April 1996
Order Number: 312487-005

Paragon™ System

C Calls

Reference Manual

Intel® Corporation

WARNING

Some of the circuitry inside this system operates at hazardous energy and electric shock voltage levels. To avoid the risk of personal injury due to contact with an energy hazard, or risk of electric shock, do not enter any portion of this system unless it is intended to be accessible without the use of a tool. The areas that are considered accessible are the outer enclosure and the area just inside the front door when all of the front panels are installed, and the front of the diagnostic station. There are no user serviceable areas inside the system. Refer any need for such access only to technical personnel that have been qualified by Intel Corporation.

CAUTION

This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to Part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference in which case the user will be required to correct the interference at his own expense.

LIMITED RIGHTS

Preface

The Paragon™ 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™ System C 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 Paragon. These calls let you:
  - Create and control parallel applications and partitions.
  - Exchange messages between processes.
  - Get information about the computing environment.
  - 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 in the Parallel File System (PFS).

This manual assumes that you are proficient in using the C programming language and the operating system.

NOTE

Programming examples in this manual are intended only to demonstrate the use of Paragon C system calls; they are not intended as examples of good programming practice. For example, in some cases, the return values of functions are not checked for error conditions. This is not recommended, but the error checks have been omitted in order to make the example shorter and easier to read.

NOTE

Do not use the Mach system call interface. This interface is not supported. It is not documented in SSD manuals, but you may read about Mach elsewhere. If you use Mach system calls, your application may fail. Mach memory allocation and Paragon memory allocation do not work together.

Organization

The manual contains a "manual page" for each operating system C 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).
- Examples.
- Limitations and workarounds information.
- Related calls.

Some of the manual pages in this manual discuss several related system calls. For example, the **cread()** manual page discusses both the **cread()** and **creadv()** 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 system 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.

Appendix B lists the error codes that can be returned in the global variable *errno* by operating system C 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>`**

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

How Errors are Handled

How the operating system operating system handles errors depends on the system call involved:

- For operating system system calls whose names begin with "nx_", 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 operating system system calls (except those whose names begin with "nx_"), the system normally displays a message on the terminal and terminates the calling process.
- For all operating system system calls (except those whose names begin with "nx_"), 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()** system call is the underscore version of the **crecv()** system call. 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 Appendix B that contains the ***errno*** manual page.

Comments and Assistance

Intel Scalable 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

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

United Kingdom Intel Corporation (UK) Ltd.
Scalable Systems Division
Pipers Way
Swindon SN3 IRJ
England
0800 212665 (toll free)
(44) 793 491056
(44) 793 431062
(44) 793 480874
(44) 793 495108

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

Germany Intel Semiconductor GmbH
Dornacher Strasse 1
85622 Feldkirchen bei Muenchen
Germany
0130 813741 (toll free)

World Headquarters
Intel Corporation
Scalable Systems Division
15201 N.W. Greenbrier Parkway
Beaverton, Oregon 97006
U.S.A.
(503) 677-7600 (Monday through Friday, 8 AM to 5 PM Pacific Time)
Fax: (503) 677-9147

If you have comments about our 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

Appendix B
Errno Manual Page

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™ System C 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 errors that can occur in the C underscore system calls.

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

Examples

The following example does a synchronous probe and runs on a two-node partition.

```
#include <nx.h>

#define INIT_TYPE 10

long iam;

main()
{
   char msgbuf[80], smsg[80];

   iam = mynode();
   if(iam==0) {
      sprintf(smsg,"Hello from node %d",iam);
      csend(INIT_TYPE, smsg, sizeof(smsg), -1, 0);
      printf("Node %d sent: %s\n",iam,smsg);
   }
   else {
      cprobe(INIT_TYPE);
      if(infocount() <= sizeof(msgbuf))  {
         crecv(INIT_TYPE, msgbuf, sizeof(msgbuf));
         printf("Node %d received: %s\n",iam,msgbuf);
      }
   }
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

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

CREAD() CREAD()

cread(), **creadv**(): Reads from a file and blocks the calling process until the read completes. (Synchronous read)

Synopsis

```
#include <nx.h>

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

#include <sys/uio.h>

void creadv(
    int fildes,
    struct iovec *iov,
    int iovcount );
```

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.

iov Pointer to an array of *iovec* structures that identifies the buffers into which the data read is placed. The *iovec* structure has the following form:

```
struct iovec {
 caddr_t iov_base;
 int     iov_len;
};
```

The *iovec* structure is defined in the *sys/uio.h* include file.

iovcount Number of *iovec* structures pointed to by the *iov* parameter.

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

Description

Other than return values and an additional error, the **cread()** and **creadv()** functions are identical to the OSF/1 **read()** and **readv()** functions, respectively. See the **read(2)** manual page in the *OSF/1 Programmer's Reference.*

These calls are synchronous system calls. The calling process waits (blocks) until the read completes. Use the **iread()** or **ireadv()** function to read a file without blocking the calling process.

NOTE

To preserve data integrity, all I/O requests are processed on a "first-in, first-out" basis. This means that if an asynchronous I/O call is followed by a synchronous I/O call on the same file, the synchronous call will block until the asynchronous operation has completed.

Reading past the end of a file causes an error. You can do one of the following to prevent end-of file errors:

- Use the **iseof()** function to detect end-of-file before calling the **cread()** or **creadv()** functions.
- Use the **lseek()** function to determine the length of a file before calling the **cread()** or **creadv()** functions.
- Use the **_cread()** or **_creadv()** function to detect end-of-file or that the number of bytes read is less than the number of bytes requested.

Return Values

Upon successful completion, the **cread()** and **creadv()** functions return control to the calling process; no values are returned. Otherwise, the **cread()** and **creadv()** functions write an error message on the standard error output and cause the calling process to terminate.

Upon successful completion, the **_cread()** and **_creadv()** functions return the number of bytes read. Otherwise, these functions return -1 and set *errno* to indicate the error. These functions return 0 (zero) if end-of-file is reached.

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

Errors

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

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

Examples

The following example does a synchronous read and runs in a multi-node partition. Note that the file */tmp/mydata* must exist in order for this example to correctly execute.

```
#include <memory.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <nx.h>

long iam;

main()
{
   int fd;
   struct stat result;
   char msgbuf[100];

   iam = mynode();

   memset(msgbuf,0,100);

   fd = gopen("/tmp/mydata", O_RDWR, M_UNIX, 0644);
   fstat(fd, &result);
   if (!iseof(fd)) {
     cread(fd, msgbuf, result.st_size);
     printf("Node %d read: %s",iam,msgbuf);
   }
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

cwrite(), **gopen()**, **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™ System C Calls Reference Manual* for more information about message type selectors.
buf	Points to the buffer where the message should be placed.
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...()** functions.

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. If the message is too long for the *buf* buffer, your application terminates with an error and the receive does not complete.

After the **crecv()** function completes, you can use the **info...()** functions 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* parameter.

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). Note that posting too many asynchronous calls can cause the application to deplete the available pool of message IDs. If no message IDs are available, **crecv()** and **crecvx()** may fail with your application terminating and the synchronous receive function not completing.

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 **_crecvx()** functions return -1 when an error occurs and set *errno* to indicate the error. Otherwise, these functions return 0.

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

Errors

The **_crecv()** and **_crecvx()** functions can return the following *errno* value:

EQMSGLONG The message received was too long for the *buf* message buffer.

EQNOMID The application has too many outstanding message requests from asynchronous system calls. No message IDs are available from the system for the synchronous receive.

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

Examples

The following example uses the **crecv()** function to do a synchronous receive. The example can run in a multi-node partition.

```
#include <nx.h>

#define INIT_TYPE 10

long iam;

main()
{
   char msgbuf[80], smsg[80];

   iam = mynode();
   if(iam==0) {
      sprintf(smsg,"Hello from node %d\n",iam);
      csend(INIT_TYPE, smsg, strlen(smsg)+1, -1, 0);
      printf("Node %d sent: %s",iam,smsg);
   }
   else {
      cprobe(INIT_TYPE);
      if(infocount() <= sizeof(msgbuf)) {
         crecv(INIT_TYPE, msgbuf, sizeof(msgbuf));
         printf("Node %d received: %s\n",iam,msgbuf);
      }
   }
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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™ System C Calls Reference Manual* for more information about message types.
buf	Points 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. 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. 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.

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 errors that can occur in the C underscore system calls.

Examples

The following example uses the **csend()** function to do a synchronous send. The example can run in a multi-node partition.

```
#include <nx.h>

#define INIT_TYPE 10

long iam;

main()
{
   char msgbuf[80], smsg[80];

   iam = mynode();
   if(iam==0) {
      sprintf(smsg,"Hello from node %d\n",iam);
      csend(INIT_TYPE, smsg, strlen(smsg)+1, -1, 0);
      printf("Node %d sent: %s",iam,smsg);
   }
   else {
      cprobe(INIT_TYPE);
      if(infocount() <= sizeof(msgbuf)) {
         crecv(INIT_TYPE, msgbuf, sizeof(msgbuf));
         printf("Node %d received: %s\n",iam,msgbuf);
      }
   }
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

cprobe(), **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™ System C Calls Reference Manual* for information on message types.
sbuf	Points to the buffer of the message to send.
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™ System C Calls Reference Manual* for more information about message type selectors.
rbuf	Points to the buffer where the message should be placed.
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 received message is too long for the *rbuf* buffer when using the **csendrecv()** function, your application terminates with an error and the receive does not complete. If the received message is too long for the *rbuf* buffer when using the **_csendrecv()** function, 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.

If you use force-type messages with the **csendrecv()** function, you are responsible for posting the receive on the receiving node before the message arrives. Otherwise, the receive will not complete and the message will be lost. The **csendrecv()** function does not do internal synchronization of messages. See Appendix A, "Message Types and Typesel Masks" on page A-1 of the *Paragon™ System C Calls Reference Manual* for more information on force-type messages.

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 complete list of errors that can occur in the C underscore system calls.

EINVAL The received message is too long for the receive buffer.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

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

CWRITE() CWRITE()

cwrite(), **cwritev**(): Writes to a file and blocks the calling process until the write completes. (Synchronous write)

Synopsis

```
#include <nx.h>

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

#include <sys/uio.h>

void cwritev(
    int fildes,
    struct iovec iov[],
    int iovcount );
```

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.

iov Pointer to an array of *iovec* structures, which identifies the buffers containing the data to be written. The *iovec* structure has the following form:

```
struct iovec {
 caddr_t iov_base;
 int     iov_len;
};
```

The *iovec* structure is defined in the *sys/uio.h* include file.

iovcount Number of *iovec* structures pointed to by the *iov* parameter.

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

Description

Other than the return values and an additional error, the **cwrite()** and **cwritev()** functions are identical to the OSF/1 **write()** and **writev()** functions, respectively. See the **write(2)** manual page in the *OSF/1 Programmer's Reference*.

These are synchronous system calls. The calling process waits (blocks) until the write completes. Use the **iwrite()** or **iwritev()** function to write a file without blocking the calling process.

NOTE

To preserve data integrity, all I/O requests are processed on a "first-in, first-out" basis. This means that if an asynchronous I/O call is followed by a synchronous I/O call on the same file, the synchronous call will block until the asynchronous operation has completed.

Use the **iseof()** function to determine whether the write moved the file pointer to the end of the file.

Return Values

Upon successful completion, the **cwrite()** and **cwritev()** functions return control to the calling process; no values are returned. Otherwise, the **cwrite()** and **cwritev()** functions write an error message on the standard error output and cause the calling process to terminate.

Upon successful completion, the **_cwrite()** and **_cwritev()** function return the number of bytes written. Otherwise, these functions return -1 and set *errno* to indicate the error.

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

Examples

The following example does a synchronous write.

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

long iam;

main()
{
   int fd;
   char buffer[80];

   iam = mynode();

   fd = gopen("/tmp/mydata",O_CREAT | O_TRUNC | O_RDWR, M_LOG,
            0644);
   sprintf(buffer,"Hello from node %d\n",iam);
   cwrite(fd, buffer, strlen(buffer));
   close(fd);
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

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

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

DCLOCK() DCLOCK()

Returns time in seconds since the system was booted.

Synopsis

```
#include <nx.h>

double dclock(void);
```

Description

The **dclock()** function measures time intervals in seconds. The time is obtained from the RPM global clock. The **dclock()** value rolls over approximately every 14 years, and has an accuracy of 100 nanoseconds on each node and 1 microsecond across all nodes.

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 errors that can occur in the C underscore system calls.

DCLOCK() *(cont.)*

DCLOCK() *(cont.)*

Examples

The following example uses the **dclock()** function to calculate the elapsed time of a program segment.

```
#include <nx.h>

long iam;

main()
{
   double start_time, end_time, elapsed_time;

   iam = mynode();
   start_time = dclock();
   sleep(5);

   end_time = dclock();
   elapsed_time = end_time - start_time;
   printf("\nNode %d elapsed time = %f\n",iam,elapsed_time);
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

***errno*, rpm**

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 signed 64-bit integers with values from -2**63 to 2**63 - 1. Extended-integer functions are for accessing extended file sizes in the Parallel File System (PFS).

Use these functions 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.

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 write an error message on the standard error output 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 both 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 or division by zero.

EADD() *(cont.)*

EADD() *(cont.)*

Examples

The following example uses the extended mathematical functions to do calculations on some extended integers.

```
#include <nx.h>

void display();

long iam;

main()
{
   static char *three = {"3"};
   static char *four  = {"4"};
   char ss[20];
   long r,r4;
   esize_t e3, e4, e_sum, e_sub, e_mul;

   printf("\n");
   e3 = stoe(three);
   e4 = stoe(four);
   r4 = 4;

   e_sum = eadd(e3,e4);
   display("e_sum = ",e_sum);

   e_sub = esub(e4,e3);
   display("e_sub = ",e_sub);

   e_mul = emul(e3,100);
   display("e_mul = ",e_mul);

   r = ecmp(e3,e4);
   printf("e_cmp = %ld\n",r);

   r = emod(e3,r4);
   printf("e_mod = %ld\n",r);

   r = ediv(e3,r4);
   printf("e_div = %ld\n",r);
}
```

EADD() *(cont.)*

EADD() *(cont.)*

```
void display(ss,eout)
char *ss;
esize_t eout;
{
   char s[20];
   etos(eout,s);
   printf("%s%s\n",ss,s);
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

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

ESEEK() ESEEK()

Moves a file's read-write file pointer.

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.

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 *offset* bytes.

Description

You can use the **eseek()** function to access regular files and extended files, while the **lseek()** function does not support extended files. A regular file cannot exceed 2G - 1 bytes.

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

ESEEK() *(cont.)*

ESEEK() *(cont.)*

This function may block while asynchronous I/O requests queued by the same process to the same file complete.

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.

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;
```

Because regular files cannot exceed 2G - 1 bytes, the resulting file offset must not exceed 2G - 1 bytes when moving the file pointer of a non-extended file. However, when working with extended files, the theoretical resulting file offset can reach a 64-bit value. Realistically though, the file offset depends on how many file systems the extended file is stripped across. Thus, any call to **eseek()** that results in a file offset that exceeds the system-dependent limit produces an error.

When the **eseek()** function does not successfully complete, it writes an error message on the standard error output and causes the calling process to terminate.

Upon successful completion, the **_eseek()** function returns the same value as the **eseek()** function. Otherwise, the **_eseek()** function returns -1 in both 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.

ESEEK() *(cont.)*

ESEEK() *(cont.)*

EFBIG The resulting offset as determined by the *whence* and *offset* parameters exceeds the maximum file offset allowable for this type of file on this particular file system.

Examples

The following example shows how to use the **eseek()** function to move the file pointer in a file.

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

long iam;

main()
{
   int fd;
   esize_t offset, new_size, new_pos;
   char s[20];

   fd = gopen("/tmp/mydata", O_RDWR, M_UNIX, 0644);

   offset = stoe("1000");
   new_size = esize(fd,offset,SIZE_SET);
   etos(new_size,s);
   printf("new size = %s\n", s);

   offset = stoe("500");
   new_pos = eseek(fd,offset,SEEK_SET);

   etos(new_pos,s);
   printf("new position = %s\n", s);
   close(fd);
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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 file 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. See the OSF/1 **truncate()** manual page for information about decreasing a file's size.

You can use the **esize()** function to access regular files and extended files, while the **lsize()** function does not support extended files. Extended files can have a size a greater than 2G - 1 bytes, while regular files cannot.

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.

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). If the new size specified by the *offset* and *whence* parameters is greater that the available disk space, the **esize()** function allocates what disk space is available and returns the new size of the file. Otherwise, the **esize()** function writes an error message on the standard error output and causes the calling process to terminate.

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 both fields of the **esize_t** structure and sets *errno* to indicate the error.

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.)*

Notes

Since NFS does not support disk block preallocation, **esize()** and **_esize()** are not supported on files that reside in remote file systems that have been NFS mounted. The **esize()** and **_esize()** functions are supported only on files in UFS and PFS file systems.

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.

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. The **esize()** function does not support files that reside in remote file systems and have been NFS mounted.

EINVAL The file is not a regular file.

ENOSPC No space left on device.

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

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

Examples

The following example shows how to use the **esize()** function to increase the size of a file.

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

long iam;

main()
{
   int fd;
   esize_t offset, new_size, new_pos;
   char s[20];

   fd = gopen("/tmp/mydata", O_RDWR, M_UNIX, 0644);

   offset = stoe("1000");
   new_size = esize(fd,offset,SIZE_SET);
   etos(new_size,s);
   printf("new size = %s\n", s);

   offset = stoe("500");
   new_pos = eseek(fd,offset,SEEK_SET);

   etos(new_pos,s);
   printf("new position = %s\n", s);
   close(fd);
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

eseek(), lsize()

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

ESTAT() ESTAT()

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

Synopsis

```
#include <nx.h>

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

long lestat(
    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.

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

The *estat* structure has the following form:

```
struct estat {
        dev_t   st_dev;
        ino_t   st_ino;
        mode_t  st_mode;
        nlink_t st_nlink;
        uid_t   st_uid;
        gid_t   st_gid;
        dev_t   st_rdev;
        esize_t st_size;
        time_t  st_atime;
        int     st_spare1;
        time_t  st_mtime;
        int     st_spare2;
        time_t  st_ctime;
        int     st_spare3;
        ulong_t st_blksize;
        long    st_blocks;
        ulong_t st_flags;
        ulong_t st_gen;

};
```

fildes File descriptor for an extended file or standard OSF/1 file open for writing. A standard OSF/1 file cannot be greater than 2G - 1 bytes.

Description

You can use the **estat()**, **lestat()**, and **festat()** functions to access regular files and extended files, while the **stat()**, **lstat()**, and **fstat()** functions do not support extended files. Extended files can have a size a greater than 2G - 1 bytes, while regular files cannot.

The **estat()**, **lestat()**, and **festat()** function semantics are identical to the OSF/1 **stat()**, **lstat()**, and **fstat()** functions, respectively. 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 the *sys/estat.h* header file.

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

The **lestat()** function is like the **estat()** function, except when the named file is a symbolic link. In this case, the **lestat()** function returns information about the link. The **estat()** and **festat()** functions return information about the file the link references. For symbolic links, the **lestat()** function sets the *st_size* field of the *estat* structure to the length of the symbolic link, and sets the *st_mode* field to indicate the file type.

The **festat()** function is identical to the **estat()** function except it returns information about an open file specified by the *fildes* parameter.

Return Values

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

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

Examples

The following example shows how to use the **festat()** and **estat()** functions to access statistics about files:

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

void display();

main()
{
   int fd;
   struct estat result;

   fd = gopen("/tmp/mydata", O_RDWR, M_UNIX, 0644);
```

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

```
    festat(fd,&result);
    printf("st_atime = %ld\n",result.st_atime);
    display("st_size = ",result.st_size);
    estat("/tmp/mydata",&result);
    printf("st_atime = %ld\n",result.st_atime);
    display("st_size = ",result.st_size);
    close(fd);
}


void display(ss,eout)
char *ss;
esize_t eout;
{
char s[20];
    etos(eout,s);
    printf("%s%s\n",ss,s);
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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 null-terminated character string.

Description

Extended integers are signed 64-bit integers with values from -2**63 to 2**63 - 1. Always use the extended-integer functions to access extended integers. The following functions perform conversion operations for extended integers:

etos()	Converts an extended integer to a character string.
stoe()	Converts a null-terminated character string 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.

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 both 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**63 -1 or an overflow occurs.

EQESIZE Illegal character in string for the **_stoe()** function.

Examples

The following example shows how to use the conversion functions for extended integers:

```
#include <nx.h>

void display();

long iam;

main()
{
   static char *three = {"3"};
   static char *four  = {"4"};
   char ss[20];

   long r,r4;
   esize_t e3, e4, e_sum, e_sub, e_mul;

   printf("\n");
   e3 = stoe(three);
```

ETOS() *(cont.)*

ETOS() *(cont.)*

```
    e4 = stoe(four);
    r4 = 4;

    e_sum = eadd(e3,e4);
    display("e_sum = ",e_sum);

    e_sub = esub(e4,e3);
    display("e_sub = ",e_sub);

    e_mul = emul(e3,100);
    display("e_mul = ",e_mul);

    r = ecmp(e3,e4);
    printf("e_cmp = %ld\n",r);

    r = emod(e3,r4);
    printf("e_mod = %ld\n",r);

    r = ediv(e3,r4);
    printf("e_div = %ld\n",r);

}

void display(ss,eout)
char *ss;
esize_t eout;
{
char s[20];
    etos(eout,s);
    printf("%s%s\n",ss,s);
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

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

FCNTL() FCNTL()

fcntl(), dup(), dup2(): Controls open file descriptors.

Synopsis

```
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#include <pfs/pfs.h>

int fcntl (
    int filedes,
    int request [ ,
    int argument | struct flock *argument ] );

int dup(
    int filedes );

int dup2(
    int old,
    int new );
```

Parameters

filedes	Specifies an open file descriptor obtained from a successful **gopen()**, **open()**, **fcntl()**, or **pipe()** function.
request	Specifies the operation to be performed.
argument	Specifies a variable that depends on the value of the *request* parameter.
old	Specifies an open file descriptor.
new	Specifies an open file descriptor that is returned by the **dup2()** function.

FCNTL() *(cont.)*

FCNTL() *(cont.)*

The following are values for the *request* parameter:

F_DUPFD Returns a new file descriptor as follows:

- Lowest numbered available file descriptor greater than or equal to the *argument* parameter, taken as type **int.**
- Same object references as the original file.
- Same file pointer as the original file. (That is, both file descriptors share one file pointer if the object is a file).
- Same access mode (read, write, or read-write).
- Same file status flags. (That is, both file descriptors share the same file status flags).
- The close-on-exec flag (FD_CLOEXEC bit) associated with the new file descriptor is cleared so that the file will remain open across **exec** functions.

F_SVR_BUFFER

Enables or disables PFS buffering for the file referenced by the *filedes* parameter. The argument parameter is interpreted as a boolean: TRUE enables server buffering; FALSE disables it. The fileservers cache stripe-file data in their memory-resident, disk-block caches. These fileservers use a read-ahead and write-behind caching algorithm. PFS buffering is recommended only when the IO request size is less than 64K bytes; otherwise, the fieservers's cache may thrash. Dirty cache buffers are flushed to disk when **F_SVR_BUFFER** changes from TRUE to FALSE.

FCNTL() *(cont.)*

FCNTL() *(cont.)*

F_GETSATTR

Gets the PFS stripe attributes of the file referred to by the *filedes* parameter. The argument parameter is taken as a pointer to a *sattr* structure, in which the stripe attributes are returned. The structure sattr has the following form:

```
struct sattr {
   size_t s_sunitsize;  /* stripe unit size */
   uint_t s_sfactor;    /* stripe factor    */
   uint_t s_start_sdir; /* base stripe dir  */
}
```

The stripe attributes returned are a subset of the default stripe attributes for the PFS file system in which the file resides, and consist of:

- The file's *stripe unit size*, in bytes. This is the unit of data interleaving used in the PFS file.
- The file's *stripe factor*. This is the size of the PFS file's stripe group. The file is striped in a round robin fashion to the number of stripe directories specified by this value.
- The file's *base stripe directory*. This is the stripe directory at which striping begins for the file. Stripe directories define the storage locations for the PFS file. The ordered set of stripe directories across which the file is striped define the file's stripe group. When a PFS file is created, it inherits its default stripe group from the PFS file system in which the file resides. (The file system stripe group is specified by the system administrator when the file system is mounted.) By default, the base stripe directory for a newly created file is selected randomly from the file's stripe group.

  When specified in the *sattr* structure, the base stripe directory is represented as an index between 0 and *stripe_factor*-1, inclusive, where *stripe_factor* is the default stripe factor of the PFS file. The file is striped in a round-robin fashion to stripe directories starting at this location.

FCNTL() *(cont.)*

FCNTL() *(cont.)*

F_SETSATTR
Sets the PFS stripe attributes of the file referred to by the *filedes* parameter. The *argument* parameter is taken as a pointer to a *sattr* structure which contains the file's new stripe attributes. The base stripe directory and the stripe factor must specify a subset of the PFS file's stripe group; in other words, the base stripe directory must be between 0 and *stripe_factor*-1 and the stripe factor must be less than or equal to *stripe_factor*, where *stripe_factor* is the current stripe factor of the PFS file.

F_GETFD Gets the value of the close-on-exec flag associated with the file descriptor *filedes*. File descriptor flags are associated with a single file descriptor and do not affect other file descriptors that refer to the same file. The *argument* parameter is ignored.

F_SETFD Sets the close-on-exec flag associated with the *filedes* parameter to the value of the *argument* parameter, taken as type **int.** If the *argument* parameter is 0 (zero), the file remains open across the **exec** functions. If the *argument* parameter is FD_CLOEXEC, the file is closed on successful execution of the next **exec** function.

F_GETFL Gets the file status flags and file access modes for the file referred to by the *filedes* parameter. The file access modes can be extracted by using the mask O_ACCMODE on the return value. File status flags and file access modes are associated with the file description and do not affect other file descriptors that refer to the same file with different open file descriptions. The *argument* parameter is ignored.

F_SETFL Sets the file status flags to the *argument* parameter, taken as type **int**, for the file to which the *filedes* parameter refers. The file access mode is not changed.

F_GETOWN Gets the process ID or process group currently receiving SIGIO and SIGURG signals. Process groups are returned as negative values.

F_SETOWN Sets the process or process group to receive SIGIO and SIGURG signals. Process groups are specified by supplying the *argument* parameter as negative; otherwise the *argument* parameter, taken as type **int,** is interpreted as a process ID.

FCNTL() *(cont.)*

FCNTL() *(cont.)*

The following values for the *request* parameter are available for record locking:

F_GETLK Gets the first lock that blocks the lock description pointed to by the *argument* parameter, taken as a pointer to type **struct flock**. The information retrieved overwrites the information passed to the **fcntl()** function in the **flock** structure. If no lock is found that would prevent this lock from being created, then the structure is left unchanged except for the lock type, which is set to F_UNLCK.

F_SETLK Sets or clears a file segment lock according to the lock description pointed to by *argument,* taken as a pointer to type **struct flock**. F_SETLK is used to establish shared locks (F_RDLCK), or exclusive locks (F_WRLCK), as well as remove either type of lock (F_UNLCK). If a shared (read) or exclusive (write) lock cannot be set, the **fcntl()** function returns immediately with a value of -1.

F_SETLKW Same as F_SETLK except that if a shared or exclusive lock is blocked by other locks, the process will wait until it is unblocked. If a signal is received while **fcntl()** is waiting for a region, the function is interrupted, -1 is returned, and **errno** is set to [EINTR].

FCNTL() *(cont.)* FCNTL() *(cont.)*

Description

The **fcntl()** function performs controlling operations on the open file specified by the *filedes* parameter.

The **fcntl(), dup(),** and **dup2()** functions, which suspend the calling process until the request is completed, are redefined so that only the calling thread is suspended.

When used to permanently set the stripe attributes of a file, you can only use **F_SETSATTR** on a PFS file that has not yet been written to (it is zero-length). Once set, the new attributes of the file are permanent; further attempts to reset the attributes of the file will result in an error. Whenever an **F_SETATTR** request is completed successfully, the file pointer for *filedes* resets to point to the beginning of the file.

The **F_SETSATTR** request also allows the stripe attributes of an already written-to file to be *temporarily* mapped to new attributes if the file is opened read-only. In this case, the new attributes apply only to the file descriptor specified by the *filedes* parameter, and go away when the file is closed. This remapping can be useful for writing a matrix out to a file using one type of decomposition, and reading the matrix back in using a different decomposition.

For a simple example, consider an 8x8 matrix with a record size of 4K bytes and a total of 64 records. If this matrix is written to a PFS file with a stripe factor of 8 and a stripe unit size of 32K bytes, the matrix will automatically be written using a column decomposition. If the stripe attributes of the file are then mapped to use a stripe unit size of 4K bytes, the matrix is read back in using a row decomposition.

The stripe attributes of a PFS file can also be displayed from the command line by using the **-P** switch with the **ls** command. See the **ls(1)** man page for more information.

The O_NDELAY and O_NONBLOCK requests affect only operations against file descriptors derived from the same **open()** function. In BSD, these apply to all file descriptors that refer to the object.

When a shared lock is set on a segment of a file, other processes are able to set shared locks on that segment or a portion of it. A shared lock prevents any other process from setting an exclusive lock on any portion of the protected area. A request for a shared lock fails if the file descriptor was not opened with read access.

An exclusive lock prevents any other process from setting a shared lock or an exclusive lock on any portion of the protected area. A request for an exclusive lock fails if the file descriptor was not opened with write access.

The **flock()** structure describes the type (**l_type**), starting offset (**l_whence**), relative offset (**l_start**), size (**l_len**) and process ID (**l_pid**) of the segment of the file to be affected.

FCNTL() *(cont.)*

FCNTL() *(cont.)*

The value of **l_whence** is set to SEEK_SET, SEEK_CUR or SEEK_END, to indicate that the relative offset **l_start** bytes is measured from the start of the file, from the current position, or from the end of the file, respectively. The value of **l_len** is the number of consecutive bytes to be locked. The **l_len** value may be negative (where the definition of **off_t** permits negative values of **l_len).** The **l_pid** field is only used with F_GETLK to return the process ID of the process holding a blocking lock. After a successful F_GETLK request, the value of **l_whence** becomes SEEK_SET.

If **l_len** is positive, the area affected starts at **l_start** and ends at **l_start + l_len - 1**. If **l_len** is negative, the area affected starts at **l_start + l_len** and ends at **l_start - 1**. Locks may start and extend beyond the current end of a file, but may not be negative relative to the beginning of the file. If **l_len** is set to 0 (zero), a lock may be set to always extend to the largest possible value of the file offset for that file. If such a lock also has **l_start** set to 0 (zero) and **l_whence** is set to SEEK_SET, the whole file is locked. Changing or unlocking a portion from the middle of a larger locked segment leaves a smaller segment at either end.

Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect. All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process in a **fork()** function.

If a regular file has enforced record locking enabled, record locks on the file will affect calls to other calls, including **creat(), open(), read(), write(), truncate(),** and **ftruncate().**

A potential for deadlock occurs if a process controlling a locked region is put to sleep by attempting to lock another process' locked region. If the system detects that sleeping until a locked region is unlocked would cause a deadlock, the **fcntl()** function fails with an [EDEADLK] error.

Notes

Care should be used when attempting to set the stripe attributes of a file that is opened from multiple nodes. Use of the **F_SETSATTR** request on a file descriptor does not affect other already-existing descriptors for the same file. Possible file corruption could result if the file is then written to using any of the already-existing descriptors. For example, if a file is opened by multiple nodes and then a single node sets the stripe attributes, the new attributes are only visible to that node. The other nodes must close and reopen the file to get the new attributes. For performance reasons, issue the **F_SETSATTR** request from *only one node*, rather than from all nodes running the application.

The **dup(***filedes)* function is equivalent to **fnctl(***filedes, F_DUPFD, 0)*.

The **dup2(***old, new)* function is equivalent to **fcntl(***old, F_DUPFD, new)*.

The file locks set by the **fcntl()** and **lockf()** functions do not interact in any way with the file locks set by the **flock() function.** If a process sets an exclusive lock on a file using the **fcntl()** or **lockf()** function, the lock will not affect any process that is setting or clearing locks on the same file using the **flock()** function. It is therefore possible for an inconsistency to arise if a file is locked by different processes using **flock()** and **fcntl().** (The **fcntl()** and **lockf()** functions use the same mechanism for record locking.)

FCNTL() *(cont.)* FCNTL() *(cont.)*

Return Values

Upon successful completion, the value returned depends on the value of the *request* parameter as follows:

F_DUPFD Returns a new file descriptor.

F_GETSATTR Returns 0 (zero).

F_SETSATTR Returns 0 (zero).

F_GETFD Returns **FD_CLOEXEC** or 0 (zero).

F_SETFD Returns a value other than -1.

F_GETFL Returns the value of file status flags and access modes. (The return value will not be negative.)

F_SETFL Returns a value other than -1.

F_GETOWN Returns the value of descriptor owner.

F_GETLK Returns a value other than -1.

F_SETLK Returns a value other than -1.

F_SETLKW Returns a value other than -1.

If the **fcntl()** function fails, a value of -1 is returned and **errno** is set to indicate the error.

Errors

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

EBADF The *filedes* parameter is not a valid open file descriptor.

EBADF The *request* parameter is **F_SETLK** or **F_SETLKW**, the type of lock (**l_type**) is a shared lock (F_RDLCK), and *filedes* is not a valid file descriptor open for reading.

EBADF The type of lock (**l_type**) is an exclusive lock (**F_WRLCK**), and *filedes* is not a valid file descriptor open for writing.

FCNTL() *(cont.)*

FCNTL() *(cont.)*

EBADF The *request* parameter is **F_SETSATTR** but the file's stripe attributes have already been permanently set by a previous call to **fcntl()**.

EEXIST The *request* parameter is **F_SETSATTR** but the file is not zero-length, or is not open read-only.

ENOTPFS The file referred to by the *filedes* parameter is not a PFS file; i.e., it is not a regular file in a PFS file system.

EMFILE The *request* parameter is **F_DUPFD** and **OPEN_MAX** file descriptors are currently open in the calling process, or no file descriptors greater than or equal to *argument* are available.

EINVAL The set of attributes specified by the *sattr* structure is not a subset of the default stripe attributes of the PFS file system in which the file resides.

EINVAL The *request* parameter is **F_DUPFD** and the *argument* parameter is negative or greater than or equal to **OPEN_MAX**.

EINVAL An illegal value was provided for the *request* parameter.

EINVAL The *request* parameter is **F_GETLK**, **F_SETLK**, or **F_SETLKW** and the data pointed to by *argument* is invalid, or *filedes* refers to a file that does not support locking.

EFAULT The *argument* parameter is an invalid address.

ESRCH The value of the *request* parameter is **F_SETOWN** and the process ID given as *argument* is not in use.

EAGAIN The ***request*** parameter is F_SETLK, the type of lock (**l_type**) is a shared (F_RDLCK) or exclusive (**F_WRLCK**) lock, and the segment of a file to be locked is already exclusive-locked by another process.

EAGAIN The *request* parameter is **F_SETLK**, and the type is an exclusive lock and some portion of the segment of a file to be locked is already shared-locked or exclusive-locked by another process.

EINTR The *request* parameter is **F_SETLKW** and the **fcntl()** function was interrupted by a signal which was caught.

ENOLCK The *request* parameter is **F_SETLK** or **F_SETLKW** and satisfying the lock or unlock request would result in the number of locked regions in the system exceeding a system-imposed limit.

FCNTL() *(cont.)* FCNTL() *(cont.)*

EDEADLK The *request* parameter is **F_SETLKW**, the lock is blocked by some lock from another process and putting the calling process to sleep, and waiting for that lock to become free would cause a deadlock.

If the **dup()** or **dup2()** function fails, **errno** may be set to one of the following values:

EBADF The *filedes* or *old* parameter is not a valid open file descriptor or the *new* parameter file descriptor is negative or greater than **OPEN_MAX**.

EMFILE The number of file descriptors exceeds **OPEN_MAX** or there is no file descriptor above the value of the *new* parameter.

EINTR The **dup2()** function was interrupted by a signal which was caught.

Examples

This example creates a new file, reads and prints its default striping attributes, sets new striping attributes, and then closes the file. After closing the file the example opens the file and gets the new striping attributes and prints them.

```
#include <stdio.h>
#include <sys/stat.h>
#include <stdarg.h>
#include <fcntl.h>
#include <sys/param.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/mount.h>
#include <pfs/pfs.h>
#include <errno.h>
#include <nx.h>

#define         PERMS   0777
#define         FILE    "/pfs/my_file"

main()
{
int             fd;
struct sattr    sattr;
/* Create new file */
    if ((fd = creat(FILE, PERMS)) == -1) {
        perror("creat");
        exit(1);
    }
```

FCNTL() *(cont.)* FCNTL() *(cont.)*

```
/* Gets current stripe attributes */
    if (fcntl(fd, F_GETSATTR, &sattr) != 0) {
        perror("default get fcntl");
        exit(1);
    }
/* Prints stripe attributes */
    printf("Default attributes for %s\n", FILE);
    printf("----------------------------------\n");
    printf("Stripe Unit Size:(s_sunitsize): %d\n",
           sattr.s_sunitsize);
    printf("Stripe Factor:(s_sfactor):      %d\n",
           sattr.s_sfactor);
    printf("Stripe Index:(s_start_sdir):    %d\n",
           sattr.s_start_sdir);
    printf("\n");

    if (2 > sattr.s_sfactor) {
    printf("New stripe factor must be less than or equal to\n");
    printf("existing default stripe factor.\n");
    printf("Read the comments at the beginning of this source\n");
    printf("code for more details.\n");
    exit(1);
    }

/* Update the sattr structure with the new stripe attributes so
they can be written later */
    sattr.s_sunitsize = 63556;
    sattr.s_sfactor = 2;
    sattr.s_start_sdir = 0;

/* Sets new stripe attributes */
    if (fcntl(fd, F_SETSATTR, &sattr) != 0) {
        perror("New set fcntl");
        exit(1);
    }

/* Close file */
    if (close(fd) != 0) {
        perror("close");
        exit(1);
    }

/* Open file */
    if ((fd = open(FILE, O_RDONLY)) == -1) {
        perror("open");
```

FCNTL() *(cont.)*

FCNTL() *(cont.)*

```
        exit(1);
    }

    /* Gets current stripe attributes */
        if (fcntl(fd, F_GETSATTR,&sattr) != 0 ) {
            perror("New get fcntl");
            exit(1);
        }

    /* Prints stripe attributes */
        printf("New attributes for %s\n", FILE);
        printf("------------------------------\n");
        printf("Stripe unit size:(s_sunitsize): %d\n",
               sattr.s_sunitsize);
        printf("Stripe Factor:(s_sfactor):      %d\n",
               sattr.s_sfactor);
        printf("Stripe Index:(s_start_sdir):    %d\n",
               sattr.s_start_sdir);
        printf("\n");

    /* Close file */
        if(close(fd) != 0) {
            perror("close");
            exit(1);
        }
    }
```

See Also

commands: **ls**(1), **showfs**(1)

Functions: **close**(2), **exec**(2), **gopen**(3), **lockf(3), open**(2), **read**(2), **setiomode**((3), **truncate**(2), **write**(2)

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.

The **flick()** function works differently depending 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 errors that can occur in the C underscore system calls.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

errno

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

FORK_REMOTE_CTL() FORK_REMOTE_CTL()

Enables or disables remote process creation.

Synopsis

```
#include <sll/sll.h>

int fork_remote_ctl(
    int flag);
```

Parameters

flag Specifies whether processes can be created on nodes other than the nodes on which the application is running. The flag value must be one of the following:

ENABLE_FORK_REMOTE
Enables remote process creation.

DISABLE_FORK_REMOTE
Disables remote process creation.

The *flag* values are specified in the include file *sll/sll.h*.

Description

The **fork_remote_ctl()** function is only available for the system administrator.

The **fork_remote_ctl()** function allows an application to create processes on nodes other than the node the application is running on. The bootmagic string *ENABLE_FORK_REMOTE* must be set to **t** or **T** (true) for remote process creation to work.

The **fork_remote_ctl()** function only specifies whether processes can be created on a remote node using the **fork()** function.

The **fork_remote_ctl()** function only affects the node it is executed on and only prevents remote process creation for processes originating on that node. Other nodes can still create processes remotely on a node, even if the **fork_remote_ctl()** with **DISABLE_FORK_REMOTE** has been executed on the node.

FORK_REMOTE_CTL() *(cont.)*

FORK_REMOTE_CTL() *(cont.)*

Return Values

If **fork_remote_ctl()** succeeds, it returns 0. If an error occurs, **fork_remote_ctl()** returns -1 and sets *errno* to indicate the error.

Errors

EINVAL The *flag* parameter was neither **ENABLE_FORK_REMOTE** nor **DISABLE_FORK_REMOTE**.

ENOSYS The boot magic string **ENABLE_FORK_REMOTE** has been set to **FALSE** at boot time.

EPERM The effective user ID of the calling process is not *root*.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

bootmagic, **load_leveld**, **parameters**

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>*.

FPGETROUND() *(cont.)* FPGETROUND() *(cont.)*

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.

FPGETROUND() *(cont.)* FPGETROUND() *(cont.)*

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 errors that can occur in the C underscore system calls.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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 errors that can occur in the C underscore system calls.

Examples

The following example shows how to use the **gcol()** function to do a global collect from all nodes in an application:

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

#define M    4
#define N   16

void display();

long iam, nbrnodes;

main()
{

   int    i, count=0;
   double x[M], y[N], dot, norm, dummy;
   char   msg[80];
   int    dpsize = 8;

   iam      = mynode();
```

GCOL() *(cont.)*

GCOL() *(cont.)*

```
    nbrnodes = numnodes();
    dot      = 0.0;
    for(i=0; i<M; i++) {
       x[i] = (double) (iam * M + i);
       printf("Node %d x[%d] = %3.1f\n",iam,i,x[i]);
    }

    for(i=0; i<M; i++)
       dot += x[i]*x[i];
    printf("Node %d dot = %f\n",iam,dot);

    gdsum(&dot, 1, &dummy);
    sprintf(msg,"dot = %f\n",dot);
    display(msg);

    norm = sqrt(dot);

    for(i=0; i<M; i++)
       x[i] = x[i]/norm;

    gcol(x, M*dpsize, y, nbrnodes*M*dpsize, &count);

    if(!iam) {
       for(i=0;i<nbrnodes*M; i++)
          printf("%3.1f ",y[i]);
       printf("\n");
    }
}

void display(dmsg)
char *dmsg;
{
    if(!iam) printf("\n%s",dmsg);
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

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

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 errors that can occur in the C underscore system calls.

Examples

The following example shows how to use the **gcolx()** function to do a global collect from all nodes in an application:

```
#include <math.h>

#define M    4
#define N   16

void display();

long iam, nbrnodes;

main()
{

   int    i, count=0;
   double x[M], y[N], dot, norm, dummy;
   char   msg[80];
   int    dpsize = 8;
   long   xlen[4];

   iam      = mynode();
   nbrnodes = numnodes();
   dot      = 0.0;
```

GCOLX() *(cont.)* GCOLX() *(cont.)*

```
   for(i=0; i<nbrnodes; i++)
      xlen[i] = 4*sizeof(double);

   for(i=0; i<M; i++) {
      x[i] = (double) (iam * M + i);
      printf("Node %d x[%d] = %3.1f\n",iam,i,x[i]);
   }

   for(i=0; i<M; i++)
      dot += x[i]*x[i];
   printf("Node %d dot = %f\n",iam,dot);

   gdsum(&dot, 1, &dummy);
   sprintf(msg,"dot = %f\n",dot);
   display(msg);

   norm = sqrt(dot);

   for(i=0; i<M; i++)
      x[i] = x[i]/norm;

   gcolx(x, xlen, y);

   if(!iam) {
      for(i=0;i<nbrnodes*M; i++)
         printf("%3.1f ",y[i]);
      printf("\n");
   }
}

void display(dmsg)
char *dmsg;
{
   if(!iam) printf("\n%s",dmsg);
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

GCOLX() *(cont.)*

GCOLX() *(cont.)*

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 data in which to find the maximum. The final result of the global maximum operation is returned in this buffer.
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 errors that can occur in the C underscore system calls.

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

Examples

The following example shows how to use the **gdhigh()** function to determine the maximum value across all nodes of an application:

```
#include <nx.h>

long iam;
main() {
   int i, numElements, maxElement, list[50];

   numElements = 10;
   iam = mynode();
   for(i=0;i<10;i++)
      list[i] = iam*10 + i;
   if(iam==0) {
      for(i=0;i<numElements;i++)
         printf(" %d:list[%d] = %d\n",iam,i,list[i]);
      gsync();
   }
   else {
      gsync();
      for(i=0;i<numElements; i++)
         printf(" %d:list[%d] = %d\n",iam,i,list[i]);
   }
   maxElement = findMin(list,numElements);
   if (iam == 0)
      printf("Max is %d\n",maxElement);
}
int findMin(list,numElements)
int list[];
int numElements;
{
   int maxElement, index;
   int temp,k;
   index = 0;
   for(k=1; k<numElements; k++)
      if (list[k] > list[index])
            index = k;
   maxElement = list[index];
   printf("%d: maxElement = %d\n",iam, maxElement);
   gihigh(&maxElement,1,&temp);
   return(maxElement);
}
```

GDHIGH() *(cont.)*

GDHIGH() *(cont.)*

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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 data in which to find the minimum. The final result of the global minimum operation is returned in this buffer.
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 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 errors that can occur in the C underscore system calls.

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

Examples

The following example shows how to use the **gilow()** function to determine the minimum value across all nodes of an application:

```
#include <nx.h>

long iam;

main() {
   int i, iam, numElements, minElement,  list[50];

   numElements = 10;
   iam = mynode();
   for(i=0;i<10;i++)
      list[i] = iam*10 + i;

   if(iam==0) {
      for(i=0;i<numElements; i++)
         printf(" %d:list[%d] = %d\n",iam,i,list[i]);
      gsync();
   }
   else {
      gsync();
      for(i=0;i<numElements; i++)
         printf(" %d:list[%d] = %d\n",iam,i,list[i]);
   }
   minElement = findMin(list,numElements);
   if (iam == 0)
      printf("Min is %d\n",minElement);
}

int findMin(list,numElements)
int list[];
int numElements;
{
   int minElement, index;
   int temp,k;
```

```
    index = 0;
    for(k=1; k<numElements; k++)
       if (list[k] < list[index])
            index = k;
    minElement = list[index];
    gilow(&minElement,1,&temp);
    return(minElement);
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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 data for the multiplication. The final result of the global multiplication operation is returned in this buffer.
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 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 errors that can occur in the C underscore system calls.

GDPROD() *(cont.)*

GDPROD() *(cont.)*

Examples

The following example shows how to use the **giprod**() function to determine a product across all nodes of an application:

```
#include <nx.h>

long iam;

main()
{
   long final, initial;
   long x[5], work[5];
   int i;

   iam = mynode();
   if(!iam) {
      for(i=0;i<5;i++)
         x[i] = i;
   }
   else {
      for(i=0; i<5; i++)
         x[i] = i;
   }

   if(!iam) {
      printf("\n");
      for(i=0;i<5;i++) {
         printf("%d ",x[i]);
      }
      printf("\n");
   }

   giprod(x,5,work);

   if(!iam) {
      printf("\n");
      for(i=0;i<5;i++) {
         printf("%d ",x[i]);
      }
      printf("\n");
   }
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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 buffer that contains the data for the addition. The final result of the global addition operation is returned in this buffer.
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 errors that can occur in the C underscore system calls.

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

Examples

The following example shows how to use the **gisum()** function to determine a sum across all nodes of an application:

```
#include <nx.h>

long iam;
main()
{
   long final, initial;
   long x[5], work[5];
   int i;

   iam = mynode();
   if(!iam) {
      for(i=0;i<5;i++)
         x[i] = i;
   }
   else {
      for(i=0; i<5; i++)
         x[i] = i;
   }

   if(!iam) {
      printf("\n");
      for(i=0;i<5;i++) {
         printf("%d ",x[i]);
      }
      printf("\n");
   }

   gisum(x,5,work);

   if(!iam) {
      printf("\n");
      for(i=0;i<5;i++) {
         printf("%d ",x[i]);
      }
      printf("\n");
   }
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

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

GETPFSINFO() GETPFSINFO()

Get the stripe attributes of mounted Parallel File System (PFS).

Synopsis

```
#include <nx.h>
#include <pfs/pfs.h>

long getpfsinfo(
    struct pfsmntinfo **attrbufp );
```

Parameters

attrbufp Points to the array of *pfsmntinfo* structures that describe the stripe attributes of each currently mounted PFS file system. The *pfsmntinfo* structure is defined in the *pfs/pfs.h* header file and has the following form:

```
struct pfsmntinfo {
     char m_mntonname[];
     struct statpfs m_statpfs;
};
```

Description

The **getpfsinfo()** function returns the mount point and stripe attributes of each currently mounted PFS file system. The **getpfsinfo()** function returns this information in the *attrbufp* parameter, which is an array of *pfsmntinfo* structures. This information is contained in a static area, so you must copy the information to save it.

The *pfsmntinfo* structure consists of two elements, the pathname of the file system mount point and a *statpfs* structure. The *pfsmntinfo* structure is of variable length, since the *statpfs* structure contains a variable number of variable length pathnames (see the description of the *p_sdirs* field).

GETPFSINFO() *(cont.)*

GETPFSINFO() *(cont.)*

The fields of the *pfsmntinfo* structure are:

m_mntonname Directory name on which the PFS file system has been mounted.

m_statpfs The *statpfs* structure which describes the PFS file system.The *pfsmntinfo* structure is defined in the *pfs/pfs.h* header file and has the following form:

```
struct statpfs {
        uint_t          p_reclen;
        long            p_magic;
        size_t          p_sunitsize;
        uint_t          p_sfactor;
        uint_t          p_reserved[2];
        pathname_t      p_sdirs;

};
```

The fields of the *statpfs* structure include the following:

p_reclen Length of this *statpfs* structure.

p_sunitsize The stripe unit size for the parallel file system, in bytes; that is, the size of the unit of data interleaving for regular files.

p_sfactor The number of stripe units per file stripe; that is, the degree of interleaving for regular files.

p_sdirs A list of pathnames specifying the set of directories that define the stripe group for this Parallel File System. The number of pathnames in the list is equal to *p_sfactor*. Each pathname is of type *pathname_t*. The pathname list can be traversed with a pointer of type (*pathname_t* *) and the use of the **NEXTPATH()** macro defined in the *pfs/pfs.h* header file.

To obtain general mount information for all types of mounted file systems, use the standard OSF/1 **getmntinfo()** function.

Return Values

Upon successful completion, the **getpfsinfo()** function returns a count of the number of elements in the array. If an error occurs, the **getpfsinfo()** function returns a value of -1 and sets *errno* to indicate the error (*attrbufp* is left unmodified).

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

mount, **mount()**, **showfs()**, **statpfs()**

OSF/1 Programmer's Reference: **getmntinfo(3)**, **mount(2)**, **mount(8)**, **statfs(2)**

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 data for the AND operation. The final result of the global AND operation is returned in this buffer.
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.

GIAND() *(cont.)* GIAND() *(cont.)*

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 errors that can occur in the C underscore system calls.

Examples

The following example shows how to use the **giand()** function to perform a global AND across all nodes of an application:

```
#include <nx.h>

long iam;

main()
{
   long final, initial;
   long x[5], work[5];
   int i;

   iam = mynode();
   if(!iam)
      for(i=0;i<5;i++)
         x[i] = i;
   else
      for(i=0; i<5; i++)
         x[i] = ~i;
```

```
        if(!iam) {
           printf("\n");
           for(i=0;i<5;i++)
              printf("%d ",x[i]);
           printf("\n");
        }

        giand(x,5,work);

        if(!iam) {
           printf("\n");
           for(i=0;i<5;i++)
              printf("%d ",x[i]);
           printf("\n");
        }
    }
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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 data for the OR operation. The final result of the global OR operation is returned in this buffer.
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 errors that can occur in the C underscore system calls.

Examples

The following example shows how to use the **gior()** function to perform a global OR across all nodes of an application:

```
#include <nx.h>

long iam;
main()
{
   long final, initial;
   long x[5], work[5];
   int i;

   iam = mynode();
   if(!iam) {
      for(i=0;i<5;i++)
         x[i] = i;
   }
   else {
      for(i=0; i<5; i++)
         x[i] = ~i;
   }
```

GIOR() *(cont.)*

GIOR() *(cont.)*

```
        if(!iam) {
           printf("\n");
           for(i=0;i<5;i++) {
              printf("%d ",x[i]);
           }
           printf("\n");
        }

        gior(x,5,work);

        if(!iam) {
           printf("\n");
           for(i=0;i<5;i++) {
              printf("%d ",x[i]);
           }
           printf("\n");
        }
    }
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

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

GOPEN() GOPEN()

Performs a global open of a file for reading or writing, sets the I/O mode of the file, and performs a global synchronization.

Synopsis

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

int gopen(
    const char *path,
    int oflag,
    int iomode,
    mode_t mode );
```

Parameters

path Pointer to a pathname of the file to be opened or created. If the *path* parameter refers to a symbolic link, the **gopen()** function opens the file pointed to by the symbolic link.

oflag Specifies the type of access, special open processing, the type of update, and the initial state of the open file. The parameter value is constructed by logically ORing the special open processing flags. These flags are defined in the *fcntl.h* header file and are described in the OSF/1 **open(2)** manual page.

iomode I/O mode to be assigned to the file associated with *fildes*. Values for the *mode* 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.

GOPEN() *(cont.)*

GOPEN() *(cont.)*

M_RECORD Each node has its own file pointer; access is first come, first served; records are in node order and of fixed length.

M_GLOBAL All nodes use the same file pointer, all nodes perform the same operations.

M_ASYNC Each node has its own file pointer; access is unrestricted; I/O atomicity is not preserved in order to allow multiple readers/multiple writers and records of variable length.

Refer to the **setiomode()** manual page for detailed information on each I/O mode.

mode Specifies the read, write, and execute permissions of the file to be created (requested by the **O_CREAT** flag in the **gopen()** interface). If the file already exists, this parameter is ignored. This parameter is constructed by logically ORing values described in the *sys/mode.h* header file.

Description

The **gopen()** function allows all nodes in an application to open and share the same file. The **gopen()** function performs a *global* open; all nodes can open the same file without issuing multiple I/O requests.

Other than the addition of the *iomode* parameter, additional return values, and additional errors, the semantics of the **gopen()** function are identical to the OSF/1 **open()** function. See the **open(2)** manual page in the *OSF/1 Programmer's Reference*.

You can use the **gopen()** function to specify the I/O mode of a shared file when it is opened, rather than requiring an additional call to the **setiomode()** subroutine. This improves performance when many nodes open and set the I/O mode of the same file. You use the *iomode* parameter to specify a file's I/O mode. See the **setiomode()** manual page for a description of the file I/O modes.

Use the **setiomode()** function to change a file's I/O mode after the file is opened. Use the **iomode()** function to return a unit's current I/O mode.

The **gopen()** function globally synchronizes all nodes in an application. Therefore, all the application's nodes must call the **gopen()** function before any node can continue executing. In the **M_LOG**, **M_SYNC**, **M_RECORD**, and **M_GLOBAL** I/O modes, closing the file also performs a global synchronizing operation.

GOPEN() *(cont.)* GOPEN() *(cont.)*

When using the OSF/1 **fork()** function to create new processes, the default I/O mode for the child process's file descriptors is determined by the file type (PFS or non-PFS) and the setting of the bootmagic variable *PSF_ASYNC_DFLT*. For information on how this default I/O mode is determined, see the **setiomode()** manual page description.

When using the OSF/1 **dup()** function to duplicate a file, the file descriptor for the duplicate file is reset to the I/O mode **M_UNIX**.

Return Values

Upon successful completion, **gopen()** returns the file descriptor representing the open file. If an error occurs, **gopen()** writes an error message on the standard error output, and causes the calling process to terminate.

Upon successful completion, the **_gopen()** function returns the file descriptor representing the open file. Otherwise, this function returns a value of -1 and sets *errno* to indicate the error.

Errors

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

EINVAL The given value for iomode is not valid.

EINVAL The file named by the *path* parameter is not a regular file.

EMIXIO The given *path* is invalid because all nodes sharing the file have not specified the same *path*.

EMIXIO The given value for *iomode* is not valid because all nodes sharing the file named by *path* have not used the same value.

GOPEN() *(cont.)* GOPEN() *(cont.)*

Examples

The following example shows how to use the **gopen()** function to open a file for writing:

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

long iam;

main()
{
   int fd;
   char buffer[80];

   iam = mynode();

   fd = gopen("/tmp/mydata",O_CREAT | O_TRUNC | O_RDWR, M_LOG,
             0644);
   sprintf(buffer,"Hello from node %d\n",iam);
   cwrite(fd, buffer, strlen(buffer));
   close(fd);
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

cread(), **cwrite()**, **eseek()**, **estat()**, **iread()**, **iseof()**, **iwrite()**, **setiomode()**

OSF/1 Programmer's Reference: **chmod(2)**, **close(2)**, **dup(2)**, **fcntl(2)**, **lockf(2)**, **lseek(2)**, **open(2)**, **read(2)**, **stat(2)**, **truncate(2)**, **umask(2)**, **write(2)**

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: the first parameter must be the same as the *x* parameter and the second parameter must be the same as the *work* parameter.

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 errors that can occur in the C underscore system calls.

Examples

The following example shows how to use the **gopf()** function in an application. The example distributes a vector over the nodes of a partition. Node 1 has the maximum element.

The global function specified in the **gopf()** function must have two parameters: the input value and an array for the contributions of other nodes. The following is an example of a global function **max_node()**. This function finds the maximum element and returns a structure that contains the maximum value and the number of the node on which it resides.

```
#include <math.h>

long max_node();

struct PIVOT_NODE {
   double max;
   long node;
};
```

GOPF() *(cont.)*

GOPF() *(cont.)*

```
main()
{
   struct PIVOT_NODE mine,work;
   double            x[10];
   long               iam, i, max_loc, xlen, N;

   N = 10;
   xlen      = sizeof(x);
   mine.node = mynode();
   iam       = mine.node;
   max_loc   = 0;

   for(i=0;i<N;i++)
      x[i] = (double) (iam*N + i);

   if(iam ==1) x[4] = 100.00;

   mine.max  = fabs(x[0]);

   if(iam==0) {
      printf("\n");
      printf(" %2d:  ",iam);
      for(i=0;i<N; i++)
         printf("  %3.1f ",x[i]);
      printf("\n");
      gsync();
  }
  else {
      gsync();
      printf(" %2d:  ",iam);
      for(i=0;i<N; i++)
         printf("  %3.1f",x[i]);
      printf("\n");
  }
  for(i=1; i<N; i++) {
     if(mine.max < fabs(x[i])) {
        mine.max = fabs(x[i]);
        max_loc  = i;
     }
  }
```

GOPF() *(cont.)* GOPF() *(cont.)*

```
    gopf((char *)&mine, sizeof(mine), (char *)&work, max_node);

    if(iam==0) {
       printf("mine.max  = %f\n",mine.max);
       printf("mine.node = %d\n",mine.node);
    }
}

long max_node(mine,work)
char *mine, *work;
{
    struct PIVOT_NODE *smine, *swork;
    int iam;

    iam = mynode();
    smine = (struct PIVOT_NODE *)mine;
    swork = (struct PIVOT_NODE *)work;

    if( smine->max <= swork->max) {
       smine->max = swork->max;
       smine->node = swork->node;
    }
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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™ System C 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.

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 errors that can occur in the C underscore system calls.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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 errors that can occur in the C underscore system calls.

GSYNC() *(cont.)* GSYNC() *(cont.)*

Examples

The following example shows how to use the **gsync()** subroutine to synchronize an application running on multiple nodes in a partition:

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

#define MAX_IDS   900

main()
{

     long   n, node;
     long   my_node, num_nodes;
     long   rmid[MAX_IDS];
     char   rbuf[10], sbuf[10];

     my_node = mynode();
     num_nodes = numnodes();

     if(my_node == 0) {
        printf("Starting ...\n");
     }

     /* Post receives */
     for(node = 0; node < num_nodes; node++) {
        rmid[node] = irecv(1, rbuf, 10);
     }

     /* Send a message to each node */
     for(node = 0; node < num_nodes; node++) {
        csend(1, sbuf, 10, node, 0);
     }

     /* Check received messages */
     for(node = 0; node < num_nodes; node++) {
        msgwait(rmid[node]);
     }

     /* Wait for all nodes to complete */
     gsync();
```

```
        if(my_node == 0) {
           printf("Finished!\n");
        }

}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

errno

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™ System C Calls Reference Manual* for more information about message type selectors.
buf	Address of the buffer where the message is received.
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 message is stored in the buffer *buf*, the calling process is interrupted, and the specified handler is started. After the handler is started, the handler and the calling process may run concurrently until the handler finishes. (In previous releases of the operating system operating system, the calling process was interrupted and did not run at all until the handler returned.)

The handler contains code that you write to process the message or information about the message after the message is received. The handler receives the following information about a message: the message's type, length, sending node, and process type. 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). If the received message is too long for the buffer *buf*, the receive completes, no error is returned, the content of *buf* is undefined, and this argument is set to 0 (zero).

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.

HRECV() *(cont.)*

HRECV() *(cont.)*

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 );
```

Because the handler and the main program may run concurrently, parts of the main program may have to be protected from being executed at the same time as the handler. Use the **masktrap**() function to ensure a critical section of code in the main program is not interrupted by the execution of the handler. If a handler is active when a **masktrap**() function is called in the main program, the main program blocks in the **masktrap**() call until the handler completes. See the **masktrap**() manual page for more information about using the **masktrap**() function to protect a section of code from interrupts.

NOTE

The **masktrap()** function may be called from a handler, but it is unnecessary and has no effect. This is supported because code that calls the **masktrap()** function may be used by both the handler and the main program. The purpose of the **masktrap()** function is to protect the main program from the handler.

CAUTION

The handler runs in the same memory space as the main program (but they have separate stacks).

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

These calls are asynchronous system calls. 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 receive system calls (for example, **irecv**()).

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.

NOTE

Once you have established a handler for a message type, do not attempt to receive a message of that type with a **crecv...()** or **irecv...()** call.

NOTE

There are a limited number of message IDs available for applications. Applications that use the **irecv()** and **irecvx()** functions must explicitly release unused message IDs. If an application runs out of message IDs, the application may fail. This can affect the **hrecv()** and **hrecvx()** functions, because they use message IDs internally.

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.

Errors

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

HRECV() *(cont.)*

HRECV() *(cont.)*

Examples

The following example shows how to use the **hrecv()** function in a message passing application running on two nodes. The example posts an **hrecv()** to receive a message type 100, and on receipt executes a handler named *proc*.

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

void proc();
long iam;
main() {

   char buf[80];
   long mask;

   iam = mynode();
   memset(buf,0,80);

   if(iam == 0) {
      printf("\n%d: Before hrecv\n", iam);
      hrecv(100,buf,sizeof(buf),proc);
      mask = masktrap(1);
      printf("%d: After hrecv\n", iam);
      printf("%d Waiting ...\n",iam);
      masktrap(mask);
      sleep(5);
      printf("%d Completed \n",iam);
   }
   else {
      sleep(1);
      sprintf(buf,"Hello from node %d\n",iam);
      printf("Node 1 sends to node 0\n");
      csend(100,buf,strlen(buf),0,0);
   }
}

void proc(type,count,node,pid)
long type, count, node, pid;
{
   printf("%d Entered handler:\n", iam);
   printf("%d type = %d\n",iam, type);
   printf("%d count = %d\n",iam, count);
   printf("%d node = %d\n",iam, node);
   printf("%d pid = %d\n",iam, pid);
}
```

HRECV() *(cont.)*

HRECV() *(cont.)*

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

cprobe(), **csend()**, **crecv()**, **csendrecv()**, ***errno***, **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™ System C Calls Reference Manual* for information on message types.
buf	Points 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 four 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. This user-written 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. 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. After the handler is started, the handler and the calling process may run concurrently until the handler finishes. (In previous releases of the operating system operating system, the calling process was interrupted and did not run at all until the handler returned.)

CAUTION

The handler runs in the same memory space as the main program (but they have separate stacks).

Because of this, parts of the main program may have to be protected from being executed at the same time as the handler.

The handler contains user-written code that runs after the send buffer is available for reuse. The handler receives information about the message including the message's type, length, receiving node, and process type.

HSEND() *(cont.)*

HSEND() *(cont.)*

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()**).

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.)*

NOTE

There are a limited number of message IDs available for applications. Applications that use the **isend()** and **isendx()** functions must explicitly release unused message IDs. If an application runs out of message IDs, the application may fail. This can affect the **hsend()** and **hsendx()** functions, because they use message IDs internally.

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 errors that can occur in the C underscore system calls.

Examples

The following example shows how to use the **hsend()** function in a message passing application running on two nodes. The example uses the **hsend()** function to send a message and execute a handler named *proc* after the send completes.

```
#include <string.h>
#include <memory.h>
#include <nx.h>

void proc();
long iam;

main() {

   char buf[80], rbuf[80];
   long mask;
```

HSEND() *(cont.)*

HSEND() *(cont.)*

```
    iam = mynode();
    memset(buf,0,80);
    memset(rbuf,0,80);
    if(iam == 0) {
       sprintf(buf,"Hello from node %d\n",iam);
       printf("\n%d: Before hsend\n", iam);
       hsend(100,buf,strlen(buf)+1,1,0,proc);
       mask = masktrap(1); /* Disable traps */
       printf("%d: After hsend: %s\n", iam,buf);
       printf("Waiting ...\n");
       mask = masktrap(mask); /* Enable traps */
       sleep(5);
    }
    else {
       printf("Node 1 receives from node 0\n");
       crecv(100,rbuf,sizeof(rbuf));
       printf("%d: %s\n",iam,rbuf);
    }
}


void proc(type,count,node,pid)
long type, count, node, pid;
{
    printf("Node %d Entered handler:\n",iam);
    printf("type = %d\n",type);
    printf("count = %d\n",count);
    printf("node = %d\n",node);
    printf("pid = %d\n",pid);
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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™ System C Calls Reference Manual* for information on message types.
sbuf	Points 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™ System C Calls Reference Manual* for more information about message type selectors.

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

rbuf Points to the buffer for storing the reply.

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 system call. The function sends a message and immediately posts a receive, specifying the handler to be invoked when the receive completes. The calling process continues to run until the receive completes. When the receive completes, the calling process is interrupted and the specified handler is started. After the handler is started, the handler and the calling process may run concurrently until the handler finishes. (In previous releases of the operating system operating system, the calling process was interrupted and did not run at all until the handler returned.)

CAUTION

The handler runs in the same memory space as the main program (but they have separate stacks).

Because of this, parts of the main program may have to be protected from being executed at the same time as the handler.

The handler contains code that you write to process the message or information about the message after the message is received. The handler receives the following information about the received message: the message's type, length, sending node, and process type.

If the send part of the **hsendrecv()** function fails, the receive is never posted. The send buffer is not available for reuse until after returning from the handler.

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

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). If the received message is too long for the buffer *rbuf*, the receive completes, no error is returned, the content of *rbuf* is undefined, and this argument is set to 0 (zero).
node	The node of the process that sent the message.
ptype	The process type of the process that sent the message.

This is an asynchronous system call. To block the calling process until the send/receive completes, use the synchronous system call **csendrecv()**. To do an asynchronous send/receive in which a message ID (MID) is provided to determine when the receive completes, use the system call **isendrecv()**.

The handler must have the following form:

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

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.

NOTE

There are a limited number of message IDs available for applications. Therefore, applications need to release unused message IDs. The **hsendrecv()** function uses message IDs internally, but does not return message IDs, like the **isendrecv()** function does. The handlers associated with the **hsendrecv()** function releases these message IDs.

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

Return Values

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

Upon successful completion, the **_hsendrecv()** 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 errors that can occur in the C underscore system calls.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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* parameter in the **msgwait**() or **msgdone**() functions represents a merged message ID (that is, it was returned by the **msgmerge**() function), the information returned for the **info...**() calls is unpredictable.

INFOCOUNT() *(cont.)* INFOCOUNT() *(cont.)*

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.

If you issue an **info...()** call before doing any message passing, the call returns -1.

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 errors that can occur in the C underscore system calls.

Examples

The following example shows how to use the **info...()** functions to get information about a message in an application.

```
long iam;

main()
{
   long node, type, ptype, count;
   char rmsg[80],smsg[80];

   iam = mynode();
```

INFOCOUNT() *(cont.)* INFOCOUNT() *(cont.)*

```
        if(!iam) {
           sprintf(smsg,"Hello from node %d\n",iam);
           csend(100,smsg,strlen(smsg) + 1,1,0);
        }
        else {
           crecv(100,rmsg,sizeof(rmsg));
           node = infonode();
           type = infotype();
           ptype  = infoptype();
           count = infocount();
           printf("node = %d\n",node);
           printf("type = %d\n",type);
           printf("ptype = %d\n",ptype);
           printf("count = %d\n",count);
        }
    }
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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	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 specified in an **iread()** call contains valid data (if the *id* parameter identifies a read operation).
- The buffer specified in an **iwrite()** call is available for reuse (if the *id* parameter identifies a write operation).
- The I/O ID specified by the *id* parameter is released for use in another 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 Read or write is not yet complete.

1 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.

Examples

The following example shows how to use the **iodone()** function to determine if an asynchronous write is complete:

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

long iam;

main()
{
   int fd, id;
   long mode;
   char buffer[80];

   iam = mynode();
```

IODONE() *(cont.)* IODONE() *(cont.)*

```
        fd = gopen("/tmp/mydata",O_CREAT | O_TRUNC | O_RDWR, M_UNIX,
                  0644);

        mode = iomode(fd);
        if(!iam) printf("%d: iomode = %d\n",iam, mode);

        sprintf(buffer,"Hello from node %d\n",iam);
        id = iwrite(fd, buffer, strlen(buffer));
        while (!iodone(id))
           printf("%d: write not done\n",iam);
        printf("%d: write done\n",iam);

        close(fd);
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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 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**, **M_RECORD**, **M_GLOBAL**, or **M_ASYNC**. Refer to the **setiomode()** manual page for descriptions of each I/O mode.

If an error occurs, the **iomode()** function writes an error message on the standard error output 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.

IOMODE() *(cont.)* IOMODE() *(cont.)*

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.

Examples

The following example show how to use the **iomode()** function to determine the I/O mode of an opened file:

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

long iam;

main()
{
   int fd, id;
   long mode;
   char buffer[80];

   iam = mynode();

   fd = gopen("/tmp/mydata",O_CREAT | O_TRUNC | O_RDWR, M_UNIX,
            0644);
   mode = iomode(fd);
   if(!iam) printf("%d: iomode = %d\n",iam, mode);

   sprintf(buffer,"Hello from node %d\n",iam);
   id = iwrite(fd, buffer, strlen(buffer));

   iowait(id);
   close(fd);
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

gopen(), **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	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 specified in an **iread()** call contains valid data (if the *id* parameter identifies a read operation).
- The buffer specified in an **iwrite()** call is available for reuse (if the *id* parameter identifies a write operation).
- The I/O ID specified by the *id* parameter is released for use in another 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.

IOWAIT() *(cont.)*

IOWAIT() *(cont.)*

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 0. 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.

Examples

The following example shows how to use the **iowait()** function to determine if an asynchronous write has completed:

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

long iam;

main()
{
   int fd, id;
   char buffer[80];
   iam = mynode();

   fd = gopen("/tmp/mydata",O_CREAT | O_TRUNC | O_RDWR, M_UNIX,
            0644);
   sprintf(buffer,"Hello from node %d\n",iam);
   id = iwrite(fd, buffer, strlen(buffer));
   printf("%d: write not done\n",iam);
   iowait(id);
   printf("%d: write done\n",iam);
   close(fd);
}
```

IOWAIT() *(cont.)*

IOWAIT() *(cont.)*

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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™ System C 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 errors that can occur in the C underscore system calls.

Examples

The following example shows how to use the **iprobe()** function to determine whether an asynchronous message is ready to be received:

```
long iam;

main() {
   long msgid, probe;
   char smsg[80], rmsg[80];

   iam = mynode();

   sprintf(smsg,"Hello from node %d\n",iam);

   probe = iprobe(-1);
   printf("%d: Before send iprobe = %d\n",iam,probe);

   csend(100, smsg, strlen(smsg)+1 ,-1,0);
   sleep(5);
   probe = iprobe(-1);
   printf("%d: After send iprobe = %d\n",iam,probe);

   msgid = irecv(100, rmsg, sizeof(rmsg));
   msgwait(msgid);

   printf("%d: received: %s\n",iam,rmsg);
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

IPROBE() *(cont.)*

IPROBE() *(cont.)*

See Also

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

IREAD() IREAD()

iread(), **ireadv**(): Reads from a file and returns immediately. (Asynchronous read)

Synopsis

```
#include <nx.h>

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


#include <sys/uio.h>

long ireadv(
    int fildes,
    struct iovec *iov,
    int iovcount );
```

Parameters

fildes File descriptor identifying the open file to be read.

buffer Pointer to the buffer in which the data is placed after it is read.

nbytes Number of bytes to read from the file associated with the *fildes* parameter.

iov Pointer to an array of *iovec* structures that identifies the buffers into which the data is placed. The *iovec* structure has the following form:

```
struct iovec {
     caddr_t iov_base;
     int     iov_len;
};
```

The *iovec* structure is defined in the *sys/uio.h* include file.

iovcount Number of *iovec* structures pointed to by the *iov* parameter.

IREAD() *(cont.)* IREAD() *(cont.)*

Description

Other than the return values, the additional errors, and the asynchronous behavior, the **iread()** and **ireadv()** functions are identical to the OSF/1 **read()** and **readv()** functions, respectively. See the **read(2)** manual page in the *OSF/1 Programmer's Reference.*

The **iread()** and **ireadv()** functions are asynchronous system calls. These functions return 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 either the **cread()** or **creadv()** function (synchronous system calls) instead of the **iread()** or **ireadv()** function, respectively.
- Use **iowait()** to wait until the read completes.
- Loop until **iodone()** returns 1, indicating that the read is complete.

NOTE

To preserve data integrity, all I/O requests are processed on a "first-in, first-out" basis. This means that if an asynchronous I/O call is followed by a synchronous I/O call on the same file, the synchronous call will block until the asynchronous operation has completed.

After an **iread()** or **ireadv()** call, you can perform other read or write calls on the same file without waiting for the read to finish.

Use the **iseof()** function to determine whether the file pointer is at the end of the file.

Return Values

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

Upon successful completion, the **_iread()** and **_ireadv()** functions return a non-negative I/O ID. The I/O ID is for use by the **iodone()** and **iowait()** functions. Otherwise, the **_iread()** and **_ireadv()** functions return -1 when an error occurs and set *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 returned I/O ID as soon as possible by calling **iodone()** or **iowait()**.

Errors

If the **_iread()** or **_ireadv()** 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** and **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.

Examples

The following example shows how to use the **iread()** and **iowait()** functions to do an asynchronous read:

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

long iam;

main()
{
   int fd,id;
   char msgbuf[18];

   iam = mynode();

   fd = gopen("/tmp/mydata", O_RDWR, M_UNIX, 0644);
   id = iread(fd, msgbuf, sizeof(msgbuf));
   iowait(id);
```

```
        printf("Node %d read: %s\niseof = %d\n",iam,msgbuf,
               iseof(fd));

}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

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

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

IREADOFF() IREADOFF()

ireadoff(), **ireadvoff**(): Asynchronous reads from a file at a specified offset.

Synopsis

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

long ireadoff(
        int fildes,
        esize_t offset,
        char *buffer,
        unsigned int nbytes );


#include <sys/types.h>
#include <sys/uio.h>

long ireadvoff(
        int fildes,
        esize_t offset,
        struct iovec *iov,
        int iovcount );
```

Description of Parameters

fildes	A file descriptor identifying the file to be read.
offset	Offset from the beginning of the file where to begin the read.
buffer	Pointer to the buffer in which the data is stored after it is read.
nbytes	The number of bytes to read from the file associated with the *fildes* parameter.
iov	Pointer to an array of *iovec* structures that identify the buffers into which the data is to be placed.
iovcount	The number of *iovec* structures pointed to by the *iov* parameter.

IREADOFF() *(cont.)* IREADOFF() *(cont.)*

Discussion

Ireadoff() reads *nbytes* asynchronously from the file specified by the descriptor *fd* starting at the offset specified by *offset* into the buffer pointed to by *buffer*. **Ireadvoff()** is similar, but it reads the data into the iovcount buffers specified by *iov*.

Ireadoff() and **ireadvoff**() are similar to **iread**() and **ireadv**() except for reading starting at a user-specified offset (instead of the offset maintained by the system file pointer) and the following additional differences:

- The current value of the system file pointer is not modifed.
- Currently only M_UNIX and M_ASYNC I/O modes are supported.
- Paragon PFS is the only file system type that currently supports these functions.

Return Values

Upon successful completion, a non-negative I/O ID for use in **iodone**(),**iowait**(), **niodone**() and **niowait**() calls is returned. If an error occurs, these functions return -1 and set *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()**, **iowait()**, **niodone()** or **niowait()**.

IREADOFF() *(cont.)*

IREADOFF() *(cont.)*

Errors

Errors are as described in OSF/1 **read()**, except that the following errors can also occur:

EMREQUEST An asynchronous 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.

EFSNOTSUPP The file referred to by *filedes* is not in a file system of a type that supports this operation. Currently only the PFS file systems support this operation.

EINVAL The file referred to by *filedes* is in an unsupported iomode. Currently only M_UNIX and M_ASYNC are supported.

See Also

cread(), gopen(), iodone(), iowait(), iread(), iseof(), niodone(), niowait(), readoff() setiomode()

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

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™ System C Calls Reference Manual* for more information about message type selectors.
buf	Points to the address where the message should be placed.
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.

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

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.

NOTE

The number of message IDs is limited. The error message "Too many requests" is returned and your application will stop when no message IDs are available for a requested asynchronous send or receive. Your program should release its message IDs as soon as possible by calling **msgcancel()**, **msgdone()**, **msgignore()**, or **msgwait()**.

Errors

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

EQNOMID Your application has used all the available message IDs and no message IDs are available. Use either the **msgcancel**(), **msgdone**(), **msgignore**(), or **msgwait**() subprogram with the receive to release message IDs.

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

IRECV() *(cont.)*

IRECV() *(cont.)*

Examples

The following example shows how to use the **irecv()** function to do an asynchronous receive:

```
long iam;

main() {

   long msgid;
   char smsg[80], rmsg[80];

   iam = mynode();
   sprintf(smsg,"Hello from node %d\n",iam);
   msgid = irecv(100, rmsg, sizeof(rmsg));
   csend(100, smsg, strlen(smsg)+1 ,-1,0);

   msgwait(msgid);

   printf("%d: received: %s\n",iam,rmsg);
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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™ System C Calls Reference Manual* for information on message types.
buf	Points 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**()).

ISEND() *(cont.)*

ISEND() *(cont.)*

Return Values

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. The error message "Too many requests" is returned and your application will stop when no message IDs are available for a requested asynchronous send or receive. Your program should release its message IDs as soon as possible by calling **msgcancel()**, **msgdone()**, **msgignore()**, or **msgwait()**.

Errors

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

EQNOMID Your application has used all the available message IDs and no message IDs are available. Use either the **msgcancel()**, **msgdone()**, **msgignore()**, or **msgwait()** function with the receive to release message IDs.

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

ISEND() *(cont.)* ISEND() *(cont.)*

Examples

The following example shows how to use the **isend()** function to do an asynchronous send:

```
#include <nx.h>

#define INIT_TYPE 10

long iam;

main()
{
   long msgid;
   char msgbuf[80], smsg[80];

   iam = mynode();
    if(!iam) {
      sprintf(smsg,"Hello from node %d\n",iam);
      msgid = isend(INIT_TYPE, smsg, sizeof(smsg), 1, 0);
      printf("Node %d sent: %s",iam,smsg);
      msgwait(msgid);
       printf("Node %d send buffer available for
               writing\n",iam);
   }
   else {
      crecv(INIT_TYPE, msgbuf, sizeof(msgbuf));
      printf("Node %d received: %s\n",iam,msgbuf);
   }
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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™ System C Calls Reference Manual* for more information about message types.
sbuf	Points 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™ System C Calls Reference Manual* for more information about message type selectors.
rbuf	Points to the buffer in which to store the reply.
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. To determine if the message sent is received, do either 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.

You can use the **info...**() system calls to get more information about a message after it is received.

For synchronous message passing applications, use the **csendrecv**() function 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. The error message "Too many requests" is returned and your application will stop when no message IDs are available for a requested asynchronous send or receive. Your program should release its message IDs as soon as possible by calling **msgcancel()**, **msgdone()**, **msgignore()**, or **msgwait()**.

Errors

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

EQNOMID Your application has used all the available message IDs and no message IDs are available. Use either the **msgcancel**(), **msgdone**(), **msgignore**(), or **msgwait**() function with the receive to release message IDs.

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

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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. This function blocks until all asynchronous requests made by the process to the same file are processed.

Return Values

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

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

Otherwise, the **iseof()** function writes an error message on the standard error output 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.

ISEOF() *(cont.)* ISEOF() *(cont.)*

Errors

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

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

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

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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.

ISNAN() *(cont.)* ISNAN() *(cont.)*

Errors

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

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

errno, **fpgetround()**

IWRITE() IWRITE()

iwrite(), **iwritev()**: Writes to a file and returns immediately. (Asynchronous write)

Synopsis

```
#include <nx.h>

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


#include <sys/uio.h>

long iwritev(
    int fildes,
    struct iovec *iov,
    int iovcount );
```

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.

iov Pointer to an array of *iovec* structures, which identifies the buffers containing the data to be written. The *iovec* structure has the following form:

```
struct iovec {
     caddr_t iov_base;
     int     iov_len;
};
```

The *iovec* structure is defined in the *sys/uio.h* include file.

iovcount Number of *iovec* structures pointed to by the *iov* parameter.

Description

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

The **iwrite()** and **iwritev()** functions are asynchronous system calls. Asynchronous system calls return 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 either the **cwrite()** or **cwritev()** function (synchronous system calls) instead of the **iwrite()** or **iwritev()** function, respectively.
- Use **iowait()** to wait until the write completes.
- Loop until **iodone()** returns a 1, indicating that the write is complete.

NOTE

To preserve data integrity, all I/O requests are processed on a "first-in, first-out" basis. This means that if an asynchronous I/O call is followed by a synchronous I/O call on the same file, the synchronous call will block until the asynchronous operation has completed.

After an **iwrite()** or **iwritev()** call, you can perform other read or write calls on the same file 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()** system call.

Return Values

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

IWRITE() *(cont.)*

IWRITE() *(cont.)*

Upon successful completion, the **_iwrite()** and **_iwritev()** functions return a non-negative I/O ID. You can use this I/O ID with the **iodone()** and **iowait()** functions. Otherwise, the **_iwrite()** and **_iwritev()** functions return -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 returned I/O ID as soon as possible by calling **iodone()** or **iowait()**.

Errors

If the **_iwrite()** or **_iwritev()** 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.

Examples

The following example shows how to use the **iwrite()**, **iodone()**, and **iowait()** functions to do an asynchronous write:

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

long iam;

main()
{
   int fd, id;
   long mode;
   char buffer[80];
```

```
    iam = mynode();

    fd = gopen("/tmp/mydata",O_CREAT | O_TRUNC | O_RDWR,
              M_UNIX, 0644);

    mode = iomode(fd);
    if(!iam) printf("%d: iomode = %d\n",iam, mode);

    sprintf(buffer,"Hello from node %d\n",iam);
    id = iwrite(fd, buffer, strlen(buffer));
    if (iam){
       while (!iodone(id))
          printf("%d: write not done\n",iam);
       printf("%d: write done\n",iam);
    }
    else {
       printf("%d: write not done\n",iam);
       iowait(id);
       printf("%d: write done\n",iam);
    }

    close(fd);
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

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

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

IWRITEOFF() IWRITEOFF()

iwriteoff(), **iwritevoff**(): Asynchronous writes to a file at a specified offset.

Synopsis

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

long iwriteoff(
        int fildes,
        esize_t offset,
        char *buffer,
        unsigned int nbytes );


#include <sys/types.h>
#include <sys/uio.h>

long iwritevoff(
        int fildes,
        esize_t offset,
        struct iovec *iov,
        int iovcount );
```

Description of Parameters

fildes	A file descriptor identifying the file to which the data is to be written.
offset	Offset from the beginning of the file at which to begin the write.
buffer	Pointer to the buffer containing the data to be written.
nbytes	The number of bytes to write to the file associated with the *fildes* parameter.
iov	Pointer to an array of *iovec* structures that identify the buffers from which the data is to be written.
iovcount	The number of *iovec* structures pointed to by the *iov* parameter.

IWRITEOFF() *(cont.)*

IWRITEOFF() *(cont.)*

Discussion

iwriteoff() writes *nbytes* asynchronously to the file specified by the descriptor *fd* starting at the offset specified by *offset* from the buffer pointed to by *buffer*. **iwritevoff**() is similar, but it writes the data from the iovcount buffers specified by *iov*.

iwriteoff() and **iwritevoff**() are identical to **iwrite**() and **iwritev**() except for writing starting at a user-specified offset (instead of the offset maintained by the system file pointer) and the following additional differences:

- The current value of the system file pointer is not modifed.
- Currently only M_UNIX and M_ASYNC I/O modes are supported.
- Paragon PFS is the only file system type that currently supports these functions.
- The O_APPEND flag used in the open function to obtain the file descriptor has no effect on the write. The write is performed at the position specified by the *offset* parameter.

Return Values

Upon successful completion, a non-negative I/O ID for use in **iodone**(), **iowait**(), **niodone**() and **niowait**() calls is returned. If an error occurs, these functions return -1 and set *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()**, **iowait()**, **niodone()** or **niowait()**.

IWRITEOFF() *(cont.)*

IWRITEOFF() *(cont.)*

Errors

Errors are as described in OSF/1 **write()**, except that the following errors can also occur:

EMREQUEST An asynchronous 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.

EFSNOTSUPP The file referred to by *filedes* is not in a file system of a type that supports this operation. Currently only the PFS file systems support this operation.

EINVAL The file referred to by *filedes* is in an unsupported iomode. Currently only M_UNIX and M_ASYNC are supported.

See Also

cwrite(), gopen(), iodone(), iowait(), iseof(), iwrite(), niodone(), niowait(), setiomode(), writeoff()

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

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. The type **off_t** is defined in *sys/types.h* (included in *nx.h*).

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

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 increases the size of a file according to the *offset* and *whence* parameters.

Use the **lsize()** function to allocate 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. See the OSF/1 **truncate()** manual page for information about decreasing a file's size.

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

The **lsize()** function has no effect on FIFO special files or directories, and does not effect the position of the file pointer. The contents of file space allocated by the **lsize()** function is undefined.

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

The **lsize()** 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.

To increase the size of an extended file, use the **esize()** function.

Note

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

Note

Because NFS does not support disk block preallocation, the **lsize()** and **_lsize()** functions are not supported on files that reside in remote file systems that have been NFS mounted. The **lsize()** and **_lsize()** functions are supported on files in UFS and PFS file systems only.

Return Values

Upon successful completion, the **lsize()** function returns the new size of the file, in bytes. If the new size specified by the *offset* and *whence* parameters is greater than the available disk space, the **lsize()** function allocates what disk space is available and returns the new size of the file. 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.

LSIZE() *(cont.)*

LSIZE() *(cont.)*

Errors

If the **_lsize()** 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. The **lsize()** function does not support files that reside in remote file systems and have been NFS mounted.

EINVAL The file is not a regular file.

ENOSPC No space left on device.

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

Examples

The following example shows how to use the **lsize()** function to increase the size of a file with different *whence* values:

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

main()
{
   int     fd;
   off_t offset;
   long newsize, new_pos;
   esize_t loc, eoffset;

   fd = gopen("/tmp/mydata", O_RDWR, M_UNIX, 0644);
```

LSIZE() *(cont.)*

LSIZE() *(cont.)*

```
    offset  = 1000;
    newsize = lsize(fd,offset,SIZE_SET);
    printf("new_size = %d\n",newsize);

    eoffset = stoe("1000");
    loc = eseek(fd, eoffset, SEEK_END);

    newsize = lsize(fd,offset,SIZE_CUR);
    printf("new_size = %d\n",newsize);

    newsize = lsize(fd,offset,SIZE_END);
    printf("new_size = %d\n",newsize);

    close(fd);
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

eseek(), esize()

OSF/1 Programmer's Reference: **fcntl(2), lseek(2), open(2), truncate(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 handlers. This function protects critical code from being interrupted by the handler procedure that is executed when using the **h...()** calls (**hrecv()**, **hsend()**, or **hsendrecv()**). A **masktrap(1)** prevents any handler from running; a **masktrap(0)** enables handlers. Any pending interrupts are honored when the mask is removed. The **masktrap()** function returns the previous masking state (1 or 0).

CAUTION

When using any of the **h...()** calls, you must use **masktrap()** around any code in the main program that could interfere with calls in the handler.

For example, if the handler performs any I/O, you must put **masktrap()** calls around any I/O call in the main program that could be called while the handler is active. If you do not do this, you could find characters from the handler's output interleaved with characters from the main program's output.

MASKTRAP() *(cont.)* MASKTRAP() *(cont.)*

Sometimes it is not as obvious which calls could interfere with each other. For example, any two library calls that could allocate or free memory could cause the memory subsystem to become confused if they were called at the same time. To be safe, keep the handler simple and use the **masktrap**() function to protect *all* library calls following the **h...**() call that could call the same subsystems as the handler while the handler is active.

Calls to the **masktrap**() function are necessary, because a handler and the main program share the same memory space and can change each other's global variables. This could cause any *non-reentrant* function to fail if it is called by both the handler and the main program at the same time.

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 errors that can occur in the C underscore system calls.

Examples

The following example runs on two nodes and shows how to use the **masktrap**() function to with the **hrecv**() function. After posting an **hrecv**(), the application must not be interrupted until the receive handler completes A **masktrap**() call with *state* parameter value set to 1 prevents the handler from executing. A **masktrap**() call with *state* parameter value set to 0 (zero) allows the handler to execute immediately.

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

void proc();
long iam;

main() {

   char buf[80];
   long mask;
```

MASKTRAP() *(cont.)* MASKTRAP() *(cont.)*

```
    iam = mynode();
    memset(buf,0,80);

    if(iam == 0) {
       printf("\n%d: Before hrecv\n", iam);
       hrecv(100,buf,sizeof(buf),proc);
       mask = masktrap(1);
       printf("%d: After hrecv\n", iam);
       printf("%d Waiting ...\n",iam);
       printf("%d: No hrecv interrupts can occur\n",iam);
       mask = masktrap(mask);
       sleep(5);
       printf("%d: Until now ....\n",iam);
       printf("%d Completed \n",iam);
    }
    else {
       sleep(1);
       sprintf(buf,"Hello from node %d\n",iam);
       printf("Node 1 sends to node 0\n");
       csend(100,buf,strlen(buf),0,0);
    }
}
void proc(type,count,node,pid)
long type, count, node, pid;
{
    printf("%d Entered handler:\n", iam);
    printf("%d type = %d\n",iam, type);
    printf("%d count = %d\n",iam, count);
    printf("%d node = %d\n",iam, node);
    printf("%d pid = %d\n",iam, pid);
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

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

MOUNT() MOUNT()

mount(), **umount()**: Mount or unmount a file system.

Synopsis

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

void mount(
    int type,
    char *mnt_path,
    int mnt_flag,
    caddr_t data );

void umount(
    char *mnt_path,
    int umnt_flag );
```

Parameters

type Defines the type of the file system. The types of file systems are MOUNT_UFS and MOUNT_PFS.

mnt_path Points to a null-terminated string containing the appropriate pathname.

mnt_flag Specifies whether certain semantics should be suppressed when accessing the file system. Valid flags are:

M_RDONLY The file system should be treated as read-only; no writing is allowed (even by a process with appropriate privilege). Physically write-protected and magnetic tape file systems must be mounted read-only or errors will occur when access times are updated, whether or not any explicit write is attempted.

M_NOEXEC Do not allow files to be executed from the file system.

M_NOSUID Do not honor setuid or setgid bits on files when executing them.

MOUNT() *(cont.)*

MOUNT() *(cont.)*

M_NODEV Do not interpret special files on the file system.

M_SYNCHRONOUS
All I/O to the file system should be done synchronously.

M_FMOUNT Forcibly mount, even if the file system is unclean.

M_UPDATE The mount command is being applied to an already mounted file system. This allows the mount flags to be changed without requiring that the file system be umounted and remounted.

M_PFS_SERVER_BUFFERING
Enable PFS server buffering. The fileservers cache stripe-file data in their memory-resident, disk-block caches. These fileservers use a read-ahead and write-behind caching algorithm. PFS buffering is recommended only when the IO request size is less than 64K bytes; otherwise, the fieservers's cache may thrash.

Some file systems may not allow all flags to be changed. For example, most file systems do not allow a change from read-write to read-only.

data Points to a structure that contains the type-specific parameters to mount.

umnt_flag Specifies one of the following values:

MNT_FORCE The file system should be forcibly umounted even if files are still active. Active special devices continue to work, but any further accesses to any other active files result in errors even if the file system is later remounted. Support for forcible unmount is filesystem dependent.

MOUNT() *(cont.)* MOUNT() *(cont.)*

Description

Except in the case of file-on-file mounting, the **mount()** function mounts a file system on the directory pointed to by the *mnt_path* parameter. Following the mount, references to *mnt_path* refer to the root of the newly mounted file system.

The *mnt_path* parameter must point to a directory or file that already exists.

For file-on-file mounting the **mount()** function mounts a file specified by the *data* parameter onto another file specified by the *mnt_path* parameter. Either file may be of any type, but *mnt_path* cannot already have a file system or another file mounted on it.

The **umount()** function unmounts a file system mounted at the directory pointed to by the *mnt_path* parameter. The associated directory reverts to its ordinary interpretation.

Notes

For file-on-file mounting the *data* argument points to a *ffs_args* structure containing flags and the file to be mounted. In *ffs_flags* if FFS_FD is true, then the file is specified by the file descriptor, *ffs_filedesc*, otherwise by the pathname **ffs_pathname*. If FFS_CLONE is true, then new mount point should exhibit CLONE behavior; specifically, calls such as **chmod()** and **chown()** should have no effect on the mounted file. (The original file is, of course, always unaffected, since the mount point hides it.) If the file descriptor refers to a pipe, a call to **stat()** will return the number of unread bytes in the *st_size* field.

If file systems other than FFS (such as UFS or NFS) are modified to permit mounts by unprivileged users, it may be appropriate to ensure that the M_NODEV flag is set in the mount structure that is created, so that users cannot obtain undeserved access through devices.

An additional argument structure, *pfs_args*, has been added to the *mount.h* header file to support mounting a parallel file system (PFS).

MOUNT() *(cont.)* MOUNT() *(cont.)*

Return Values

The **mount()** function returns 0 (zero) if the file system was successfully mounted. Otherwise, -1 is returned. The mount can fail if the *mnt-path* parameter does not exist or is of the wrong type. For a UFS file system, the mount can fail if the special device specified in the **ufs_args** structure is inaccessible, is not an appropriate file, or is already mounted. A mount can also fail if there are already too many file systems mounted, either system wide, or for a specific file system type.

The **umount()** function returns 0 (zero) if the file system was successfully unmounted. Otherwise, -1 is returned. The unmount will fail if there are active files in the mounted file system, unless the MNT_FORCE flag is set and the file system supports forcible unmounting.

Errors

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

EPERM The caller does not have appropriate privilege.

ENAMETOOLONG
A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX characters.

ELOOP Too many symbolic links were encountered in translating a pathname.

ENOENT A component of the *mnt-path* parameter does not exist.

ENOTDIR A component of the *name* parameter is not a directory, or a path prefix of the *special* parameter is not a directory.

EINVAL A pathname contains a character with the high-order bit set.

EBUSY Another process currently holds a reference to the *mnt-path* parameter.

EDIRTY The file system is not clean and M_FORCE is not set.

EFAULT The *mnt-path* parameter points outside the process' allocated address space.

The following errors can occur for a UFS file system mount:

ENODEV A component of **ufs_args fspec** does not exist.

ENOTBLK The **fspec** field is not a block device.

MOUNT() *(cont.)* MOUNT() *(cont.)*

ENXIO The major device number of **fspec** is out of range (this indicates no device driver exists for the associated hardware).

EBUSY The device pointed to by the **fspec** field is already mounted.

EMFILE No space remains in the mount table.

EINVAL The super block for the file system had a bad magic number or an out of range block size.

ENOMEM Not enough memory was available to read the cylinder group information for the file system.

EIO An I/O error occurred while reading the super block or cylinder group information.

EFAULT The **fspec** field points outside the process' allocated address space.

The following errors can occur for a NFS compatible file system mount:

ETIMEDOUT NFS timed out trying to contact the server.

EFAULT Some part of the information described by **nfs_args** points outside the process' allocated address space.

The following errors can occur for a PFS compatible file system mount:

ENODEV A component of the *pfs_args fspec* field does not exist

ENOTBLK The *fspec* field is not a block device.

ENXIO The major device number of *fspec* is out of range (this indicates no device driver exists for the associated hardware).

EBUSY The device pointed to by the *fspec* field is already mounted.

EMFILE No space remains in the mount table.

EINVAL The super block for the file system had a bad magic number or an out of range block size.

ENOMEM Not enough memory was available to read the cylinder group information for the file system.

MOUNT() *(cont.)*

MOUNT() *(cont.)*

EIO An I/O error occurred while reading the super block or cylinder group information.

EFAULT Some part of the information described by *pfs_args* points outside the process's allocated address space.

EINVAL The value specified by the *stripe_unit_size* field of the *pfs_args* structure is invalid; for example, the value is not positive or is greater than the maximum file size supported by the file system.

ENOTDIR A path name specified in the *stripe_dir* field of the *pfs_args* structure does not refer to a directory.

ENOENT A path name specified in the *stripe_dir* field of the *pfs_args* structure does not exist.

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

EPERM The caller does not have appropriate privilege.

ENOTDIR A component of the path is not a directory.

EINVAL The pathname contains a character with the high-order bit set.

ENAMETOOLONG
A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX characters.

ELOOP Too many symbolic links were encountered in translating the pathname.

EINVAL The requested directory is not in the mount table.

EBUSY A process is holding a reference to a file located on the file system.

EIO An I/O error occurred while writing cached file system information.

EFAULT The *mnt-path* parameter points outside the process' allocated address space.

MOUNT() *(cont.)*

MOUNT() *(cont.)*

See Also

files: **fstab(4)**, **pfstab(4)**

Calls: **getpfsinfo(3)**, **getmntinfo(3)**, **statfs(2)**, **statpfs(3)**

Commands: **mount(8)**

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()**.

MSGCANCEL() *(cont.)* MSGCANCEL() *(cont.)*

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 errors that can occur in the C underscore system calls.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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.)*

If the *mid* parameter in the **msgdone()** function represents a merged message ID (that is, it was returned by the **msgmerge()** function), the information returned for the **info...()** calls is unpredictable.

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 errors that can occur in the C underscore system calls.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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()**.

MSGIGNORE() *(cont.)*

MSGIGNORE() *(cont.)*

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 errors that can occur in the C underscore system calls.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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.

MSGMERGE() *(cont.)* MSGMERGE() *(cont.)*

Return Values

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 list of errors that can occur in the C underscore system calls.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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()**.

MSGWAIT() *(cont.)* MSGWAIT() *(cont.)*

If the *mid* parameter in the **msgwait()** function represents a merged of message ID (that is, it was returned by the **msgmerge()** function), the information returned for the **info...()** calls is unpredictable.

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 errors that can occur in the C underscore system calls.

Examples

The following example shows how to use the **msgwait()** function to wait until an asynchronous receive completes:

```
#include <nx.h>

long iam;

main() {
   long msgid;
   char smsg[80], rmsg[80];

   iam = mynode();
   sprintf(smsg,"Hello from node %d\n",iam);
   msgid = irecv(100, rmsg, sizeof(rmsg));
   csend(100, smsg,     strlen(smsg)+1 ,-1,0);

   msgwait(msgid);

   printf("%d: received: %s\n",iam,rmsg);
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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 errors that can occur in the C underscore system calls.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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 Values

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 the node number 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 errors that can occur in the C underscore system calls.

MYNODE() *(cont.)* MYNODE() *(cont.)*

Examples

The following example shows how to use the **mynode()** function to get the node number of the calling process and use the node number in an application:

```
long iam;

main()
{
   long node, type, ptype, count;
   char rmsg[80],smsg[80];

   iam = mynode();

   if(!iam) {
      sprintf(smsg,"Hello from node %d\n",iam);
      csend(100,smsg,strlen(smsg) + 1,1,0);
   }
   else {
      crecv(100,rmsg,sizeof(rmsg));
      node = infonode();
      type = infotype();
      ptype  = infoptype();
      count = infocount();
      printf("node = %d\n",node);
      printf("type = %d\n",type);
      printf("ptype = %d\n",ptype);
      printf("count = %d\n",count);
   }
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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 errors that can occur in the C underscore system calls.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

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

NIODONE() NIODONE()

Determine whether an asynchronous read or write operation is complete and return the number of bytes transferred if the operation is complete.

Synopsis

```
#include <nx.h>

long niodone(
        long id );
```

Description of Parameters

id	The non-negative I/O ID returned by **iread()** or **iwrite()**.

Discussion

Use **niodone()** to determine whether the asynchronous read or write operation (e.g., **iread(), ireadoff(), iwrite()** or **iwriteoff()**) identified by *id* is complete. If **niodone()** returns a non-negative number (indicating that the 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 number that identifies the asynchronous read or write *(id)* is released for use in a future asynchronous read or write.

NOTE

You must use one of **iodone()**, **iowait()**, **niodone()** or **niowait()** after an asynchronous read or write to ensure that the operation is complete and to release the I/O ID number.

NIODONE() *(cont.)* NIODONE() *(cont.)*

Return Values

Upon successful completion, **niodone()** returns

> 0	If the read or write is complete. The number represents the number of bytes transferred in the I/O.
-1	If the read or write is not complete. If the read or write is complete but unsuccessful, errno is set to the error.

Errors

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

See Also

iodone(), iowait(), iread(), ireadoff(), iwrite(), iwriteoff(), niowait()

NIOWAIT() NIOWAIT()

Wait (block) until an asynchronous read or write operation completes. Return the number of bytes transferred if the operation completed successfully.

Synopsis

```
#include <nx.h>

long niowait(
        long id );
```

Description of Parameters

id	The non-negative I/O ID returned by **niread()** or **niwrite()**.

Discussion

Use **niowait()** to cause a process to wait until the asynchronous read or write operation (e.g., **iread()** **ireadoff(), iwrite() or iwriteoff()**) identified by *id* completes. When **niowait()** 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 number that identifies the asynchronous read or write (*id*) is released for use in a future asynchronous read or write.

NOTE

You must use one of **iodone(), iowait(), niodone()** or **niowait()** after an asynchronous read or write to ensure that the operation is complete and to release the I/O ID number.

Return Values

Upon successful completion, **niowait()** simply returns the number of bytes transferred by the I/O operation. If an error occurs, **niowait()** sets errno to indicate the error and returns -1.

NIOWAIT() *(cont.)* NIOWAIT() *(cont.)*

Errors

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

See Also

iodone(), iowait(), iread(), ireadoff(), iwrite(), iwriteoff(), niodone()

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 errors that can occur in the C underscore system calls.

Examples

The following example shows how to use the **msgwait()** function to wait until an asynchronous receive completes:

```
#include <math.h>

#define M    4
#define N   16

void display();

long iam, nbrnodes;
```

NUMNODES() *(cont.)*

NUMNODES() *(cont.)*

```
main()
{
   int    i, count=0;
   double x[M], y[N], dot, norm, dummy;
   char   msg[80];
   int    dpsize = 8;
   long   xlen[4];

   iam      = mynode();
   nbrnodes = numnodes();
   dot      = 0.0;

   for(i=0; i<nbrnodes; i++)
      xlen[i] = 4*sizeof(double);

   for(i=0; i<M; i++) {
      x[i] = (double) (iam * M + i);
      printf("Node %d x[%d] = %3.1f\n",iam,i,x[i]);
   }

   for(i=0; i<M; i++)
      dot += x[i]*x[i];
   printf("Node %d dot = %f\n",iam,dot);

   gdsum(&dot, 1, &dummy);
   sprintf(msg,"dot = %f\n",dot);
   if(!iam) printf("\n%s",msg);

   norm = sqrt(dot);

   for(i=0; i<M; i++)
      x[i] = x[i]/norm;

   gcolx(x, xlen, y);

   if(!iam) {
      for(i=0;i<nbrnodes*M; i++)
         printf("%3.1f ",y[i]);
      printf("\n");
   }
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

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

NX_APP_NODES() NX_APP_NODES()

Returns the list of nodes allocated to an application.

Synopsis

```
#include <nx.h>

long nx_app_nodes(
    pid_t pgroup,
    nx_nodes_t *node_list,
    unsigned long *list_size);
```

Parameters

pgroup	Process group ID for the application, 0 (zero) to specify the calling application. The **pid_t** type is defined in the include file *sys/types.h*. If the process group ID is not that of the calling process, the calling process's group ID must either be root or the same user ID as the specified application.
node_list	Pointer variable that specifies the address of the list of nodes for the application. The node numbers are root-partition node numbers. The **nx_nodes_t** type is defined in the include file *allocsys.h*. The call allocates memory and fills in the values for this parameter. Free this memory using the **free()** function.
list_size	Address of a variable into which the **nx_app_nodes()** function stores the number of elements in the *node_list* parameter. The call fills in the value for this parameter.

Description

The **nx_app_nodes()** function returns the list of node numbers for the nodes an application is running on. You must have read permission on the partition the application is running in to use this call.

Return Values

On successful completion, the **nx_app_nodes()** function returns 0 (zero). Otherwise, -1 is returned and *errno* is set to indicate the error.

NX_APP_NODES() *(cont.)* NX_APP_NODES() *(cont.)*

Examples

The following example prints the list of nodes for an application:

```
#include <nx.h>

main() {

   nx_nodes_t     mynodes;
   unsigned long  nnodes;
   int            i, status;

   status = nx_app_nodes(0, &mynodes, &nnodes);

   if(status != 0) {
       nx_perror("nx_app_nodes()");
       exit(1);
   }

   for(i = 0; i < nnodes; i++) {
       printf("%d\n", mynodes[i]);
   }

   free(mynodes);
}
```

Note the use of the & operator in the call to **nx_app_nodes()**.

Errors

EANOEXIST The specified process group does not exist.

EPACCESS Insufficient access permission for this operation on the partition.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

mynode(), **nx_part_nodes()**, **nx_failed_nodes()**

NX_APP_RECT() NX_APP_RECT()

nx_app_rect(), mypart(): Returns the height and width of the rectangle of nodes allocated to the current application.

Synopsis

```
#include <nx.h>

long nx_app_rect(
    long *rows,
    long *cols);

long mypart(
    long *rows,
    long *cols);
```

Parameters

rows Address of a long integer variable that specifies the number of rows in the set of nodes for the application. If the set of nodes is not a rectangle, the value pointed to by *rows* is set to 1.

cols Address of a long integer variable that specifies the number of columns in the set of nodes for the application. If the node set is not a rectangle, the value pointed to by *cols* is set to the number of nodes in the application.

Description

The **nx_app_rect()** function returns the rectangular dimensions of the node set of the application from which the function call is made.

The **mypart()** function is identical to the **nx_app_rect()** function and is provided for compatibility with the Touchstone DELTA system.

Return Values

On successful completion, the **nx_app_rect()** function returns 0 (zero). Otherwise, -1 is returned and *errno* is set to indicate the error.

NX_APP_RECT() *(cont.)* NX_APP_RECT() *(cont.)*

Errors

Refer to the ***errno*** manual page for a list of errors that can occur in this system call.

Examples

This example returns the number of rows and columns used by the application. Note the use of "&rows" and "&cols" indicating that these variables must have space allocated prior to passing the pointers to **nx_add_rect()**.

```
main() {

   long           rows, cols, result;
   int            status;

   if (mynode() == 0) {
      status = nx_app_rect(&rows, &cols);

      if(status != 0) {
        nx_perror("nx_app_rect()");
        exit(1);
      }

      printf("\n");
      printf("\nNumber of rows = %d", rows);
      printf("\nNumber of columns = %d", cols);
      printf("\n");
   }
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

application, **mkpart**, **nx_app_nodes()**, **nx_initve_rect()**, **nx_mkpart()**, **nx_part_attr()**, **nx_root_nodes()**

NX_CHPART_EPL() NX_CHPART_EPL()

nx_chpart_epl(), nx_chpart_mod(), nx_chpart_name(), nx_chpart_owner(), nx_chpart_rq(), nx_chpart_sched(): 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 );

long nx_chpart_sched(
    char *partition,
    long sched_type );
```

NX_CHPART_EPL() *(cont.)* NX_CHPART_EPL() *(cont.)*

Parameters

partition Pointer to the relative or absolute pathname of an existing partition for which you are changing the characteristics.

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

The calling process must have write permission for the partition to use the **nx_chpart_epl()** function.

mode (**nx_chpart_mod()** only)
New protection modes for the partition, expressed as an octal number. See the **chmod()** function in the *OSF/1 Programmer's Reference* for more information on specifying protection modes.

The calling process must be the owner of the partition or *root* user to use the **nx_chpart_mod()** function.

name (**nx_chpart_name()** only)
New name for the partition, expressed as a string of any length containing only uppercase letters, lowercase letters, digits, and underscores. The **nx_chpart_name()** function can only change the partition's name "in place;" there is no way to move a partition to a different parent partition.

The calling process must have write permission on the parent partition of the specified partition to use the **nx_chpart_name()** function.

owner (**nx_chpart_owner()** only)
New owner for the partition, expressed as a numeric user ID (UID). If the *owner* parameter is -1, the partition's owner 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.

The permissions required for the **nx_chpart_owner()** function depend on the operation. To change the partition's ownership, the calling process must be the system administrator.

group (**nx_chpart_owner**() only)
New group for the partition, expressed as a numeric group ID (GID). If the *group* parameter is -1, the group 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.

The permissions required for the **nx_chpart_owner()** function depend on the operation. To change the partition's group, the calling process must either be the system administrator or must be the partition's owner and changing the group to a group that the calling process belongs to.

rollin_quantum (**nx_chpart_rq**() only)
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.

The minimum rollin quantum can be set in the *allocator.config* file. See the **allocator.config** manual page for more information.

The calling process must have write permission for the partition to use the **nx_chpart_rq**() function.

sched_type (**nx_chpart_sched**() only)
Type of scheduling for the partition. These scheduling types are defined in the *nx.h* include file and can be specified:

NX_GANG Gang scheduling (rollin quantum = 0).

NX_SPS Space sharing.

The calling process must have write permission for the partition to use the **nx_chpart_sched**() function.

Description

The following functions change specific characteristics of a partition:

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

nx_chpart_mod()
Changes the partition's protection modes.

NX_CHPART_EPL() *(cont.)*

NX_CHPART_EPL() *(cont.)*

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_sched()
Changes the partition's scheduling type.

When you create a partition with the **mkpart** command or the **nx_mkpart...()** functions, you set a partition's initial characteristics. You can set specific characteristics or use the default characteristics. After creating a partition, you are the partition's owner and you can use the **nx_chpart...()** functions or the **chpart** command to change the partition's characteristics.

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

The **nx_chpart_name()** function changes the partition's name. 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. The owner and group must be specified as a numeric ID, not as a name. Use the OSF/1 **getpwnam()** function to convert an owner name to a user ID, and 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 modes consist 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 modes are 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.

NX_CHPART_EPL() *(cont.)* NX_CHPART_EPL() *(cont.)*

The **nx_chpart_sched()** function changes the partition's scheduling to either space sharing (**NX_SPS**) or gang scheduling (**NX_GANG**). The **nx_chpart_sched()** function has the following restrictions:

- You cannot change a partition's scheduling to or from standard scheduling.
- You cannot change a partition's scheduling to space sharing if the partition contains any active applications or overlapping partitions.

The allocator may limit the number of partitions that can use gang scheduling. For information on the allocator, see the **allocator** manual page in the *Paragon™ XP/S System Commands Reference Manual*. You cannot change a partition's scheduling to gang scheduling if the request exceeds the maximum number of partitions allocated for gang scheduling. The rollin quantum is automatically set to 0 (zero) when changing to gang-scheduling.

Return Values

On successful completion, the partition's characteristic was successfully changed and 0 (zero) is returned. Otherwise, the partition's characteristic is not changed, -1 is returned, and *errno* is set to indicate the error.

Errors

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

EEXCEEDCONF
The request would exceed the configuration parameters.

EPACCES The application has insufficient access permission on a partition.

EPALLOCERR
An internal error occurred in the node allocation server.

EPINGRP An invalid group ID was specified.

EPINRN You specified a partition name that was not a simple name. You cannot change a partition's relationship within a partition hierarchy.

EPINUSER An invalid user ID was specified.

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

EPINVALPRI An invalid priority level was specified.

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

EPPARTEXIST
The specified partition already exists.

ESCHEDCONF
The scheduling parameters conflict with the allocator configuration.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

Paragon™ System C Calls Reference Manual: **nx_mkpart()**, **nx_pspart()**, **nx_rmpart()**

Paragon™ XP/S System Commands Reference Manual: **allocator**, **allocator.config**, **chpart**, **lspart**, **mkpart**, **pspart**, **rmpart**

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

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

NX_EMPTY_NODES() NX_EMPTY_NODES()

Returns the list of empty nodes in the root partition.

Synopsis

```
#include <nx.h>

int nx_empty_nodes(
    nx_nodes_t *node_list,
    unsigned long *list_size);
```

Parameters

node_list Pointer variable into which the **nx_empty_nodes()** function stores the address of the list of empty nodes found in the root partition. The node numbers are root-partition node numbers. The **nx_nodes_t** type is defined in the include file *allocsys.h*, which is included by the include file *nx.h*. The call allocates memory for this parameter. Free this memory using the **free()** function.

list_size Address to a variable into which the **nx_empty_nodes()** function stores the number of elements in the *node_list* array.

Description

The **nx_empty_nodes()** function returns the list of empty nodes in the root partition. An empty node is a node in the root partition that does not have a node board in the corresponding slot. An empty node is specified as "empty" in the *SYSCONFIG.TXT* file. An empty node shows up as a dash (-) in the display of the **showpart** command.

NOTE

Do not call the **nx_empty_nodes()** function on more than a few nodes at once.

If many nodes use the **nx_empty_nodes()** function at the same time, the node allocator daemon can become overwhelmed with requests. If all the nodes in your application need this information, you should have one node make the call and then distribute the information to the other nodes.

NX_EMPTY_NODES() *(cont.)* NX_EMPTY_NODES() *(cont.)*

Return Values

On successful completion, the **nx_empty_nodes()** function returns 0 (zero). Otherwise, -1 is returned and *errno* is set to indicate the error.

Examples

The following example prints the list of the empty nodes in the root partition:

```
#include <nx.h>
main() {

    nx_nodes_t      empty;
    unsigned long   nempty;
    int             i, status;

    status = nx_empty_nodes(&empty, &nempty);

    if(status != 0) {
        nx_perror("nx_empty_nodes()");
        exit(1);
    }

    for(i = 0; i < nempty; i++) {
        printf("%d\n", empty[i]);
    }

    free(empty);
}
```

Note the use of the & operator in the call to **nx_empty_nodes()**.

Errors

Refer to the ***errno*** manual page for a list of errors that can occur in this system call.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

Paragon™ System C Calls Reference Manual: **nx_app_nodes()**, **nx_failed_nodes()**

Paragon™ XP/S System Commands Reference Manual: **showpart**

NX_FAILED_NODES() NX_FAILED_NODES()

Returns a list of the failed nodes in the root partition.

Synopsis

```
#include <nx.h>

int nx_failed_nodes(
    nx_nodes_t *node_list,
    unsigned long *list_size);
```

Parameters

node_list Pointer variable into which the **nx_failed_nodes()** function stores the address of the list of failed nodes found in the root partition. The node numbers are root-partition node numbers. The **nx_nodes_t** type is defined in the include file *allocsys.h*, which is included by the include file *nx.h*. The call allocates memory for this parameter. Free this memory using the **free()** function.

list_size Address to a variable into which the **nx_failed_nodes()** function stores the number of elements in the *node_list* array.

Description

The **nx_failed_nodes()** function returns the list of failed nodes in the root partition. The system boots the nodes that are listed in the *SYSCONFIG.TXT* file on the diagnostic station. If a node fails to boot, it is listed as a bad or failed node. A failed node shows up as an **X** in the display of the **showpart** command.

NOTE

Do not call the **nx_failed_nodes()** function on more than a few nodes at once.

If many nodes use the **nx_failed_nodes()** function at the same time, the node allocator daemon can become overwhelmed with requests. If all the nodes in your application need this information, you should have one node make the call and then distribute the information to the other nodes.

NX_FAILED_NODES() *(cont.)*

NX_FAILED_NODES() *(cont.)*

Return Values

On successful completion, the **nx_failed_nodes()** function returns 0 (zero). Otherwise, -1 is returned and *errno* is set to indicate the error.

Examples

The following example prints the list of the failed nodes in the root partition:

```
#include <nx.h>
main() {
   nx_nodes_t      failed;
   unsigned long   nfailed;
   int             i, status;

   status = nx_failed_nodes(&failed, &nfailed);

   if(status != 0) {
       nx_perror("nx_failed_nodes()");
       exit(1);
   }

   for(i = 0; i < nfailed; i++) {
       printf("%d\n", failed[i]);
   }

   free(failed);
}
```

Note the use of the & operator in the call to **nx_failed_nodes()**.

Errors

Refer to the ***errno*** manual page for a list of errors that can occur in this system call.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

NX_FAILED_NODES() *(cont.)*

NX_FAILED_NODES() *(cont.)*

See Also

calls: **mynode()**, **nx_app_nodes()**, **nx_empty_nodes()**

commands: **allocator**, **showpart**

NX_INITVE() NX_INITVE()

nx_initve(), **nx_initve_rect()**: Initializes a parallel application on a partition.

Synopsis

```
#include <nx.h>

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

long nx_initve_rect(
    char *partition,
    long anchor_node,
    long rows,
    long cols,
    char *account,
    int *argc,
    char *argv[]);
```

Parameters

partition Relative or absolute pathname of the partition in which to run the application. A null string ("" or **NULL**) specifies using the default partition. The default partition is the partition specified by the *NX_DFLT_PART* environment variable, or is the *.compute* partition if the *NX_DFLT_PART* environment variable is not set. The specified partition must exist and must give execute permission to the calling process.

If the **-pn** switch is specified on the command line, the specified partition pathname overrides the *partition* parameter, unless you set the value of *argc* to 0 (zero).

size Number of nodes to run the application on. A value of 0 (zero) species the default size. The default size is the size specified by the *NX_DFLT_SIZE* environment variable, or all nodes of the partition if the *NX_DFLT_SIZE* environment variable is not set. The *size* parameter must be a non-negative integer.

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

If the **-sz** or **-nd** switch is specified on the command line, it overrides the value of the *size* parameter, unless you set the value of *argc* to 0 (zero).

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 *argc* value is 0 (zero), the command line and all command line arguments are ignored. When **nx_initve()** and **nx_initve_rect()** return, *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*.

anchor_node Node number of the node in the upper left-hand corner of the partition's rectangle. If the node number is -1, the allocator chooses the partition placement. For node numbers greater than or equal to 0 (zero), the partition is anchored on that node.

rows Number of rows in the partition's rectangle.

cols Number of columns in the partition's rectangle.

Description

The **nx_initve()** and **nx_initve_rect()** functions initialize an application to run in a specified partition. These functions create a new, empty application. The process that calls the **nx_initve()** or **nx_initve_rect()** function becomes the new application's *controlling process*. Use the **nx_initve()** and **nx_initve_rect()** functions as follows in an application:

- Call either function before any other Paragon system calls.
- Call either function only once.
- Use the **-lnx** switch to link a program that calls either function. Do not use the **-nx** option.

The **nx_initve()** and **nx_initve_rect()** functions just initialize a program. Use the **nx_loadve()**, **nx_load()**, or **nx_nfork()** calls to start a program's processes.

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

The **nx_initve()** function initializes an application to run in a specified number of nodes. Other than specifying a size, you cannot control how the nodes for your application are allocated. The **nx_initve()** function attempts to allocate a square group of nodes if it can. If this is not possible, the **nx_initve()** function attempts to allocate a rectangular group of nodes that is either twice as wide as it is high or twice as high as it is wide. If this is not possible, the **nx_initve()** function allocates any available nodes. In this case, nodes allocated to the application may not be contiguous (that is, they may not all be physically next to each other).

The **nx_initve_rect()** function initializes an application to run in a specified set of nodes allocated as a rectangle. You can specify the size and shape of the partition using the *rows* and *cols* parameters. You can specify the placement of the application within its partition using the *anchor_node* parameter. If you specify *anchor_node* to be -1, the allocator places the application wherever it fits. The **nx_initve_rect()** function fails if the specified rectangle cannot be allocated, even if the equivalent number of nodes are available in a non-rectangular shape.

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

The **nx_initve()** and **nx_initve_rect()** functions do not recognize the command line arguments **-pt**, **-on**, and **\;** *application*. If you want your application to have the same interface as an application linked with the **-nx** switch, you must parse the argument list for these arguments and pass the appropriate values to the **nx_load()** or **nx_loadve()** function.

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()** and **nx_initve_rect()** functions, the calling process becomes the controlling process of the application. If the calling process is not already the process group leader, the **nx_initve()** and **nx_initve_rect()** functions disassociate the calling process from its current process group, create a new process group, and make the calling process the process group leader of the new process group.

The **nx_initve()** and **nx_initve_rect()** functions do 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.

EAINVALMBF
The memory buffer size is invalid or out of range.

EAINVALMEA
The memory each size is invalid or out of range.

EAINVALMEX
The memory export size is invalid or out of range.

EAINVALPKT
The packet size is invalid or out of range.

EAINVALSTH
The send threshold size is invalid or out of range.

EAINVALGTH
The give threshold size is invalid or out of range.

EAOVLP A partition or application overlaps with another partition or application.

EAREJPLK An application cannot use the **-plk** switch in a gang-scheduled partition.

EINCOMPAT Your application's code is no longer up to date with the current release of the installed operating system. You must relink your application.

EPALLOCERR
An internal error occurred in the node allocation server.

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

EPBADNODE A bad node was specified.

EPINVALPRI An invalid priority value was specified.

EPINVALPART
The specified partition was not found.

EPXRS The request exceeds the partition resources.

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

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

Paragon™ System C Calls Reference Manual: **nx_app_rect()**, **nx_load()**, **nx_nfork()**

Paragon™ XP/S System Commands Reference Manual: **allocator**, ***application***

NX_INITVE_ATTR() NX_INITVE_ATTR()

Initializes a new application with specified attributes.

Synopsis

```
#include <nx.h>

long nx_initve_attr(
    char *partition,
    int *argc,
    char *argv[],
    [ int attribute, {long | char* | long *} value] ...
    NX_ATTR_END);
```

Parameters

partition Relative or absolute pathname of the partition in which to run the application. A null string ("" or **NULL**) specifies the default partition. The default partition is the partition specified by the *NX_DFLT_PART* environment variable, or is the *.compute* partition if the *NX_DFLT_PART* environment variable is not set. The specified partition must exist and must give execute permission to the calling process.

If you use the **-pn** switch on the command line, the specified partition pathname overrides the *partition* parameter (unless the value of *argc* is zero).

argc Pointer to an integer that is the number of arguments on the command line (including the application name). If the *argc* value is zero, the command line and all command-line arguments are ignored. When **nx_initve_attr()** 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*.

attribute Attribute constant to use for creating the new partition. The *attribute* parameter must be followed by the *value* parameter. The *value* parameter sets the value of the attribute. See the "Attributes" section for the list of attribute constants you can use with the *attribute* parameter.

value Value of the attribute specified by the *attribute* parameter. A *value* parameter must follow each *attribute* parameter. The data type of the *value* parameter depends on the preceding *attribute* parameter. See the "Attributes" section for a description of values.

NX_ATTR_END
Constant that marks the end of the list of *attribute*, *value* pairs.

Description

The **nx_initve_attr()** function initializes an application to run in a specific partition. The **nx_initve_attr()** function has the functionality of the **nx_initve()** and **nx_initve_rect()** functions, but you use attributes to specify how to initialize the application.

You specify attributes in the argument list of the function as a set of zero or more *attribute, value* pairs: an attribute constant and a value. The attribute constant is the name of the attribute. The attribute value can be either an integer, array of integers, or a character string depending on the attribute. You use the *attribute* parameter to specify the attribute constant and the *value* parameter to specify the value of the attribute. See the "Attributes" section for the list of the attributes that can be set in the **nx_initve_attr()** function.

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

The **nx_initve_attr()** function does not recognize the command line arguments **-pt**, **-on**, and **\;** *application*. If you want your application to have the same interface as an application linked with the **-nx** switch, you must parse the argument list for these arguments and pass the appropriate values to the **nx_load()** or **nx_loadve()** function.

When calling the **nx_initve_attr()** function, the calling process becomes the controlling process of the application. If the calling process is not already the process group leader, the **nx_initve_attr()** 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 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.

NX_INITVE_ATTR() *(cont.)*

NX_INITVE_ATTR() *(cont.)*

Attributes

The *attribute* parameter can be set with the following attribute constants:

Attribute Constant	Description
NX_ATTR_ANCHOR	Specifies the node number of the node in the upper left-hand corner of the partition rectangle. The *value* parameter must be of type **long**. You may only specify **NX_ATTR_ANCHOR** when **NX_ATTR_RECT** is present. If the *value* parameter is -1, the system chooses the partition placement. For node numbers greater than or equal to zero, the partition is anchored on that node.
NX_ATTR_GTH	Specifies the threshold for the "give me more messages" message in bytes. The *value* parameter must be of type **long**. If you use the **-gth** *give_threshold* switch from the command line and *argc* is not zero (i.e. it is in the *argc/argv* list), it overrides the value of the **NX_ATTR_GTH** value.
NX_ATTR_MBF	Specifies the total amount of memory allocated to message buffers in bytes. The *value* parameter must be of type **long**. If you use the **-mbf** *memory_buffer* switch from the command line and *argc* is not zero, it overrides the value of the **NX_ATTR_MBF** value.
NX_ATTR_MEA	Specifies the amount of memory allocated to buffering messages from each other node in bytes. The *value* parameter must be of type **long**. If you use the **-mea** *memory_each* switch from the command line and *argc* is not zero, it overrides the value of the **NX_ATTR_MEA** value.

NX_INITVE_ATTR() *(cont.)*

NX_INITVE_ATTR() *(cont.)*

Attribute Constant	Description
NX_ATTR_MEX	Specifies the total amount of memory allocated to buffering messages from other nodes in bytes. The *value* parameter must be of type **long**. If you use the **-mex** *memory_export* switch from the command line and *argc* is not zero, it overrides the value of the **NX_ATTR_MEX** value.
NX_ATTR_NOC	Specifies the total number of other processes from which each process expects to receive messages. The *value* parameter must be of type **long**. The default *value* is the number of nodes allocated for the application. If you use the **-noc** *correspondents* switch from the command line and *argc* is not zero, it overrides the value of the **NX_ATTR_NOC** value.
NX_ATTR_PKT	Specifies the size of each message packet in bytes. The *value* parameter must be of type **long**. If you use the **-pkt** *packet_size* switch from the command line and *argc* is not zero, it overrides the value of the **NX_ATTR_PKT** value.
NX_ATTR_PLK	Specifies whether to lock the data area of each process into memory. The *value* parameter must be of type **long**. The value 1 locks the data area of each process into memory, while the value 0 (zero) does not. This attribute is the same as **-plk** in *argv* list. The existing interaction between **-plk** and **REJECT_PLK** is preserved.
NX_ATTR_PRI	Specifies the priority at which the application runs. The *value* parameter must be of type **long**. If you use the **-pri** *priority* switch from the command line and *argc* is not zero, it overrides the value of the **NX_ATTR_PRI** value.

NX_INITVE_ATTR() *(cont.)*

NX_INITVE_ATTR() *(cont.)*

Attribute Constant	Description
NX_ATTR_RECT	Specifies running the application on a rectangular node set. The *value* parameter must be of type **long ***. The *value* parameter is a pointer to an array of two integers; the first integer is the height of the rectangle, while the second is its width. If you specify **NX_ATTR_SEL**, all the nodes in the rectangle must be consistent with the selected attributes. If you use either a **-sz** or a **-nd** switch from the command line and *argc* is not zero, it overrides the value of the **NX_ATTR_RECT** value.
NX_ATTR_RELAXED	Specifies whether to relax the requirement that all nodes requested must be available and eligible for allocation. The *value* parameter must be of type **long**. The value 0 does not relax the requirement, while the value 1 relaxes the requirement. If you specify a value of 1 and also use **NX_ATTR_RECT** and **NX_ATTR_RECT**, the requirement that all requested nodes must be allocated for the application is relaxed.
NX_ATTR_SCT	Specifies the number of bytes to send right away when the available memory is above *send_threshold*. The *value* parameter must be of type **long**. If you use the **-sct** *send_count* switch from the command line and *argc* is not zero, it overrides the value of the **NX_ATTR_SCT** value.
NX_ATTR_STH	Specifies the send threshold for sending multiple packets. The *value* parameter must be of type **long**. If you use the **-sth** *send_threshold* switch from the command line and *argc* is not zero, it overrides the value of the **NX_ATTR_STH** value.

NX_INITVE_ATTR() *(cont.)*

NX_INITVE_ATTR() *(cont.)*

Attribute Constant	Description
NX_ATTR_SZ	Specifies the size of the application (number of nodes to run the application on).The *value* parameter must be of type **long**. The default for *value* is 0 (zero). A *value* of 0 (zero) or -1 specifies using the default size set by the *NX_DFLT_SIZE* environment variable, or when *NX_DFLT_SIZE* is not set, is all nodes of the partition. If you use either a **-sz** or a **-nd** switch from the command line and *argc* is not zero, it overrides the value of the **NX_ATTR_SZ** value. Nodes are selected using the criteria specified by the **NX_ATTR_SEL** attribute, if any. If the value of the **NX_ATTR_RELAXED** attribute is specified as 1, fewer nodes than the requested number may be allocated and the application will run.
NX_ATTR_SEL	Specifies a pointer to a node attribute string. The *value* parameter must be of type **char ***. If you specify multiple **NX_ATTR_SEL** attributes, the result is the logical AND of all of them. Node attribute strings are case-insensitive. If you use the **-nt** *node_type* switch from the command line and *argc* is not zero, it overrides the values of both the **NX_ATTR_SEL** and **NX_MKPART_ATTR_EXCL** values.

NX_ATTR_SEL Values

The following shows the format of the *value* parameter for the **NX_ATTR_SEL** attribute.

node_attribute	Selects nodes having the specified attribute. For example, when *node_attribute* equals the string **mp**, only MP nodes are selected. The standard node attributes are shown in the "Node Attributes" section.

NX_INITVE_ATTR() *(cont.)*

NX_INITVE_ATTR() *(cont.)*

!*node_attribute*	Selects nodes *not* having the specified attribute. For example, when *node_attribute* equals the string **!io**, only nodes that are *not* I/O nodes are selected. Note that no white space may appear between the **!** and *node_attribute*.
[*relop*][*value*]*node_attribute*	Selects nodes having a specified value or range of values for the attribute. For example, the string **>=16mb** selects nodes with 16M bytes or more of RAM. The string **32mb** selects nodes with exactly 32M bytes of RAM. And, the string **>proc** selects nodes with more than one processor. The *relop* can be **=**, **>**, **>=**, **<**, **<=**, **!=**, or **!** (**!=** and **!** mean the same thing). If the *relop* is omitted, it defaults to **=**. The *value* can be any nonnegative integer. If the *value* is omitted, it defaults to 1. The *node_attribute* can be any attribute shown in the "Node Attributes" section, but is usually either **proc** or **mb**. (Other attributes have the value 1 if present or 0 if absent.) No white space may appear between the *relop*, *value*, and *attribute*.
ntype[,*ntype*]...	Selects nodes having *all* the attributes specified by the list of *ntype*s, where each *ntype* is a node type specifier of the form *node_attribute*, !*node_attribute*, or [*relop*][*value*]*node_attribute*. For example, the string **32mb, !io** selects non-io nodes with 32M bytes of RAM. You can use white space (space, tab, or newline) on either side of each comma, but not within an *ntype*.

NX_INITVE_ATTR() *(cont.)*

NX_INITVE_ATTR() *(cont.)*

Node Attributes

The following shows the most common values for *node_attribute*. A node attribute that is indented is a more specific version of the attribute from the previous level of indentation. For example, **net** and **scsi** nodes are specific types of **io** node; **enet** and **hippi** nodes are specific types of **net** node (and also specific types of **io** node).

Attribute	Meaning
bootnode	Boot node.
gp	GP (two-processor) node.
mp	MP (three-processor) node.
mcp	Node with a message coprocessor.
*n***proc**	Node with *n* application processors (not counting the message coprocessor).
*n***mb**	Node with *n*M bytes of physical RAM.
io	Any I/O nodes.
net	I/O node with any type of network interface.
enet	Network node with Ethernet interface.
hippi	Network node with HIPPI interface.
scsi	I/O node with a SCSI interface.
disk	SCSI node with any type of disk.
raid	Disk node with a RAID array.
tape	SCSI node with any type of tape drive.
3480	Tape node with a 3480 tape drive.
dat	Tape node with a DAT drive.
IDstring	SCSI node whose attached device returned the specified *IDstring*. For example, a disk node might have the *IDstring* **NCR ADP-92/01 0304**.

Specifying the Nodes Allocated to the Application

The **nx_initve_attr**() function provides the following ways to specify the nodes allocated to the application:

- Using **NX_ATTR_SZ** alone requests the specified number of nodes. A *value* of 0 or -1 requests the number of nodes specified by *$NX_DFLT_SIZE*, or all the nodes of the partition if *$NX_DFLT_SIZE* is not set.

NX_INITVE_ATTR() *(cont.)*

NX_INITVE_ATTR() *(cont.)*

NX_ATTR_SZ attempts to allocate a square group of nodes. If this is not possible, it attempts to allocate a rectangular group of nodes that is either twice as wide as it is high or twice as high as it is wide. If this is not possible, it allocates any available nodes. In this case, the nodes allocated to the application may not be contiguous.

- Using **NX_ATTR_RECT** alone requests a rectangle of nodes specified by height and width. The system places the rectangle within the partition.

- Using both **NX_ATTR_RECT** and **NX_ATTR_ANCHOR** requests a rectangle of nodes specified by height and width, whose upper left corner is located at the specified anchor node. You can place **NX_ATTR_RECT** and **NX_ATTR_ANCHOR** in any order within the argument list. If you supply a value of -1 for **NX_ATTR_ANCHOR**, the system determines the anchor node within the partition.

- Using **NX_ATTR_SEL** alone requests all nodes by attribute (of a specific node type) in the partition.

- Using **NX_ATTR_SEL** together with **NX_ATTR_SZ**, **NX_ATTR_RECT**, and/or **NX_ATTR_ANCHOR** requests the nodes specified by the **NX_ATTR_SZ**, **NX_ATTR_RECT**, and/or **NX_ATTR_ANCHOR**, all of which must have the attributes specified by the **NX_ATTR_SEL**.

- Not using **NX_ATTR_SEL**, **NX_ATTR_SZ**, **NX_ATTR_RECT**, or **NX_ATTR_ANCHOR** requests the number of nodes specified by *$NX_DFLT_SIZE*. When $NX_DFLT_SIZE is not set, all nodes of the partition are requested.

- Using **NX_ATTR_RELAXED** with a *value* of 1 together with **NX_ATTR_SEL**, **NX_ATTR_SZ**, **NX_ATTR_MAP**, **NX_ATTR_RECT**, or **NX_ATTR_ANCHOR** requests all the *available* nodes (nodes that meet the attribute requirements) in the specified node set (requested size and/or shape), *up to* the number of nodes requested. For **NX_INITVE_ATTR()** to return successfully, at least one of the specified nodes must be available.

You can override all the attributes with command-line switches, particularly the node set size and location. For example, either the **-sz** or **-nd** switch overrides **NX_ATTR_SZ**, **NX_ATTR_RECT**, and **NX_ATTR_ANCHOR**. If you override an attribute with a command-line switch, the effect is as though you had specified it in the **nx_initve_attr()** call.

The following combinations of these attributes are invalid:

- **NX_ATTR_ANCHOR** without **NX_ATTR_RECT**.

- **NX_ATTR_SZ** or **NX_ATTR_MAP** together with **NX_ATTR_RECT**.

- **NX_ATTR_RELAXED** together with **NX_ATTR_RECT**, unless you also specify **NX_ATTR_ANCHOR** with a *value* other than -1.

Using any of these combinations of attributes causes **nx_initve_attr()** to fail with the error "invalid attribute specified."

Examples

The following example creates an application whose characteristics (partition, number of nodes, and so on) are determined using command-line switches. If you run this program without command-line switches, it runs on the default number of nodes in your default partition.

```
#include <nx.h>

main(int argc, char *argv[]) {
    int n;
      .
      .
      .
    n = nx_initve_attr("", &argc, argv, NX_ATTR_END);
      .
      .
      .
}
```

After this call, the variable *n* contains the number of nodes in the new application, or a -1 if any error occurs. The variable *argc* contains the count of arguments not recognized and subsequently removed by **nx_initve()**. The array *argv* contains pointers to the arguments.

NX_INITVE_ATTR() *(cont.)* NX_INITVE_ATTR() *(cont.)*

The following example creates an application that consists of all available nodes in a rectangle 10 nodes high and 20 nodes wide whose upper left corner is node 0 (the upper left corner of the partition) in the partition *mypart*. The example ignores any command-line switches that you provide:

```
#include <nx.h>
long rect[2];
int i, n;

rect[0] = 10;
rect[1] = 20;
i = 0;
     .
     .
     .
n = nx_initve_attr("mypart", &i, NULL,
                   NX_ATTR_RELAXED, 1,
                   NX_ATTR_RECT, rect,
                   NX_ATTR_ANCHOR, 0,
                   NX_ATTR_END);
     .
     .
     .
}
```

After any of these calls, the variable *n* contains the number of nodes in the new application, or a -1 if any error occurs.

Return Values

> 0	Allocated nodes: The number of nodes allocated for the application.
-1	Error: No nodes matched the attributes specified in the attribute selector. An error has occurred and *errno* has been set. Note that the error occurs even if *NX_ATTR_RELAXED* is set to 1.

NX_INITVE_ATTR() *(cont.)* NX_INITVE_ATTR() *(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.

EAINVALMBF
The memory buffer size is invalid or out of range.

EAINVALMEA
The memory each size is invalid or out of range.

EAINVALMEX
The memory export size is invalid or out of range.

EAINVALPKT
The packet size is invalid or out of range.

EAINVALSTH
The send threshold size is invalid or out of range.

EAINVALGTH
The give threshold size is invalid or out of range.

EAOVLP A partition or application overlaps with another partition or application.

EAREJPLK An application cannot use the **-plk** switch in a gang-scheduled partition.

EINCOMPAT Your application's code is no longer up to date with the current release of the installed operating system. You must relink your application.

EPALLOCERR
An internal error occurred in the node allocation server.

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

EPBADNODE A bad node was specified.

NX_INITVE_ATTR() *(cont.)* NX_INITVE_ATTR() *(cont.)*

EPINVALPRI An invalid priority value was specified.

EPINVALPART
The specified partition was not found.

EPNOMATCH
Some nodes in the map or rectangle do not qualify. An attribute selector was specified with nodes in the map or rectangle that do not.have all the specified node attributes.

EPXRS The request exceeds the partition resources.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

commands: ***application***, **chpart**, **lspart**, **mkpart**, **pspart**, **rmpart**

calls: **nx_initve()**, **nx_mkpart_attr()**, **nx_mkpart_epl()**, **nx_rmpart()**

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.

NOTE

Do not specify the same node number more than once. If you specify the same node twice, two processes are created on the specified node, but one of the processes is terminated shortly after creation with the error `setptype: Ptype already in use`.

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).

NX_LOAD() *(cont.)*

NX_LOAD() *(cont.)*

ptype Process type of the new process(es).

pathname Pathname of the executable image to load and start.

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 *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.

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.

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.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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™ System 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. Do not specify the same node number more than once.

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.

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. You specified the same node number more than once in the *node_list* parameter.

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.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

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

NX_MKPART_ATTR() NX_MKPART_ATTR()

Creates a new partition with specified attributes.

Synopsis

```
#include <nx.h>

long nx_mkpart_attr(
    char *partition,
    [ int attribute, {long | char * | long *} value,] ...
    NX_ATTR_END);
```

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.

attribute Attribute constant to use for creating the new partition. The *attribute* parameter must be followed by the *value* parameter which sets the value of the attribute. See the "Attributes" section for the list of attribute constants you can use with the *attribute* parameter.

value Value of the attribute specified by the *attribute* parameter. A value parameter must follow each *attribute* parameter. The data type of the *value* parameter depends on the preceding *attribute* parameter. See the "Attributes" section for a description of the values for the

NX_ATTR_END
Constant that marks the end of the list of *attribute*, *value* pairs.

Description

The **nx_mkpart_attr()** function provides the functionality of the **nx_mkpart()**, **nx_mkpart_rect()**, or **nx_mkpart_map()** functions to create partitions for your application programs.

The **nx_mkpart_attr()** function creates a partition using attributes that specify the partition's characteristics. You specify the attributes in the function's argument list. An attribute consists of an attribute constant and a value. The attribute constant is the name of the attribute. The attribute value can be either an integer, array of integers, or a character string depending on the attribute. You use the *attribute* parameter to specify the attribute constant and the *value* parameter to specify the value of the attribute. See the "Attributes" section for the list of the attributes that can be set in the **nx_mkpart_attr()** function.

NX_MKPART_ATTR() *(cont.)* NX_MKPART_ATTR() *(cont.)*

When you create a partition with the **nx_mkpart_attr()** function, the new partition gets default characteristics. The partition's owner and group are set to the owner and group of the calling process. Other characteristics including the effective priority limit, protection mode, and rollin quantum are set, by default, to the same values as the parent partition, but can be changed using attributes.

Attributes

The *attribute* parameter can be set with the following attribute constants. The values for the *value* parameter are described in the "Description" column.

Attribute Constant	Description
NX_ATTR_ANCHOR	Specifies the upper-left corner of a rectangular partition when used with the **NX_ATTR_RECT** attribute. The *value* parameter must be of type **long**. If **NX_ATTR_SEL** is specified, the selected attributes must be consistent with all nodes in the list unless **NX_ATTR_RELAXED** is specified.
NX_ATTR_EPL	Specifies the effective priority limit of the new partition. The *value* parameter must be of type **long** and be an integer that ranges from 0 to 10, inclusive (0 is low priority, while 10 is high). The new partition uses gang scheduling. **NX_ATTR_EPL** can be used with or without **NX_ATTR_SCHED**. However, if **NX_ATTR_SCHED** is present, it must be set to **NX_GANG** or **NX_SPS**. If **NX_ATTR_EPL** is not specified, and the partition is to be gang scheduled (**NX_ATTR_RQ** or **NX_ATTR_SCHED** equals **NX_GANG** or **NX_SPS**), the partition has the same effective priority limit as its parent.
NX_ATTR_MAP	Specifies a set of nodes to use for a partition. The *value* parameter must be of type **long *.** It functions as a pointer to an array of node numbers. **NX_ATTR_SZ** must also be specified to give the length of the array, but need not precede it in the list of arguments. If **NX_ATTR_SEL** is specified, the selected attributes must be consistent with all nodes in the list unless **NX_ATTR_RELAXED** is specified. Do not specify the same node number more than once.

NX_MKPART_ATTR() *(cont.)*

NX_MKPART_ATTR() *(cont.)*

Attribute Constant	Description
NX_ATTR_MOD	Specifies the protection modes for the partition. The *value* parameter must be of type **long**.
NX_ATTR_RECT	Specifies a rectangular partition. The *value* parameter must be of type **long *.** It functions as a pointer to an array of two integers; the first integer is the height of the rectangle and the second integer is its width. If **NX_ATTR_SEL** is specified but **NX_ATTR_RELAXED** is not, the selected attributes must be consistent with all nodes in the rectangle.
NX_ATTR_RELAXED	Specifies whether to relax the requirement that all nodes requested must be available and eligible for allocation. The *value* parameter must be of type **long**. The *value* of 0 has no effect; the *value* of 1 relaxes the requirement.
NX_ATTR_RQ	Specifies the rollin quantum for the new partition.The *value* parameter must be of type **long**. It specifies milliseconds and must not be larger than 86,400,000 (24 hours). A value of 0 means infinite; once rolled in, an application runs to completion. **NX_ATTR_RQ** can be used with or without **NX_ATTR_SCHED**. However, if **NX_ATTR_SCHED** is present, it must be set to **NX_GANG**. If **NX_ATTR_RQ** is not specified, and the partition is to be gang scheduled (**NX_ATTR_SCHED** equals **NX_GANG**), the partition has the same rollin quantum as its parent.
NX_ATTR_SCHED	Specifies the new partition's scheduling type. The *value* parameter must be of type **long**. It must be **NX_STD** for standard, **NX_SPS** for space sharing or **NX_GANG** for gang scheduling. If you do not specify a type, it defaults to that of the parent partition. The scheduling type names are specified in the *nx.h* include file. See the *Paragon™ System User's Guide* for more information about partitions and scheduling.

NX_MKPART_ATTR() *(cont.)*

NX_MKPART_ATTR() *(cont.)*

Attribute Constant	Description
NX_ATTR_SZ	Specifies the number of nodes in the new partition. The *value* parameter must be of type **long**. A 0 (zero) or -1 for *value* requests that all nodes in the parent partition that meet the criteria specified by **NX_ATTR_SEL** be allocated. If *value* is smaller than the parent partition is specified, the nodes are selected by the system and are not necessarily contiguous.
NX_ATTR_SEL	A pointer to a Node Attribute string. The *value* parameter must be of type **char ***. If you specify multiple **NX_ATTR_SEL's**, the Attribute Selector is the logical and of all of them. Node Attribute strings are case-insensitive. The Node Attribute string may consist of a comma-separated list of selectors. See the "NX_ATTR_SEL Values" section for information on how to specify *value*.

NX_ATTR_SEL Values

The following shows the format of the *value* parameter for the **NX_ATTR_SEL** attribute.

node_attribute	Selects nodes having the specified attribute. For example, when *node_attribute* equals the string **mp**, only MP nodes are selected. The standard node attributes are shown in the "Node Attributes" section.
!*node_attribute*	Selects nodes *not* having the specified attribute. For example, when *node_attribute* equals the string **!io**, only nodes that are *not* I/O nodes are selected. Note that no white space may appear between the **!** and *node_attribute*.

NX_MKPART_ATTR() *(cont.)*

NX_MKPART_ATTR() *(cont.)*

[relop][value]node_attribute

Selects nodes having a specified value or range of values for the attribute. For example, the string **>=16mb** selects nodes with 16M bytes or more of RAM. The string **32mb** selects nodes with exactly 32M bytes of RAM. And, the string **>proc** selects nodes with more than one processor.

The *relop* can be **=**, **>**, **>=**, **<**, **<=**, **!=**, or **!** (**!=** and **!** mean the same thing). If the *relop* is omitted, it defaults to **=**.

The *value* can be any nonnegative integer. If the *value* is omitted, it defaults to 1.

The *node_attribute* can be any attribute shown in the "Node Attributes" section, but is usually either **proc** or **mb**. (Other attributes have the value 1 if present or 0 if absent.)

No white space may appear between the *relop*, *value*, and *attribute*.

ntype[**,***ntype*]...

Selects nodes having *all* the attributes specified by the list of *ntype*s, where each *ntype* is a node type specifier of the form *node_attribute*, **!***node_attribute*, or [*relop*][*value*]*node_attribute*. For example, the string **32mb, !io** selects non-io nodes with 32M bytes of RAM.

You can use white space (space, tab, or newline) on either side of each comma, but not within an *ntype*.

NX_MKPART_ATTR() *(cont.)*

NX_MKPART_ATTR() *(cont.)*

Node Attributes

The following shows the most common values for *node_attribute*. A node attribute that is indented is a more specific version of the attribute from the previous level of indentation. For example, **net** and **scsi** nodes are specific types of **io** node; **enet** and **hippi** nodes are specific types of **net** node (and also specific types of **io** node).

Attribute	Meaning
bootnode	Boot node.
gp	GP (two-processor) node.
mp	MP (three-processor) node.
mcp	Node with a message coprocessor.
***n*proc**	Node with *n* application processors (not counting the message coprocessor).
***n*mb**	Node with *n*M bytes of physical RAM.
io	Any I/O nodes.
net	I/O node with any type of network interface.
enet	Network node with Ethernet interface.
hippi	Network node with HIPPI interface.
scsi	I/O node with a SCSI interface.
disk	SCSI node with any type of disk.
raid	Disk node with a RAID array.
tape	SCSI node with any type of tape drive.
3480	Tape node with a 3480 tape drive.
dat	Tape node with a DAT drive.
IDstring	SCSI node whose attached device returned the specified *IDstring*. For example, a disk node might have the *IDstring* **NCR ADP-92/01 0304**.

Specifying the Nodes Allocated to the Partition

nx_mkpart_attr() provides the following ways to specify the nodes allocated to the partition:

- Using **NX_ATTR_SZ** alone requests the specified number of nodes. A *value* of 0 or -1 requests all the nodes in the parent partition.

  NX_ATTR_SZ attempts to create a square partition. If this is not possible, it attempts to create a rectangular partition that is either twice as wide as it is high or twice as high as it is wide. If this is not possible, it uses any available nodes. In this case, the nodes allocated to the partition may not be contiguous.

- Using both **NX_ATTR_MAP** and **NX_ATTR_SZ** requests the specified list of nodes. **NX_ATTR_MAP** and **NX_ATTR_SZ** can appear in any order in the argument list.

NX_MKPART_ATTR() *(cont.)*

NX_MKPART_ATTR() *(cont.)*

- Using **NX_ATTR_RECT** alone requests a rectangular partition of the specified height and width. The system places the rectangle within the parent partition.

- Using both **NX_ATTR_RECT** and **NX_ATTR_ANCHOR** requests a rectangular partition of the specified height and width, whose upper left corner is located at the specified anchor node within the parent partition. **NX_ATTR_RECT** and **NX_ATTR_ANCHOR** can appear in any order in the argument list. If the value of **NX_ATTR_ANCHOR** is -1, the system determines the anchor node within the parent partition.

- Using **NX_ATTR_SEL** alone requests all the nodes by attribute (of a specified node type) in the parent partition.

- Using **NX_ATTR_SEL** together with **NX_ATTR_SZ**, **NX_ATTR_MAP**, **NX_ATTR_RECT**, and/or **NX_ATTR_ANCHOR** requests the nodes specified by the **NX_ATTR_SZ**, **NX_ATTR_MAP**, **NX_ATTR_RECT**, and/or **NX_ATTR_ANCHOR**, all of which must have the node type specified by the **NX_ATTR_SEL**.

- Not using **NX_ATTR_SEL**, **NX_ATTR_SZ**, **NX_ATTR_MAP**, **NX_ATTR_RECT**, or **NX_ATTR_ANCHOR** requests all the nodes in the parent partition.

- Using **NX_ATTR_RELAXED** with a *value* of 1 together with **NX_ATTR_SEL**, **NX_ATTR_SZ**, **NX_ATTR_MAP**, **NX_ATTR_RECT**, or **NX_ATTR_ANCHOR** requests all the *available* nodes (nodes that meet the attribute requirements) in the specified node set (requested size and/or shape), *up to* the number of nodes requested. For **NX_MKPART_ATTR()** to return successfully, at least one of the specified nodes must be available.

The following combinations of these attributes are invalid:

- **NX_ATTR_MAP** without **NX_ATTR_SZ**.

- **NX_ATTR_ANCHOR** without **NX_ATTR_RECT**.

- **NX_ATTR_SZ** or **NX_ATTR_MAP** together with **NX_ATTR_RECT**.

- **NX_ATTR_RELAXED** together with **NX_ATTR_RECT**, unless you also specify **NX_ATTR_ANCHOR** with a *value* other than -1.

Using any of these combinations of attributes causes **nx_mkpart_attr()** to fail with the error "invalid attribute specified."

NX_MKPART_ATTR() *(cont.)*

NX_MKPART_ATTR() *(cont.)*

Examples

The following example creates a new partition called *newpart* (using a relative partition pathname) whose parent partition is the *.compute* partition. The new partition consists of all the nodes in the *.compute* partition and has the same scheduling type, rollin quantum, and effective priority limit as the *.compute* partition. In this example (and those following), the variable *n* is assigned the number of nodes in the new partition, or -1 if any error occurred.

```
include <nx.h>
int n;
     .
     .
     .
n = nx_mkpart_attr("newpart", NX_ATTR_END);
     .
     .
     .
}
```

The following example creates a new space-shared partition called *mypart* (using an absolute partition pathname) whose parent partition is the *.compute* partition and which has 54 nodes:

```
#include <nx.h>
int n;
     .
     .
     .
n = nx_mkpart_attr(".compute.mypart",
                   NX_ATTR_SZ, 54,
                   NX_ATTR_SCHED, NX_SPS,
                   NX_ATTR_END);
     .
     .
     .
}
```

NX_MKPART_ATTR() *(cont.)*

NX_MKPART_ATTR() *(cont.)*

The following example creates a new gang-scheduled partition called *rect* whose parent partition is *mypart*. It is 3 nodes high and 4 nodes wide, and has its upper left corner at node 1 of *mypart*. It has a rollin quantum of 600,000 milliseconds (10 minutes) and the same effective priority limit as *mypart*:

```
#include <nx.h>
long rect[2];
int n;
     .
     .
     .
rect[0] = 3;
rect[1] = 4;

n = nx_mkpart_attr(".compute.mypart.rect",
                   NX_ATTR_RECT, rect,
                   NX_ATTR_ANCHOR, 1,
                   NX_ATTR_RQ, 600000,
                   NX_ATTR_END);
     .
     .
     .
}
```

The following example creates a new gang-scheduled partition called *corners* whose parent partition is *rect* and consists of the four corner nodes of *rect*. It has an effective priority limit of 3. All other characteristics are the same as *rect*:

```
#include <nx.h>
long nodes[4];
int n;
     .
     .
nodes[0] = 0;
nodes[1] = 3;
nodes[2] = 8;
nodes[3] = 11;
n = nx_mkpart_attr(".compute.mypart.rect.corners",
                   NX_ATTR_MAP, nodes,
                   NX_ATTR_SZ, 4,
                   NX_ATTR_EPL, 3,
                   NX_ATTR_END);
     .
     .
}
```

NX_MKPART_ATTR() *(cont.)* NX_MKPART_ATTR() *(cont.)*

The following example creates a new partition called *bigmem* whose parent partition is the *.compute* partition and consists of all *available* nodes with 64M bytes or more of physical RAM. All other characteristics of *bigmem* are the same as those of the *.compute* partition:

```
include <nx.h>
int n;
    .
    .
    .
n = nx_mkpart_attr("bigmem",
                   NX_ATTR_SEL, ">=64mb",
                   NX_ATTR_RELAXED, 1,
                   NX_ATTR_END);
    .
    .
    .
}
```

Return Values

> 0	Allocated nodes: The number of nodes allocated for the partition.
-1	Error: No nodes matched the attributes specified in the attribute selector. An error has occurred and *errno* has been set. Note that the error occurs even if *NX_ATTR_RELAXED* is set to 1.

Errors

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

EINVAL	Invalid attribute specified in the *attribute* parameter, including error in the Some nodes in the map or rectangle do not qualify attribute selector.
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.

EPNOMATCH
Some nodes in the map or rectangle do not qualify. An attribute selector was specified with nodes in the map or rectangle that do not.have all the specified node attributes.

EPPARTEXIST
The specified partition already exists.

EPXRS Request exceeds the partition's resources.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

commands: ***application*, chpart, lspart, mkpart, pspart, rmpart**

calls: **nx_mkpart_epl(), nx_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.

NOTE

Do not specify the same node number more than once. If you specify the same node twice, two processes are created on the specified node, but one of the processes is terminated shortly after creation with the error `setptype: Ptype already in use.`

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).

NX_NFORK() *(cont.)*

NX_NFORK() *(cont.)*

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. Valid *pid_list* values exist only for the calling process. The values in the *pid_list* arrays of any child processes created by **nx_nfork()** are invalid.

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.

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.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

nx_initve(), **nx_load()**, **setptype()**

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

NX_PART_ATTR() NX_PART_ATTR()

Returns information about a partition.

Synopsis

```
#include <nx.h>

int nx_part_attr(
    char *partition,
    nx_part_info_t *attributes);
```

Parameters

partition Relative or absolute pathname of a partition. The partition must exist and give read permission to the calling process.

attributes Pointer to an **nx_part_info_t** structure that contains information about the partition specified by the *partition* parameter. The **nx_part_info_t** type is defined in the include file *allocsys.h* (included in the include file *nx.h*). You must allocate space for this structure.

Description

The **nx_part_attr()** function returns the partition characteristics of the partition specified by the *partition* parameter.

The **nx_part_info** structure includes the following fields:

uid User ID for the partition's owner.

gid Group ID for the partition's owner.

access Access permissions for the partition. A three-digit octal number.

sched Scheduling type for the partition (defined in *nx.h*):

NX_GANG Gang scheduling.

NX_SPS Space sharing.

NX_STD Standard scheduling.

NX_PART_ATTR() *(cont.)* NX_PART_ATTR() *(cont.)*

rq	Rollin quantum for the partition. The value is 0 (zero) for a standard-scheduled or space-shared partition.
epl	Effective priority limit for the partition. The value is 0 (zero) for a standard-scheduled partition.
nodes	Number of nodes in the partition.
mesh_x	Width of the partition (*columns*). This is set only if the node set is a contiguous rectangle.
mesh_y	Height of the partition (*rows*). This is set only if the node set is a contiguous rectangle.
enclose_mesh_x	Width of the smallest rectangle that completely encloses the partition.
enclose_mesh_y	Height of the smallest rectangle that completely encloses the partition.

Return Values

On successful completion, the **nx_part_info()** function returns 0 (zero). Otherwise, -1 is returned and *errno* is set to indicate the error.

Example

The following example prints the rollin quantum and effective priority limit for the partition *mypart*:

```
#include <nx.h>
main() {

    nx_part_info_t info;
    int            status;

    status = nx_part_attr("mypart", &info);

    if(status != 0) {
        nx_perror("nx_part_attr()");
        exit(1);
    }

    printf("rq = %d, epl = %d\n", info.rq, info.epl);
}
```

Note the use of the & operator on the structure *info* in the call to **nx_part_attr()**.

Errors

EPACCES The application has insufficient access permission on a partition.

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

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

chpart, **lspart**, **nx_chpart_epl()**, **nx_pspart()**, **nx_part_nodes()**, **pspart**, **showpart**

NX_PART_NODES() NX_PART_NODES()

Returns the root partition node numbers for a partition.

Synopsis

```
#include <nx.h>

int nx_part_nodes(
    char *partition,
    nx_nodes_t *node_list,
    unsigned long *list_size);
```

Parameters

partition	Relative or absolute pathname of a partition. The specified partition must exist and must give read permission to the calling process.
node_list	Pointer variable into which the **nx_part_nodes()** function stores the address of the list of nodes in *partition*. The call allocates memory for this parameter. Free this memory using the **free()** function.
list_size	Address of a variable into which the **nx_part_nodes()** function stores the number of elements in the *node_list* array.

Description

The **nx_part_nodes()** function returns the root partition node numbers for the partition specified by the *partition* parameter.

Return Values

On successful completion, the **nx_part_nodes()** function returns 0 (zero). Otherwise, -1 is returned and *errno* is set to indicate the error.

NX_PART_NODES() *(cont.)* NX_PART_NODES() *(cont.)*

Examples

The following example prints the root node numbers for the partition *mypart*:

```
#include <nx.h>
main() {

    nx_nodes_t      mynodes;
    unsigned long   nnodes;
    int             i, status;

    status = nx_part_nodes("mypart", &mynodes, &nnodes);

    if(status != 0) {
        nx_perror("nx_part_nodes()");
        exit(1);
    }

    for(i = 0; i < nnodes; i++) {
        printf("%d\n", mynodes[i]);
    }

    free(mynodes);
}
```

Errors

EPACCES The application has insufficient access permission on a partition.

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

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

mynode(), nx_app_nodes(), nx_empty_nodes(), nx_failed_nodes()

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.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

NX_PERROR() *(cont.)*

NX_PERROR() *(cont.)*

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™ System 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.

Errors

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

EANOEXIST The specified process group is an invalid value. For example, you specified a negative number for the process group value.

EPALLOCERR An internal error occurred in the node allocation server.

EPERM The calling process does not have permission to change the application's priority.

EPINVALPRI The specified priority is out of the range of priority values.

ESRCH The specified process group does not exist.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

nx_chpart(), nx_initve(), nx_nfork(), nx_load()

NX_PSPART() NX_PSPART()

Returns information about the applications and active partitions in a specified partition.

Synopsis

```
#include <sys/time.h>
#include <nx.h>

int nx_pspart(
    char *partition,
    nx_pspart_t **pspart_list,
    unsigned long *list_size);
```

Parameters

partition Relative or absolute pathname of a partition. The specified partition must exist and must give read permission to the calling process.

pspart_list Pointer variable into which the **nx_pspart()** function stores the address of an array of **nx_pspart_t** structures. Each structure contains information about an application or active partition in the partition specified by the *partition* parameter. The **nx_pspart_t** type is defined in the include file *allocsys.h*, which is included by the include file *nx.h*. The call allocates memory for this parameter. Free this memory using the **free()** function.

list_size Pointer variable into which the **nx_pspart()** function stores the number of elements in the *pspart_list* parameter.

Description

The **nx_pspart()** function provides information about the status of the applications and active partitions in a specified partition. The **nx_pspart_t** structure contains the following information:

object_type Indicates if the object is an active partition (**NX_PARTITION**) or an application (**NX_APPLICATION**).

object_id Process group ID for an application or a partition ID (arbitrary integer) for a partition.

uid Numeric user ID of the object's (partition or application) owner.

NX_PSPART() *(cont.)* NX_PSPART() *(cont.)*

gid Numeric group ID of the object's group.

size Number of nodes in the object.

priority Priority of the object.

rolled_in Amount of time the object has been rolled in during the current rollin quantum, in milliseconds.

rollin_q Rollin quantum of the object's parent partition (the partition specified in the **nx_pspart()** call), in milliseconds.

elapsed Total amount of time the object has been rolled in since it was started, in milliseconds.

active Indicates whether the object is active (rolled in), inactive (rolled out), and/or has been dumping core. The values are as follows:

0	Object is inactive and is or has not been dumping core.
1	Object is active and is or has not been dumping core.
2	Object is inactive and is either currently dumping core or has dumped core. This *active* value applicable only when object is an application.
3	Object is active and is either currently dumping core or has dumped core. This *active* value applicable only when object is an application.

time_started Time the object was started, as returned by the **time()** call. If the object is a subpartition, the time is when the oldest application started in the subpartition.

Return Values

On successful completion, the **nx_pspart()** function returns 0 (zero). Otherwise, -1 is returned and *errno* is set to indicate the error.

NX_PSPART() *(cont.)* NX_PSPART() *(cont.)*

Examples

The following example prints the numeric user ID and size for every application and subpartition in the partition *mypart*:

```
#include <nx.h>
main() {

   nx_pspart_t   *info;
   nx_pspart_t   *ptr;
   unsigned long  nobjs;
   int            status, i;

   status = nx_pspart("mypart", &info, &nobjs);

   if(status != 0) {
       nx_perror("nx_pspart()");
       exit(1);
}
   ptr = info;
   for(i = 0; i < nobjs; i++) {
       printf("uid = %d, size = %d\n", ptr->uid, ptr->size);
   ptr++;
   }

   free(info);
}
```

Note the use of the & operator on the structure *info* and the variable *nobjs* in the call to **nx_pspart()**.

Errors

EPACCES The application has insufficient access permission on a partition.

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

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

pspart

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.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

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.

If the **nx_waitall()** function detects that one of the processes being waited for has been terminated by the signal **SIGBUS**, **SIGFPE**, **SIGILL**, **SIGSEGV**, or **SIGSYS**, the **nx_waitall()** function terminates the whole application by sending a **SIGKILL** to the process group.

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.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

NX_WAITALL() *(cont.)*

NX_WAITALL() *(cont.)*

See Also

nx_nfork(), nx_load()

OPEN() OPEN()

open(), **creat()**: Opens or creates a file for reading or writing.

Synopsis

```
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>

int open(
    const char *path,
    int oflag [ ,
    mode_t mode ] );

int creat(
    const char *path,
    mode_t mode );
```

Parameters

path Specifies the file to be opened or created. If the *path* parameter refers to a symbolic link, the **open**() function opens the file pointed to by the symbolic link.

oflag Specifies the type of access, special open processing, the type of update, and the initial state of the open file. The parameter value is constructed by logically ORing special open processing flags. These flags are defined in the **fcntl.h** header file and are described below.

mode Specifies the read, write, and execute permissions of the file to be created (requested by the O_CREAT flag in the **open**() interface). If the file already exists, this parameter is ignored. This parameter is constructed by logically ORing values described in the **sys/mode.h** header file.

OPEN() *(cont.)*

OPEN() *(cont.)*

Description

The **open**() and **creat**() functions establish a connection between the file named by the *path* parameter and a file descriptor. The opened file descriptor is used by subsequent I/O functions, such as **read**() and **write**(), to access that file.

The returned file descriptor is the lowest file descriptor not previously open for that process. No process can have more than **OPEN_MAX** file descriptors open simultaneously.

The **open**() and **creat**() functions, which suspend the calling process until the request is completed, are redefined so that only the calling thread is suspended.

The file offset, marking the current position within the file, is set to the beginning of the file. The new file descriptor is set to remain open across **exec** functions. (See the **fcntl**() function.)

The file status flags and file access flags are designated by the *oflag* parameter. The *oflag* parameter is constructed by bitwise-inclusive ORing exactly one of the file access flags (**O_RDONLY**, **O_WRONLY**, or **O_RDWR**) with one or more of the file status flags.

File Access Flags

The file access flags are as follows:

O_RDONLY The file is open for reading only.

O_WRONLY The file is open for writing only.

O_RDWR The file is open for reading and writing.

Exactly one of the file access values (**O_RDONLY**, **O_WRONLY**, or **O_RDWR**) must be specified. If none is set, **O_RDONLY** is assumed.

File Status Flags

File status flags that specify special open processing are as follows:

O_CREAT If the file exists, this flag has no effect except as noted under **O_EXCL**. If the file does not exist, a regular file is created with the following characteristics:

- The owner ID of the file is set to the effective user ID of the process.
- The group ID of the file is set to the group ID of its parent directory.

OPEN() *(cont.)*

OPEN() *(cont.)*

- The file permission and attribute bits are set to the value of the *mode* parameter, modified as follows:

  All bits set in the process file mode creation mask are cleared.

  The set-user ID attribute (**S_ISUID** bit) is cleared.

  The set-group ID attribute (**S_ISGID** bit) is cleared.

  The **S_ISVTX** attribute bit is cleared.

The calling process must have write permission to the file's parent directory with respect to all access control policies to create a new file.

O_EXCL If **O_EXCL** and **O_CREAT** are set, the open fails if the file exists.

O_NOCTTY If the *path* parameter identifies a terminal device, this flag assures that the terminal device does not become the controlling terminal for the process.

O_TRUNC If the file does not exist, this flag has no effect. If the file exists and is a regular file, and if the file is successfully opened **O_RDWR** or **O_WRONLY**:

- The length of the file is truncated to 0 (zero).
- The owner and group of the file are unchanged.
- The set-user ID attribute of the file mode is cleared.
- The set-user ID attribute of the file is cleared.

The open fails if either of the following conditions are true:

- The file supports enforced record locks and another process has locked a portion of the file.
- The file does not allow write access.

If the *oflag* parameter also specifies **O_SYNC**, the truncation is a synchronous update.

A program can request some control over when updates should be made permanent for a regular file opened for write access.

OPEN() *(cont.)*

OPEN() *(cont.)*

File status flags that define the initial state of the open file are as follows:

O_SYNC If set, updates and writes to regular files and block devices are synchronous updates. File update is performed by:

- **fclear**()
- **ftruncate**()
- **open**() with **O_TRUNC**
- **write**()

On return from a function that performs a synchronous update (any of the above system calls, when **O_SYNC** is set), the calling process is assured that all data for the file has been written to permanent storage, even if the file is also open for deferred update.

O_APPEND If set, the file pointer is set to the end of the file prior to each write.

O_NONBLOCK, O_NDELAY
If set, the call to **open**() will not block, and subsequent **read**() or **write**() operations on the file will be nonblocking.

General Notes on oflag Parameter Flag Values

The effect of **O_CREAT** is immediate.

When opening a FIFO with **O_RDONLY**:

- If neither **O_NDELAY** nor **O_NONBLOCK** is set, the **open**() function blocks until another process opens the file for writing. If the file is already open for writing (even by the calling process), the **open**() function returns without delay.
- If **O_NDELAY** or **O_NONBLOCK** is set, the **open**() function returns immediately.

When opening a FIFO with **O_WRONLY**:

- If neither **O_NDELAY** nor **O_NONBLOCK** is set, the **open**() function blocks until another process opens the file for reading. If the file is already open for reading (even by the calling process), the **open**() function returns without delay.
- If **O_NDELAY** or **O_NONBLOCK** is set, the **open**() function returns an error if no process currently has the file open for reading.

OPEN() *(cont.)*

OPEN() *(cont.)*

When opening a block special or character special file that supports nonblocking opens, such as a terminal device:

- If neither **O_NDELAY** nor **O_NONBLOCK** is set, the **open**() function blocks until the device is ready or available.
- If **O_NDELAY** or **O_NONBLOCK** is set, the **open**() function returns without waiting for the device to be ready or available. Subsequent behavior of the device is device-specific.

Numbered Files

If three or more # characters are in a file name, these characters are replaced by the number of the node (within the application) that opens the file. For example, assume that the same program is running on several nodes, and each node opens a file named *file###*. The result is that each node opens a separate file. Node 0 opens *file000*, node 1 opens *file001*, node 2 opens *file002*, and so on.

If the node number has more than three digits but the filename has only three # characters, the filename is lengthened by the number of characters necessary to add the extra digits to the name. For example, opening *data.###* on every node of an application running on 2000 nodes opens files *data.000*, *data.001*, *data.002*, ..., *data.999*, *data.1000*, *data.1001*, ..., *data.1998*, and *data.1999*.

Less than three # characters in the file name appear as actual # characters. For example, the file *file##1* is a single file accessible by each node.

Return Values

Upon successful completion, the **open**() and **creat**() functions return the file descriptor, a nonnegative integer. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

Errors

If the **open**() or **creat**() function fails, **errno** may be set to one of the following values:

EACCES Search permission is denied on a component of the path prefix, or the type of access specified by the *oflag* parameter is denied for the named file, or the file does not exist and write permission is denied for the parent directory, or **O_TRUNC** is specified and write permission is denied.

EAGAIN The **O_TRUNC** flag is set, the named file exists with enforced record locking enabled, and there are record locks on the file.

OPEN() *(cont.)*

OPEN() *(cont.)*

EDQUOT The directory in which the entry for the new link is being placed cannot be extended because the quota of disk blocks or i-nodes defined for the user on the file system containing the directory has been exhausted.

EEXIST The **O_CREAT** and **O_EXCL** flags are set and the named file exists.

EFAULT The *path* parameter is an invalid address.

EINTR A signal was caught during the **open**() function.

EISDIR The named file is a directory and write access is requested.

ELOOP Too many links were encountered in translating *path*.

EMFILE The system limit for open file descriptors per process has already reached **OPEN_MAX**.

ENAMETOOLONG
The length of the *path* string exceeds **PATH_MAX**, or a pathname component is longer than **NAME_MAX**.

ENFILE The system file table is full.

ENOENT The **O_CREAT** flag is not set and the named file does not exist, or **O_CREAT** is set and the path prefix does not exist, or the *path* parameter points to the empty string.

ENOSPC The directory that would contain the new file cannot be extended, the file does not exist, and **O_CREAT** is requested.

ENOTDIR A component of the path prefix is not a directory.

ENXIO The named file is a character special or block special file, and the device associated with this special file does not exist.

The named file is a multiplexed special file and either the channel number is outside of the valid range or no more channels are available.

The **O_NONBLOCK** flag is set, the named file is a **FIFO**, **O_WRONLY** is set, and no process has the file open for reading.

EOPNOTSUPP The named file is a socket bound to the file system (a UNIX domain socket) and cannot be opened.

OPEN() *(cont.)*

OPEN() *(cont.)*

EROFS The named file resides on a read-only file system and write access is required.

ETXTBSY The file is being executed and *oflag* is **O_WRONLY** or **O_RDWR**.

See Also

Functions: **chmod(2)**, **close(2)**, **fcntl(2)**, **lockf(3)**, **lseek(2)**, **read(2)**, **stat(2)**, **truncate(2)**, **umask(2)**, **write(2)**

PFS_HOST_INIT() PFS_HOST_INIT()

Populate an emulator's PFS stripe directory cache.

Synopsis

int **pfs_host_init**(
char **pfs_name*);

Parameters

pfs_name Pointer to the root of a PFS file system (for example, "/pfs").

Description

The **pfs_host_init**() call populates the calling task's emulator-resident, PFS-stripe-directory cache with (Mach IPC) ports for each of the PFS stripe directories. These ports allow the emulator to communicate directly with the file server that services each PFS stripe file.Without this cache, the emulator sends pathname operations to the boot-node file server, which redirects them to the file server that services the stripe file. Using the **pfs_host_init**() call results in a significant Mach IPC load reduction for the boot-node.

The cache exists in the portion of the emulator's memory that is inherited across the **fork**() family of system calls. Consequently, the **pfs_host_init**() call need only be called by the parent of a parallel program; all children will inherit the PFS-stripe-directory cache.

The **pfs_host_init**() call is most effective for those programs that do repetitive pathname system calls (**open**(), **stat**(), **unlink**(), **access**(), and so on) on PFS-resident files. Virtually any system call that has a pathname argument that references a PFS file will benefit from using the **pfs_host_init**() call.

Return Values

Return values are those defined in */usr/include/errno.h*:

ESUCCESS Indicates success.

ENOENT Indicates a bad PFS path or one that is not a PFS file system.

PFS_HOST_INIT() *(cont.)*

PFS_HOST_INIT() *(cont.)*

Limitations and Workarounds

The **pfs_host_init**() call can be used only once per application.

Only one PFS file system can be cached per application.

If a cached PFS file system is dismounted and then remounted, the cache will be invalid.

RMKNOD() RMKNOD()

Creates a special file on a remote I/O node

Synopsis

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

int rmknod (
    const char *path,
    int mode,
    dev_t device,
    long node);
```

Parameters

path	Names the new file. If the final component of the path parameter names a symbolic link, the link will be traversed and pathname resolution will continue.
mode	Specifies the file type, attributes, and access permissions. This parameter is constructed by logically ORing values described in the *sys/mode.h* header file.
device	Depends upon the configuration and is used only if the mode parameter specifies a block or character special file. If the file you specify is a remote file, the value of the device parameter must be meaningful on the node where the file resides.
node	Node number of a remote I/O node that can be the boot node or any other I/O node.

Description

Other than the addition of the *node* parameter, the **rmknod()** function is identical to the OSF/1 **mknod()** function. See the **mknod(2)** manual page in the *OSF/1 Programmer's Reference*.

The **rmknod()** function creates a special file that references a remote I/O node specified by the *node* parameter. The remote I/O node can be the boot node or any other I/O node. This function requires superuser privilege.

RMKNOD() *(cont.)* RMKNOD() *(cont.)*

Return Values

Upon successful completion of the **rmknod()** function a value of 0 (zero) is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

Errors

If the **rmknod()** function fails, *errno* may be set to one of the error code values described for the OSF/1 **mknod()** function.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

rmknod

OSF/1 Programmer's Reference: **chmod(2)**, **mkdir(2)**, **mknod(2)**, **open(2)**, **umask(2)**, **stat(2)**

OSF/1 Command Reference: **chmod(1)**, **mkdir(1)**, **mknod(8)**

READOFF() READOFF()

readoff(), **readvoff**(): Synchronous reads from a file at a specified offset.

Synopsis

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

int readoff(
        int fildes,
        esize_t offset,
        char *buffer,
        unsigned int nbytes );


#include <sys/types.h>
#include <sys/uio.h>

int readvoff(
        int fildes,
        esize_t offset,
        struct iovec *iov,
        int iovcount );
```

Description of Parameters

fildes	A file descriptor identifying the file to be read.
offset	Offset from the beginning of the file where to begin the read.
buffer	Pointer to the buffer in which the data is stored after it is read.
nbytes	The number of bytes to read from the file associated with the *fildes* parameter.
iov	Pointer to an array of *iovec* structures that identify the buffers into which the data is to be placed.
iovcount	The number of *iovec* structures pointed to by the *iov* parameter.

READOFF() *(cont.)* READOFF() *(cont.)*

Discussion

Readoff() and **readvoff**() perform the read operation starting at the offset specified by the *offset* parameter.

These functions do not modify the system file pointer(s) (unlike **read**() and **readv**()).

Currently these functions can be used only on files on the Paragon PFS.

Currently only M_UNIX and M_ASYNC I/O modes are supported.

Return Values

Upon successful completion, a non-negative integer representing the number of bytes read is returned. If an error occurs, these functions return -1 and set *errno* to indicate the error.

Errors

Errors are as described in OSF/1 **read**(), except that the following errors can also occur:

EFSNOTSUPP The file referred to by *filedes* is not in a file system of a type that supports this operation. Currently only the PFS file systems support this operation.

EINVAL The file referred to by *filedes* is in an unsupported iomode. Currently only M_UNIX and M_ASYNC are supported.

See Also

cread(), **gopen**(), **iodone**(), **iowait**(), **iread**(), **ireadoff**(), **iseof**(), **niodone**(), **niowait**(), **setiomode**()

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

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.

M_GLOBAL All nodes use the same file pointer, all nodes perform the same operations.

M_ASYNC Each node has its own file pointer; access is unrestricted; I/O atomicity is *not* preserved in order to allow multiple readers/multiple writers and records of variable length.

Refer to the "Description" section for detailed information on each mode.

SETIOMODE() *(cont.)*

SETIOMODE() *(cont.)*

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 all nodes in an application. To explicitly specify an I/O mode at the time a file is opened, use the **gopen()** function.

The default I/O mode shared files are opened with depends on two things: the type of file and the value of the *PFS_ASYNC_DFLT* bootmagic string. Behavior is as follows:

non-PFS files — The default I/O mode is **M_UNIX** for all non-PFS files. This behavior holds true regardless of the *PFS_ASYNC_DFLT* bootmagic string.

PFS files — The default I/O mode is **M_UNIX** when *PFS_ASYNC_DFLT* is set to any value other than 1. When *PFS_ASYNC_DFLT* is set to 1, the default I/O mode is **M_ASYNC**.

This method of determining the default I/O mode also holds true during **fork()** operations. In other words, the I/O modes associated with the parent process' file descriptors are not inherited by the child process. Instead, all I/O modes in the child process default accordingly. When using the **dup()** function to duplicate a file, the file descriptor for the duplicate file is reset to the I/O mode **M_UNIX**.

NOTE

To determine the current setting for *PFS_ASYNC_DFLT*, use the **getmagic** command. For information on this command, see the **getmagic** manual page.

Each node calling **setiomode()** must specify a file descriptor with the *fildes* parameter that refers to the same file. The file pointer must be in the same position in the file for each node at the time the call to **setiomode()** is made.

In addition to setting the file's I/O mode, **setiomode()** performs a global synchronizing operation like that of the **gsync()** call. All nodes must call the **setiomode()** function before any node can continue executing. In the **M_LOG**, **M_SYNC**, **M_RECORD**, and **M_GLOBAL** I/O modes, closing the file also performs a global synchronizing operation.

Use the **iomode()** function to return a file's current I/O mode.

M_UNIX (Mode 0)

The features of this mode are as follows:

- Each node has a unique file pointer.
- Nodes are not synchronized.
- Variable-length, unordered records.

This mode conforms with standard UNIX file sharing semantics for different processes accessing the same file. 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 one node overwrites the data written previously by the other node.

This mode is often used when each node is responsible for data in a specific area of a file.

Although nodes are not synchronized as in the **M_SYNC** mode, this mode currently supports only a single reader/single writer. If multiple readers/multiple writers are required, use the **M_RECORD** or **M_ASYNC** modes. If all nodes read the same data, use the **M_GLOBAL** mode.

Depending on the shared file type (PFS or non-PFS) and the *PFS_ASYNC_DFLT* bootmagic variable setting, M_UNIX can be the default I/O mode (see the "Description" section for more information).

M_LOG (Mode 1)

The features of this mode are as follows:

- Shared file pointer.
- Nodes are not synchronized.
- Variable-length, unordered records.

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_RECORD**, **M_GLOBAL**, and **M_ASYNC** modes.

M_SYNC (Mode 2)

The features of this mode are as follows:

- Shared file pointer.
- Nodes are synchronized.
- Variable-length records, stored in node order.

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 requested by the application to be read or written may vary from node to node.

In this mode, all nodes must perform the same file operations in the same order. The only valid use of the **lseek()** and **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 than the **M_RECORD**, **M_GLOBAL**, and **M_ASYNC** modes.

M_RECORD (Mode 3)

The features of this mode are as follows:

- Unique file pointer.
- Nodes are not synchronized.
- Fixed-length records, stored in node order.
- Highly parallel.

In this mode, each node maintains its own file pointer and the application 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. This guarantees that each node reads/writes to separate areas of the file, allowing the file system to provide access to the file in a highly parallel fashion.

NOTE

No verification is performed. You must make sure that all the nodes in the application make the same calls and read and write the same number of bytes.

Files created in this mode resemble files created in the **M_SYNC** mode (that is, the data appear in node order). The application should perform the same file operations in the same order on all nodes. However, for higher performance only the **lseek()** and **eseek()** system calls are synchronized. 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 all nodes may access the file in parallel when either reading or writing, this mode offers higher performance than the **M_UNIX**, **M_LOG**, and **M_SYNC** modes.

M_GLOBAL (Mode 4)

The features of this mode are as follows:

- Shared file pointer.
- Nodes are synchronized.
- Variable-length, unordered records.
- All nodes access the same data.
- Data read/written from/to disk only once.

This mode coordinates I/O requests so that multiple identical I/O requests to the same file from different nodes are not issued.

In the **M_GLOBAL** mode, all nodes use the same file pointer for a file, and each I/O request from an application is a global operation in which all nodes must perform the same file accesses in the same order. All nodes read the same data and all nodes write the same data, although the data written is not checked. All write operations return the same number of bytes written. The only valid use for the **lseek()** or **eseek()** functions is for all nodes to seek to the same position in the file prior to an access.

Because identical requests are combined into a single request, the **M_GLOBAL** mode provides a higher-performance alternative to the **M_UNIX** mode when all nodes read and write the same data. For example, this mode is useful for parallel applications that initialize by having all nodes sequentially read the same data file.

M_ASYNC (Mode 5)

The features of this mode are as follows:

- Each node has a unique file pointer.
- Nodes are not synchronized.
- Variable-length, unordered records.
- Multiple readers/multiple writers are allowed with no restrictions.

The **M_ASYNC** mode is similar to the **M_UNIX** mode, except it does not support standard UNIX file sharing semantics for different processes accessing the same file. This mode does not guarantee that I/O operations are atomic. For example, if multiple nodes write to the same area of a file at the same time, parts of the file area may contain data from one write while other parts may contain data from other writes. If a node reads from the same area of the file at this time, the returned data may consist partially of old data and partially of new data. Other I/O modes guarantee that I/O operations are atomic, so that only the data from one write is seen in areas of the file where multiple processes are writing simultaneously, and all nodes are notified when the file size changes.

In this mode, an application must control parallel access to the file. This allows multiple readers and/or multiple writers to access the file simultaneously with no restrictions on record size or file offset.

If a file is opened with the **O_APPEND** flag and multiple nodes write to the file simultaneously, the results are unpredictable because nodes are not synchronized whenever the end-of-file changes.

It is not required that all nodes read or write to the file, and there are no restrictions on using **lseek()** or **eseek()**.

Because all nodes may access the file in parallel when either reading or writing, this mode offers higher performance than the **M_UNIX**, **M_LOG**, and **M_SYNC** modes.

You can cause **M_ASYNC** mode to be the default I/O mode by setting the *PFS_ASYNC_DFLT* bootmagic string to one (1).

SETIOMODE() *(cont.)* SETIOMODE() *(cont.)*

Return Values

Upon successful completion, the **setiomode()** function returns control to the calling process; no values are returned. Otherwise, the **setiomode()** function writes an error message on the standard error output 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.

EINVAL The file referenced by *filedes* is not a regular file.

EMIXIO The given *filedes* is invalid because all nodes have not specified a *filedes* that represents the same file.

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**, **M_RECORD**, or **M_GLOBAL**, all nodes sharing the file have not set the file pointer to the same location.

SETIOMODE() *(cont.)*

SETIOMODE() *(cont.)*

Examples

The following example shows how to use the **setiomode()** function to set the I/O mode after opening a file, but before writing to the file.

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

long iam;
main()
{
   int fd;
   char buffer[80];

   iam = mynode();

fd = gopen("/tmp/mydata",O_CREAT | O_TRUNC | O_RDWR, M_UNIX,
0644);

/* Read some data from the file and do some computation */
/* on the data before changing the file mode and writing */
/* the file. */

   setiomode(fd, M_RECORD);

   sprintf(buffer,"Hello from node %d\n",iam);
   cwrite(fd, buffer, strlen(buffer));
   close(fd);
}
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

cread(), **cwrite()**, **gopen()**, **iomode()**, **iread()**, **iwrite()**

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

SETPTYPE() SETPTYPE()

Sets the process type of the calling 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**30 - 1.

Description

The calling process's process type can be set only if the process type is currently **INVALID_PTYPE**. A process cannot change it's process type once it has been set to a valid value.

The **setptype**() function sets the process type of a calling process. A process type is an integer that uniquely distinguishes a process from another process in the same application on the same node. You can use process types with processes as follows:

- A process can have one process type only.
- Processes on different nodes may have the same process type.
- Multiple processes running on the same node in the same application must have different process types (ptypes).
- Multiple processes running on the same node may have the same process type only if they belong to different applications.
- A process may not change its process type once it has set a valid process type.
- Once a process has used a process type, the process type is associated with the process for the life of the application. No other process on the same node in the same application can use that process type, even if the original process terminates.

SETPTYPE() *(cont.)*

SETPTYPE() *(cont.)*

The **setptype**() function has the following restrictions:

- Do not use the **setptype**() function in applications linked with the **-nx** switch. Instead, link with the **-lnx** switch. For all processes in applications linked with the **-nx** switch, the process type is set automatically to the value specified with the **-pt** switch. The default process type value is 0 (zero).
- Do not use the **setptype**() function in processes created with the **nx_nfork**(), **nx_load**(), or **nx_loadve**() functions. These functions have a *ptype* parameter for specifying the process type of newly created processes in an application.
- Do not use the **setptype**() function in controlling processes that do not use message passing, because the **setptype**() function assigns memory for message buffering that will be unused.

If an application creates additional processes after it starts up and no process type is specified for the new process, the process type of the new process is set to the 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. A process must call **setptype**() to set its process type to a valid value before it can send or receive any messages. (This is the only valid use of the **setptype**() function.)

The standard OSF/1 **fork**() function creates a new process on the same node as the process that calls it. The **fork**() function does not provide any way to specify the new process's process type. The process type of a process created by **fork**() is set to **INVALID_PTYPE**. The new process must call the **setptype**() function before it can send or receive messages. The specified process type must be different from the parent's process type and different from the process type of any other process in the same application on the same node.

A process's process type is inherited across an **exec**() function call. If you call the **fork**() function followed by a call to the **exec**() function, you can call the **setptype**() function either before or after the **exec**() function (either **fork(); setptype(); exec();** or **fork(); exec(); setptype();**).

If a process has multiple threads of control, the threads have the same process types. (See the **pthread_create**() function in the *OSF/1 Programmer's Reference* for information on threads.) When a thread is created, it has the same process type as the thread (process) that created it. Do not use the **setptype**() function to set the process type of a thread.

SETPTYPE() *(cont.)* SETPTYPE() *(cont.)*

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 errors that can occur in the C underscore system calls.

Examples

The following example shows a message-passing application that uses the **setptype()** function to set the process type for the calling process:

```
#include <nx.h>
#define MSGTYPE 100

main()
{
   long buf;
   long len;
   long parent_ptype, child_ptype;

   len = sizeof(buf);
   parent_ptype = myptype();
   child_ptype = parent_ptype + 1;

   if (fork() == 0) { /* Child */
      setptype(child_ptype);
      csend(MSGTYPE, &buf, len, mynode(), parent_ptype);
   }
   else { /* Parent */
      csend(MSGTYPE, &buf, len, mynode(), child_ptype);
   }
   crecv(MSGTYPE, &buf, len);
   printf("Node %d, ", mynode());
   printf("ptype %d, msg from node %d, ", myptype(), infonode());
   printf("ptype %d\n", infoptype());
}
```

SETPTYPE() *(cont.)* SETPTYPE() *(cont.)*

The output for this example is as follows:

```
% setptype -sz 1
Node 0, ptype 0 received msg from node 0, ptype 1
Node 0, ptype 1 received msg from node 0, ptype 0
% setptype -sz 2
Node 0, ptype 0 received msg from node 0, ptype 1
Node 0, ptype 1 received msg from node 0, ptype 0
Node 1, ptype 0 received msg from node 1, ptype 1
Node 1, ptype 1 received msg from node 1, ptype 0
```

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

commands: ***application***,

functions: ***errno***, **myptype()**, **nx_load()**, **nx_nfork()**

OSF/1 Programmer's Reference: **exec(2)**, **fork(2)**, **pthread_create(3)**

STATPFS() STATPFS()

statpfs(), **fstatpfs**(): Gets Parallel File System (PFS) statistics.

Synopsis

```
#include <sys/mount.h>
#include <nx.h>
#include <pfs/pfs.h>

long statpfs(
    char *path,
    struct estatfs  *fs_buffer,
    struct statpfs  *pfs_buffer,
    unsigned int  pfs_bufsize );

long fstatpfs(
    int             fildes,
    struct estatfs *fs_buffer,
    struct statpfs *pfs_buffer,
    unsigned int    pfs_bufsize );
```

Parameters

path	Pointer to a pathname of a file within a mounted PFS file system.
fs_buffer	Pointer to a buffer that is an *estatfs* structure in which the status information of the file system is returned. If this value is set to NULL, no status information is returned. The *estatfs* structure is described in the *pfs/pfs.h* header file.
pfs_buffer	Pointer to a buffer that is a *statpfs* structure in which the PFS stripe attributes of the file system are returned. If this value is set to NULL, no PFS information is returned. The *statpfs* structure is described in the *pfs/pfs.h* header file.
pfs_bufsize	Size in bytes of the *pfs_buffer* parameter. If this parameter is 0 (zero), no *statpfs* structure is returned in the *pfs_buffer* parameter.
fildes	File descriptor for an open file within a mounted PFS file system.

STATPFS() *(cont.)*

STATPFS() *(cont.)*

Description

The **statpfs**() and **fstatpfs**() functions return the file system statistics of a mounted file system. If the mounted file system is a PFS file system, stripe attribute information is also returned. Stripe attributes determine how the PFS file system stripes regular files. The file system statistics for the mounted file system are returned in the format of an *estatfs* structure.The stripe attributes are returned in the format of a *statpfs* structure. The *estatfs* and *statpfs* data structures are defined in the *pfs/pfs.h* header file.

Upon successful completion, the **statpfs**() and **fstatpfs**() functions return an *estatfs* structure in the *fs_buffer* parameter. The *estatfs* structure is similar to the *statfs* structure returned by the **statfs**() and **fstatfs**() system calls, except that extended (64-bit) fields are used where appropriate. The *estatfs* structure is specified in the *pfs/pfs.h* header file and has the following form:

```
struct estatfs {
     short       f_type;
     short       f_flags;
     long        f_fsize;
     long        f_bsize;
     esize_t     f_blocks;
     esize_t     f_bfree;
     esize_t     f_bavail;
     long        f_files;
     long        f_ffree;
     mnt_fsid_t  f_fsid;
     long        f_spare[9];
     char        f_mntonname[MNT_MNAMELEN];
     char        f_mntfromname[MNT_MNAMELEN];
};
```

The fields of the *estatfs* structure include the following:

f_type Type of the file system as defined in *sys/mount.h.*

f_flags Copy of the mount flags used when the file system was mounted.

f_fsize File system fragment size. This is the smallest unit of data that is transferred between the file system and the media on which the data is stored. If the file system is of type PFS, this is the fragment size of the file systems containing the stripe data. If the file systems containing the stripe data do not all have the same fragment size, this field is set to -1.

STATPFS() *(cont.)*

STATPFS() *(cont.)*

f_bsize File system block size. This is the optimal unit of data transfer between the file system and the media on which the data is stored. If the file system is of type PFS, this is the block size of the file systems containing the stripe data. If the file systems containing the stripe data do not all have the same block size, this field is set to -1.

f_blocks Total number of data blocks in the file system. If the file system is of type PFS, this is the total number of data blocks available for the stripe data. This field contains an extended (64-bit) value and is expressed in 1K byte units.

f_bfree Number of free blocks in the file system. If the file system is of type PFS, this is the total number of free data blocks available for stripe data. This field contains an extended (64-bit) value and is expressed in 1K byte units.

f_bavail Number of free blocks in the file system available to non-super user. If the file system is of type PFS, this is the total number of free blocks available for stripe data. This field contains an extended (64-bit) value and is expressed in 1K byte units.

f_files Total file nodes in the file system. If the file system is of type PFS, this is the total number of file nodes available in the disk partition that the PFS file system was mounted on.

f_ffree Free file nodes in the file system. If the file system is of type PFS, this is the number of free file nodes in the disk partition that the PFS file system was mounted on.

f_fsid File system identifier.

f_spare Reserved for later use; not used.

f_mntonname Directory on which the file system is mounted.

f_mntfromname Disk partition containing the file system that is mounted on *f_mntonname*.

STATPFS() *(cont.)* STATPFS() *(cont.)*

If the mounted file system is a PFS file system, upon successful completion the **statpfs()** and **fstatpfs()** functions return a *statpfs* structure in the buffer pointed to by the *pfs_buffer* parameter. The *statpfs* structure is of variable length since it contains a variable number of variable length pathnames (see the description of the *p_sdirs* field). To determine if the entire structure fit into the buffer, check the *p_reclen* field. If the entire structure was not received, try again using a buffer of size greater than or equal to the *p_reclen* field. The *statpfs* structure is specified in the *pfs/pfs.h* header file and has the following form:

```
struct statpfs {
     uint_t     p_reclen;
     size_t     p_sunitsize;
     uint_t     p_sfactor;
     pathname_t p_sdirs;
};
```

The fields of the *statpfs* structure include the following:

p_reclen The total length of the *statpfs* structure. If the file system is not of type PFS, then this field is set to 0 (zero).

p_sunitsize The stripe unit size for this parallel file system, in bytes; that is, the size of the unit of data interleaving for regular files.

p_sfactor The number of stripe units per file stripe, that is, the degree of interleaving for regular files.

p_sdirs A list of pathnames specifying the set of directories that define the stripe group for this parallel file system. The number of pathnames in the list is equal to *p_sfactor*. Each pathname is of type *pathname_t*. You can search the pathname list using a pointer of type (*pathname_t* *) and the **NEXTPATH()** macro defined in *pfs/pfs.h*.

To obtain a preallocated array of *statpfs* structures describing the stripe attributes of each currently mounted PFS file system, use the **getpfsinfo()** function. To obtain general mount information for any type of mounted file system, use the standard OSF/1 **statfs()** or **fstatfs()** function.

Return Values

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

STATPFS() *(cont.)* STATPFS() *(cont.)*

Errors

If the **statpfs()** or **fstatpfs()** functions fail, *errno* may be set to one of the values described in the OSF/1 **statfs(2)** manual page.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

getpfsinfo(), **estat()**, **showfs()**

OSF/1 Programmer's Reference: **getmntinfo(3)**, **stat(2)**, **statfs(2)**

TABLE() TABLE()

Examines or updates elements from a system table

Synopsis

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

int table(id, index, addr, nel, lel)
    int id;
    int index;
    char *addr;
    int nel;
    u_int lel;
```

Parameters

id	ID of the system table that contains the elements.
index	Index of an element within the table.
addr	Pointer to a character variable to copy the element values to (on examine) or from (on update).
nel	Signed number that specifies how many elements to copy and in which direction. A positive value requests copying the elements from the kernel to the address *addr*. A negative value copies the elements from the address *addr* to the kernel.
lel	Expected size of a single element.

Description

The **table()** function is used to examine or update one or more elements in a system table. The system table is specified by the *id* parameter and the starting element is specified by index.

The **table()** function copies the element values to or from the address specified by the *addr* parameter. The *nel* parameter specifies the number of elements to copy, starting from the value of the *index* parameter. A positive value indicates an examine operation. The elements are copied from the kernel to address *addr*. A negative value indicates an update operation. The elements are copied from the address *addr* to the kernel.

TABLE() *(cont.)*

TABLE() *(cont.)*

The *lel* parameter specifies the expected element size. If multiple elements are specified, successive addresses are calculated for the *addr* parameter by incrementing it by the value of *lel* for each element copied. If the size of a given element is larger than the *lel* value, the **table()** function truncates excess data on an update (from the address *addr* to the kernel) and stores only the expected size on an examine (from the kernel to address *addr*). If the size of a given element is smaller than the *lel* value, the **table()** function copies only the valid data on an update and pads the element value on an examine.

The **table()** function guarantees that an update operation will not change the offset and size of any field within an element. New fields are added only at the end of an element. The **table()** function returns a count of the elements examined or updated. To determine the actual number of elements in a table before requesting any data, call the **table()** function with the *lel* parameter set to 0 (zero) and the *nel* parameter set to the maximum positive integer. The *id* parameter must specify one of the following tables:

TBL_NODEINFO
The index is by node slot, which is incremented by one for successive elements. Each element is a signed integer that represents a node number. The elements are sorted in ascending order. This table is examine only. It cannot be updated.

TBL_U_TTYD
The controlling terminal device number table. The index is by process ID and exactly one element may be requested. If the process ID is 0 (zero), the current process is indexed. Only 0 and the current process ID are currently supported. The element is of type *dev_t* as defined in the include file *sys/types.h*. This table can be examined only; it cannot be updated.

TBL_UAREA The U-area table. The index is by process ID. See include file *user.h* for the (pseudo) structure *user* that is returned.

TBL_LOADAVG
The system load average vector (pseudo) table. The index must be 0 (zero) and exactly one element may be requested. The element has the following structure:

```
struct tbl_loadavg {
   union {
      long l[3];
      double d[3];
   } tl_avenrun;
   int tl_lscale;
   long tl_mach_factor[3];
};
```

If the scale is 0 (zero), the load average vector is the floating point variant. If the scale is non-zero, the load average vector has been scaled by the indicated factor (typically 1000) to produce the long integer variant. This table can be examined only; it cannot be updated.

TBL_INCLUDE_VERSION

The system include file version number (pseudo) table. The index must be 0 (zero) and exactly one element may be requested. The include version is a unique integer. It identifies the layout of kernel data structures that are imported by certain kernel-dependent programs. This table can be examined only; it cannot be updated.

TBL_ARGUMENTS

The process command argument table containing the saved arguments for processes. The index is by process ID and exactly one element may be requested. Arguments for processes other than the current process can be accessed only by the *root*. This table can be examined only; it cannot be updated.

TBL_MAXUPRC

The maximum process count per user ID table. The index is by process ID and exactly one element may be requested. If the process ID is 0 (zero), the current process is indexed. Only 0 and the current process ID are currently supported. The element is of short integer type. The maximum count includes all processes running under the current user ID even though the limit affects only the current process and any children created with that limit in effect. The limit can be changed only by *root*.

TABLE() *(cont.)* TABLE() *(cont.)*

TBL_PGINFO

The pager information table. The index must be a valid node number to return information about a single node, or -1 to indicate all nodes. This table can be examined only; it cannot be updated. Each element is a *tbl_pginfo_10* structure defined as follows:

```
struct tbl_pginfo_10
{
     unsigned long pg_free;            /* Number of unallocated pages */
     unsigned long pg_npgs;            /* Total number of pages */
     unsigned long pg_pagein_count;    /* Number of page read requests */
     unsigned long pg_pagein_fail;     /* Number of page read errors */
     unsigned long pg_pageout_count;   /* Number of page write requests */
     unsigned long pg_pageout_fail;    /* Number of page write errors */
     unsigned long pg_pageinit_count; /* Number of page initialisations */
     unsigned long pg_pageinit_write; /* Number of " " actually written */
     unsigned long pg_hipage;          /* Highest page number allocated */
     int           pg_type;            /* Type of paging file */
#define PG_KERN_DEFAULT    0          /* Kernel default paging file */
#define PG_VNODE_FILE      1          /* Vnode pager paging file */
#define PG_VNODE_RAWPART   2          /* Vnode pager-paging to raw partition
*/
     char pg_name[PATH_MAX+1];        /* Paging file pathname */
     int         pg_prefer;           /* Preferred paging file*/
     int         pg_node;             /* For vnode pager:the node that
services
                                       * this file/partition
                                       * For kernel default pager: node number
                                       */
};
```

TABLE() *(cont.)*

TABLE() *(cont.)*

TBL_PROCINFO

The process status information table. The index is by system-wide process slot entry number. Status information for processes other than the current process can be accessed only by *root*. This table can be examined only; it cannot be updated. Each element is a *tbl_procinfo* structure defined as follows:

```
#define PI_COMLEN 19 /* length of command name */

struct tbl_procinfo {
   int pi_uid; /* user ID */
   int pi_pid; /* proc ID */
   int pi_ppid; /* parent proc ID */
   int pi_pgrp; /* proc group ID */
   int pi_ttyd; /* controlling terminal number */
   int pi_status; /* process status: */
   #define PI_EMPTY 0 /* - no process */
   #define PI_ACTIVE 1 /* - active process */
   #define PI_EXITING 2 /* - exiting */
   #define PI_ZOMBIE 3 /* - zombie */
   int pi_flag; /* other random flags */
   char pi_comm[PI_COMLEN+1];/*short command name
*/
   int pi_ruid; /* (real) user ID */
   int pi_svuid; /* saved (effective) user ID */
   int pi_rgid; /* (real) group ID */
   int pi_svgid; /* saved (effective) group ID */
   int pi_session; /* session ID */
   int pi_tpgrp; /* tty pgrp */
   int pi_tsession; /* tty session id */
   int pi_jobc; /* # procs qualifying pgrp
                   for job control */
   int pi_cursig;
   int pi_sig; /* signals pending */
   int pi_sigmask; /* current signal mask */
   int pi_sigignore; /* signals being ignored */
   int pi_sigcatch; /* signals being caught by
                       user */
};
```

TBL_ENVIRONMENT

The process environment table. The index is by process ID and exactly one element may be requested. Environment information for processes other than the current process can be accessed only by *root*. This table can be examined only; it cannot be updated.

TABLE() *(cont.)*

TABLE() *(cont.)*

TBL_SYSINFO

The system time information table. The index must be 0 (zero) and exactly one element may be requested. The system information table contains ticks of time accumulated in the various system states: **user**, **nice**, **system**, and **idle**. The system tick frequency and profiling (if configured) frequency are also provided for conversion from ticks to time values. This table can be examined only; it cannot be updated. The element has the following structure:

```
struct tbl_sysinfo {
   long si_user; /* User time */
   long si_nice; /* Nice time */
   long si_sys; /* System time */
   long si_idle; /* Idle time */
   long si_hz; /* System clock ticks per second */
   long si_phz; /* System profile clock (if used)
*/
   long si_boottime; /* Boot time in seconds */
 };
```

TBL_DKINFO The disk statistics table. The index is by disk number. This table can be examined only; it cannot be updated. The element has the following structure:

```
#define DI_NAMESZ 8
struct tbl_dkinfo {
   int di_ndrive; /* Maximum no. of disks providing
                     statistics */
   int di_busy;   /* Bit mask of disks currently
                     busy */
   long di_time; /* Amount of time requested disk
                    is busy */
   long di_seek; /* Number of seeks for requested
                    disk */
   long di_xfer; /* Number of data transfer
                    operations */
   long di_wds;  /* Number of words transferred */
   long di_wpms; /* Words transferred per
                    millisecond */
   int di_unit;  /* The disk unit */
   char di_name[DI_NAMESZ+1]; /* The disk name */
};
```

TABLE() *(cont.)*

TABLE() *(cont.)*

TBL_TTYINFO

The TTY statistics table. The index must be 0 (zero) and exactly one element may be requested. This table can be examined only; it cannot be updated. The element has the following structure:

```
struct tbl_ttyinfo {
   long ti_nin; /* Number of characters input */
   long ti_nout; /* Number of characters output */
   long ti_cancc; /* Portion of input chars on
                     CANNON queue */
   long ti_rawcc; /* Portion of input chars on
                     RAW queue */
};
```

TBL_MSGDS The message queue ID table. The index is the index into the queue array. Each element is a *msqid_ds* structure as defined in **msqid_ds**(). This table can be examined only; it cannot be updated.

TBL_SEMDS The semaphore ID table. The index is the index into the array of semaphore IDs. Each element is a *semid_ds* structure as defined in **semid_ds**(). This table can be examined only; it cannot be updated.

TBL_SHMDS The shared memory region ID table. The index is the index into the array of shared memory region IDs. Each element is a *shmid_ds* structure as defined in shmid_ds(). This table can be examined only; it cannot be updated.

TBL_MSGINFO

The message information table. This table can be examined only; it cannot be updated. The message information structure is defined in the include file *sys/msg.h* as follows:

```
struct msginfo {
   int msgmax; /* max message size */
   int msgmnb; /* max # bytes on queue */
   int msgmni; /* # of message queue identifiers */
   int msgtql; /* # of system message headers */
};
```

The index is by field position within the message information structure as follows:

MSGINFO_MAX
The maximum message size.

MSGINFO_MNB
The maximum number of bytes on the queue.

TABLE() *(cont.)*

TABLE() *(cont.)*

MSGINFO_MNI
The number of message queue IDs.

MSGINFO_TQL
The number of system message headers.

TBL_SEMINFO
The semaphore information table. This table can be examined only; it cannot be updated. The semaphore information structure is defined in the include file *sys/sem.h* as follows:

```
struct seminfo {
   int semmni; /* # of semaphore identifiers */
   int semmsl; /* max # of semaphores per id */
   int semopm; /* max # of operations per semop
                  call */
   int semume; /* maxnumber of undo entries per
                  process */
   int semvmx; /* semaphore maximum value */
   int semaem; /* adjust on exit max value */
};
```

The index is by field position within the semaphore information structure as follows:

SEMINFO_MNI
The number of semaphore IDs.

SEMINFO_MSL
The maximum number of semaphores per ID.

SEMINFO_OPM
The maximum number of operations per the **semop()** call.

SEMINFO_UME
The maximum number of undo entries per process.

SEMINFO_VMX
The semaphore maximum value.

SEMINFO_AEM
The maximum adjust on exit value

TABLE() *(cont.)*

TABLE() *(cont.)*

TBL_SHMINFO

The shared memory information table. This table can be examined only; it cannot be updated. The shared memory information structure is defined in the include *sys/shm.h* as the follows:

```
struct shminfo {
   int shmmax; /* max shared memory segment size */
   int shmmin; /* min shared memory segment size */
   int shmmni; /* number of shared memory
                  identifiers */
   int shmseg; /* max attached shared memory
                  segments per process */
};
```

The index is by field position within the shared memory information structure as follows:

SHMINFO_MAX
The maximum shared memory region size.

SHMINFO_MIN
The minimum shared memory region size.

SHMINFO_MNI
The number of shared memory IDs.

SHMINFO_SEGSHMINFO_SEG
The maximum number of attached shared memory regions per process.

TBL_INTR The system interrupt information table. The system interrupt structure is defined in the include *sys/table.h* as follows:

```
struct tbl_intr {
   long in_devintr; /* Device interrupts
                       (non-clock) */
   long in_context; /* Context switches */
   long in_syscalls; /* Syscalls */
   long in_forks; /* Forks */
   long in_vforks; /* Vforks */
};
```

There is no index into the table. This table can be examined only; it cannot be updated.

TABLE() *(cont.)*

TABLE() *(cont.)*

Return Values

A positive return value indicates that the call succeeded for that number of elements. A return value of -1 indicates that an error occurred, and an error code is stored in the global variable *errno*.

Errors

EFAULT	The *addr* parameter is an invalid address.
EINVAL	The table specified by the *id* parameter is not defined.
EINVAL	The *index* parameter is not valid for the specified table.
EINVAL	The specified table allows only an index of the current process ID with exactly one element. Some other index or element number was specified.
EINVAL	An element length of 0 (zero) was supplied for the TBL_ARGUMENTS table.
EINVAL	An attempt was made to update an examine-only table.
EPERM	An attempt was made to change the maximum number of processes or the account ID, and the caller was not *root*.
ESRCH	The process specified by a process ID index cannot be found.

See Also

Interfaces: **setmodes(1)**, **acct(2)**, **tty(4)**, **acct(5)**

WRITEOFF() WRITEOFF()

Synchronous writes to a file at a specified offset.

Synopsis

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

int writeoff(
        int fildes,
        esize_t offset,
        char *buffer,
        unsigned int nbytes );


#include <sys/types.h>
#include <sys/uio.h>

int writevoff(
        int fildes,
        esize_t offset,
        struct iovec *iov,
        int iovcount );
```

Description of Parameters

fildes	A file descriptor identifying the file to be written to.
offset	Offset from the beginning of the file where to begin the write.
buffer	Pointer to the buffer containing the data is to be written.
nbytes	The number of bytes to write to the file associated with the *fildes* parameter.
iov	Pointer to an array of *iovec* structures that identify the buffers from which the data is to be written.
iovcount	The number of *iovec* structures pointed to by the *iov* parameter.

WRITEOFF() *(cont.)* WRITEOFF() *(cont.)*

Discussion

Writeoff() and **writevoff**() perform the write operation starting at the offset specified by the *offset* parameter.

These functions do not modify the system file pointer(s) (unlike **write**() and **writev**()).

Currently these functions can be used only on files on the Paragon PFS.

Currently only M_UNIX and M_ASYNC I/O modes are supported.

The O_APPEND flag used in the open function to obtain the file descriptor has no effect on the write. The write is performed at the position specified by the *offset* parameter.

Return Values

Upon successful completion, a non-negative integer representing the number of bytes written is returned. If an error occurs, these functions return -1 and set *errno* to indicate the error.

Errors

Errors are as described in OSF/1 **write()**, except that the following errors can also occur:

EFSNOTSUPP The file referred to by *filedes* is not in a file system of a type that supports this operation. Currently only the PFS file systems support this operation.

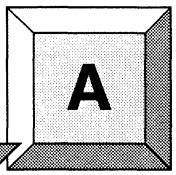
EINVAL The file referred to by *filedes* is in an unsupported iomode. Currently only M_UNIX and M_ASYNC are supported.

See Also

cwrite(), **gopen**(), **iodone**(), **iowait**(), **iseof**(), **iwrite**(), **iwriteoff**(), **niodone**(), **niowait**(), **setiomode**()

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

Message Types and Typesel Masks



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()** functions. 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.

If you use force-type messages with the **csendrecv()** function, you are responsible for posting the receive on the receiving node before the message arrives. Otherwise, the receive will not complete and the message will be lost. The **csendrecv()** function does not do internal synchronization of messages.

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 1 in 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.

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
```

Enter the following in your program code:

```
crecv (0x80001026, buf, len);
```

If you want to receive any message except type `0`, use the following:

```
crecv (0xFFFFFFFE, buf, len);
```

Table A-1 shows the hexadecimal number associated with bits 0-31.

Table A-1. Typesel Mask List (1 of 2)

Type	Hex Number
0	0x00000001
1	0x00000002
2	0x00000004
3	0x00000008
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

Table A-1. Typesel Mask List (2 of 2)

Type	Hex Number
27	0x08000000
28	0x10000000
29	0x20000000
Other types	0x40000000

Errno Manual Page B

This appendix contains the manual page that describes the *errno* global variable, possible *errno* values, and error handling using operating system C system calls.

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 operating system 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.

ERRNO ERRNO

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

Synopsis

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

Description

There are two versions of the operating system 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 operating system 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 operating system 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™ System 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™ System 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™ System 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™ System 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™ System 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™ System 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™ System 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.*
EANXACCT	141	*NX accounting permission denied.*
EAOVLP	141	*Request overlaps with nodes in use.* A partition or application overlaps with another partition or application.

ERRNO (cont.)

ERRNO (cont.)

EAREJPLK	144	*Use of -plk not allowed in gang-scheduled partition.* An application cannot use the **-plk** switch in a gang-scheduled partition.
EBADF	9	*Bad file number.* A socket or file descriptor parameter is invalid.
EBADID	215	*Asynchronous request ID invalid.* The *id* parameter is not a valid I/O ID.
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.
ECFPS	199	*Seek to different file pointers.* Two or more application processes are calling **lseek()** with different shared I/O modes (**M_SYNC**, **M_RECORD**, or **M_GLOBAL**).
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.*

ERRNO (cont.)

ERRNO (cont.)

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.
EEXCEEDCONF	146	*Exceeded allocator configuration parameters.* The application exceeded the configuration parameters for the partition. See the **allocator** manual page.
EFAULT	14	*Bad address.* The requested address is in some way invalid.
EFBIG	27	*File too large.* The file size exceeds the process' file size limit, or the requested semaphore number is invalid. For the **stat()**, **lstat()**, or **fstat()** system call, the file is an extended file (the file size can exceed 2G - 1 bytes). Use the **estat()**, **lestat()**, or **festat()** system call, respectively.
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**, **M_RECORD**, or **M_GLOBAL**.

ERRNO (cont.)

ERRNO (cont.)

EINCOMPAT	216	*The application and the OS are of incompatible revisions.* Your applications code is no longer with the current release of the installed operating system. You must your application.
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.* In **M_SYNC** or **M_GLOBAL** I/O mode, nodes are attempting different operations (reads and writes) to a shared file. In these modes, all nodes must perform the same operation. In the **M_GLOBAL** I/O mode, nodes are attempting different sized reads (using the *nbytes* parameter) from a shared file. See the **setiomode()** function for a description of the I/O modes for file operations.

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.

ERRNO (cont.)

ERRNO (cont.)

ENOLCK	77	*No locks available.* The lock table is full because too many regions are already locked.
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.
ENOSDIR	82	*PFS stripe dir not available.*
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.*
ENOTPFS	212	*Non-striped regular file in a PFS.*

ERRNO (cont.)

ERRNO (cont.)

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.
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.* The application has insufficient access permission on a partition.
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.*
EPFSBUSY	214	*PFS stripe file in use.*
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.* The specified partition (or its parent) does not exist.
EPINVALPRI	134	*Invalid priority.*
EPINVALSCHED	138	*Invalid Scheduling.*

ERRNO (cont.)

ERRNO (cont.)

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.*
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.

ERRNO *(cont.)*

ERRNO *(cont.)*

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.
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.*

ERRNO (cont.)

ERRNO (cont.)

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.
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.

ERRNO (cont.)

ERRNO (cont.)

ERPCMISMATCH	73	*RPC version is wrong.*
ESCHEDCONF	145	*Scheduling parameters conflict with allocator configuration parameters.*The scheduling parameters conflict with the allocator configuration. See the **allocator** manual page.
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.
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.

Limitations and Workarounds

For information about limitations and workarounds, see the release notes files in */usr/share/release_notes*.

See Also

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

Index

L

M

N

O

P

R

S

T

U