guarantee that a receive is posted for the message.

Typesel Masks

The *typesel* parameter used in receive calls is an integer value that specifies the type(s) of message you are waiting for in a probe, receive, or flush operation. You assign a *type* to a message when you initiate a send operation. The *typesel* (type selector) allows you to select a specific message type or a set of message types based on a 32-bit mask. The *typesel* can be set as follows:

- If *typesel* is a non-negative integer, a specific message type will be recognized. All other messages will be ignored.
- If *typesel* is -1, the first message to arrive for the process that initiated a probe or receive operation will be recognized. After the first message has been received, you can use `-1` again to receive or probe the next message, and so on.
- If *typesel* is any negative number other than -1, a set of message types will be recognized. In this case, bits 0-29 of the *typesel* correspond to types `0-29`. For example, if bit number 3 is set to 1in the *typesel*, then a message of type 3 will be recognized. If bit number 3 is set to `0`, then a message of type `3` will be ignored.

  Bit 30 allows you to select all types greater than 29 as a group. Bit 30 can be used in conjunction with bits 0-29, as desired. Bit 31 set to 1 makes the *typesel* parameter negative and indicates that it is a mask.

Table A-1 shows the hexadecimal numbers associated with bits 0-31. To generate a mask, add the constant `0x80000000` and the hexadecimal numbers associated with the *types* you want to select. For example, if you want to receive message types `1`, `2`, `5`, and `12`, add the following hex numbers:

```
0x80000000 + 0x2 + 0x4 + 0x20 + 0x1000 = 0x80001026
```

then enter

```
crecv (0x80001026, buf, len);
```

Or, if you want to receive any message except type `0`, use:

```
crecv (0xFFFFFFFE, buf, len);
```

Table A-1. Typesel Mask List (1 of 2)

Type	Hex Number
0	0x00000001
1	0x00000002
2	0x00000004
3	0x00000008

Table A-1. Typesel Mask List (2 of 2)

Type	Hex Number
4	0x00000010
5	0x00000020
6	0x00000040
7	0x00000080
8	0x00000100
9	0x00000200
10	0x00000400
11	0x00000800
12	0x00001000
13	0x00002000
14	0x00004000
15	0x00008000
16	0x00010000
17	0x00020000
18	0x00040000
19	0x00080000
20	0x00100000
21	0x00200000
22	0x00400000
23	0x00800000
24	0x01000000
25	0x02000000
26	0x04000000
27	0x08000000
28	0x10000000
29	0x20000000
Other types	0x40000000

Index

S