April 1996
Order Number: 312488-005

Paragon™ System

Fortran 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™ System Fortran Calls Reference 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 that have been optimized for the Intel supercomputer's architecture.
- Perform 64-bit integer arithmetic (necessary when manipulating PFS file pointers, which can exceed 32 bits).
- Read and write files.

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

See the *Paragon™ System Fortran Compiler User's Guide* for more information about the Fortran interface to the operating system.

NOTE

Programming examples in this manual are intended to demonstrate the use of Paragon Fortran system calls; they are not intended as examples of good programming practice. For example, in some cases, error checks have been omitted in order to make an 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 body of this manual contains a "manual page" for each system call, organized alphabetically. Each manual page provides the following information:

- Synopsis (showing the call's syntax, parameter declarations, and any needed include files).
- Description of any parameters.
- Discussion (may include hints on when and how to use the call).
- Return values (if applicable).
- Error messages (including probable cause and suggested remedy).
- Limitations and workarounds
- Related calls.

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

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

Notational Conventions

This section describes the following notational conventions:

- Type style conventions.
- System call syntax descriptions.

Type Style Conventions

The text of 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. Bold is also used in examples of code to call attention to specific lines.

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 are 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, the syntax of each system call is described in the "Synopsis" section, which contains the following:

- Include files needed by the system call.
- Syntax of the system call.
- Parameter declarations of any 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 'fnx.h'

INTEGER FUNCTION IPROBE(typesel)

INTEGER typesel
```

The **INCLUDE** statement shows only the filename of the include file, not its full pathname. If the include file is not in the default location ("default" according to the **if77** compiler), you must provide a complete pathname, either with the **INCLUDE** statement or with the **-I** option on the **if77** compiler invocation line.

Applicable Documents

For more information, refer to the following documents:

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

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

List of Tables

CPROBE() CPROBE()

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

Synopsis

INCLUDE 'fnx.h'

SUBROUTINE **CPROBE**(*typesel*)

INTEGER *typesel*

SUBROUTINE **CPROBEX**(*typesel*, *nodesel*, *ptypesel*, *info*)

INTEGER *typesel*
INTEGER *nodesel*
INTEGER *ptypesel*
INTEGER *info*(8)

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 Fortran Calls Reference Manual* for more information about message type selectors.
nodesel	Node number of the sender. Setting *nodesel* to -1 probes a message from any node.
ptypesel	Process type of the sender. Setting *ptypesel* to -1 probes a message from any process type.
info	Eight-element array of 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.

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

Description

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

- Use the **cprobe()** subroutine to wait for a message of a specified type.
- Use the **cprobex()** subroutine to wait for a message of a specified type from a specified sender and place information about the message in an array.

When a synchronous call returns, you know that a message of the specified type is available. You can then use one of the receive system calls (for example, the **crecv()** or **irecv()** system calls) to receive the message. With the **cprobe()** subroutine, you can use the **info...()** system calls to get more information about the message.

These are synchronous 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 calls (for example, the **iprobe()** system call).

Examples

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

```
      include 'fnx.h'

      integer      iam, msg_type
      integer      count, node, ptype, type
      character*80 msg, smsg, rmsg
      parameter    (msg_type = 10)

c  Identify self.

      iam = mynode()

c  If node 0, then ...

      if(iam .eq. 0) then
         print *, 'Starting ...'

c    Build message.

         msg = 'Hello from node '
```

CPROBE() *(cont.)*

CPROBE() *(cont.)*

```
            write(smsg, 100) msg, iam
100         format(a16, i3, '.')

c     Send message.

            call csend(msg_type, smsg, len(smsg), -1, myptype())

            write(*, 200) iam, smsg
200         format('Node ', i3, ' sent: ', a20)

c  if not node 0, then ...

          else

c     Probe for message.

            call cprobe(msg_type)

c     Receive message.

            if(infocount() .le. len(rmsg)) then
               call crecv(msg_type, rmsg, len(rmsg))
               count = infocount()
               type  = infotype()
               ptype = infoptype()
               node  = infonode()

c     Report receipt of message.

               write(*, 300) iam, count, type, ptype, node
300            format('Node ', i3,
     1            ' reports ', i3
     2            '-byte message of type ', i2,
     3            ' received from ptype ', i2,
     4            ' on node ', i3, '.')

               write(*, 400) iam, rmsg
400            format('Node ', i3, ' received: ', a30)

            endif

          endif

          end
```

Limitations and Workarounds

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

See Also

crecv(), **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 'fnx.h'

SUBROUTINE CREAD(unit, buffer, nbytes)

INTEGER unit
INTEGER buffer(*)
INTEGER nbytes

SUBROUTINE CREADV(unit, iov, iovcnt)

INTEGER unit
INTEGER iov(*)
INTEGER iovcnt
```

Parameters

unit	Unit number (an integer between 1 and 100) assigned when the file was opened.
buffer	Buffer in which to store the data after it is read from the file. The buffer can be of any valid data type.
nbytes	Size (in bytes) of the *buffer* parameter.
iov	Array of iovec entries that identifies the buffers into which the data is to be placed. An iovec entry is a pair of integers. The first integer contains the address of the buffer. The second integer contains the number of bytes in the buffer.
iovcnt	Number of iovec entries in the *iov* array.

CREAD() *(cont.)*

CREAD() *(cont.)*

Description

The **cread()** and **creadv()** subroutines perform high-speed, synchronous data reads from a file. The **creadv()** subroutine performs the same actions as the **cread()** subroutine, but scatters the input data into the buffers specified by the *iov* parameter.

These calls are synchronous calls. The calling process waits (blocks) until the read completes. To read a file without blocking the calling process, use one of the corresponding asynchronous read calls, either **iread()** or **ireadv()**.

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.

To open a file for reading, use the Fortran **open()** statement with the *form* parameter set to `'unformatted'` or use the **gopen()** subroutine.

For a given file, mixing the operating system read and write calls (for example, **cread()** or **cwrite()**) with the Fortran **read()** and **write()** statements causes an error.

You can automatically create files using a Fortran **read()** or **write()** statement without an **open()** statement. These kind of files are named with the form f*node.unit*, where *node* is the node number and *unit* is the value of the *unit* parameter. These kind of files do not have the correct format for high-speed system reads using **cread()** or **creadv()** subroutines. However, you can read these kind of files with a **read()** statement.

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

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

Errors

NOTE

The majority of the Fortran I/O errors that you are likely to receive are described in the "Runtime Error Messages" appendix of the *Paragon™ System Fortran Compiler User's Guide.* This section describes additional errors that you may receive.

`Attempt to mix standard and PFS I/O calls`

You cannot mix the **cread()** subroutine with Fortran **read()** and **write()** statements on the same file.

`Bad file descriptor`

The *unit* parameter does not specify a valid file unit that is open for reading.

`Invalid argument`

Check arguments.

`I/O error`

Make sure file is open and of the proper format.

`Mixed file operations`

In I/O mode **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.

`No such unit`

The *unit* parameter must be a positive integer no larger than 100.

`Too many open files`

Only 64 files can be open at one time for any process.

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

`Tries to read past EOF`

Attempt was made to read past the end-of-file (EOF).

`Unformatted I/O to FORMATTED file`

Use the Fortran **open()** statement to open the file, setting proper format.

Examples

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

```
      include 'fnx.h'

      integer      iam
      character*13 buf

c  Identify self.

      iam = mynode()

c  Globally open file with the M_UNIX I/O mode

      call gopen(12, '/tmp/mydata', M_UNIX)

c    Read and close the file.

      call cread(12, buf, 13)

      write(*, 100) iam, buf
100   format('Node ', i3, ' read:  ', a13)

      close(12)

      end
```

Limitations and Workarounds

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

CREAD() *(cont.)*

CREAD() *(cont.)*

See Also

cwrite(), gopen(), iread(), iwrite(), iseof(), lseek(), setiomode()

CRECV() CRECV()

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

Synopsis

```
INCLUDE 'fnx.h'

SUBROUTINE CRECV(typesel, buf, count)

INTEGER typesel
INTEGER buf(*)
INTEGER count

SUBROUTINE CRECVX(typesel, buf, count, nodesel, ptypesel, info)

INTEGER typesel
INTEGER buf(*)
INTEGER count
INTEGER nodesel
INTEGER ptypesel
INTEGER info(8)
```

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 Fortran Calls Reference Manual* for more information about message type selectors.
buf	Buffer for storing the received message. The buffer can be of any valid data type, but should match the data type of the buffer in the corresponding send operation.
count	Length (in bytes) of the *buf* parameter.
nodesel	Node number of the message source (that is, the sending node). Setting the *nodesel* parameter to -1 receives a message from any node.

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

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

info Eight-element array of integers in which to store message information. The first four elements contain the message's length, type, 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.

Description

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

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

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

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

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

These are synchronous 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 call (for example, **irecv()** or a handler receive call (for example, **hrecv()**). 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.

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

Errors

`Received message too long for buffer`

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

`Too many requests`

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

Examples

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

```
      include 'fnx.h'

      integer      iam, msg_type
      integer      count, node, pid, type
      character*80 msg, smsg, rmsg
      parameter    (msg_type = 10)

c  Identify self.

      iam = mynode()

c  If node 0, then ...

      if(iam .eq. 0) then
         print *, 'Starting ...'

c    Build message.

         msg = 'Hello from node '
         write(smsg, 100) msg, iam
100      format(a16, i3, '.')
```

CRECV() *(cont.)*

CRECV() *(cont.)*

```
c     Send message.

          call csend(msg_type, smsg, len(smsg), -1, mypid())

          write(*, 200) iam, smsg
200       format('Node ', i3, ' sent: ', a20)

c  if not node 0, then ...

       else

c     Probe for message.

          call cprobe(msg_type)

c     Receive message.

          if(infocount() .le. len(rmsg)) then
             call crecv(msg_type, rmsg, len(rmsg))
             count = infocount()
             type  = infotype()
             pid   = infopid()
             node  = infonode()

c     Report receipt of message.

             write(*, 300) iam, count, type, pid, node
300          format('Node ', i3,
     1          ' reports ', i3
     2          '-byte message of type ', i2,
     3          ' received from PID ', i2,
     4          ' on node ', i3, '.')

             write(*, 400) iam, rmsg
400          format('Node ', i3, ' received: ', a30)

          endif

       endif

       end
```

CRECV() *(cont.)*

CRECV() *(cont.)*

Limitations and Workarounds

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

See Also

cprobe(), **csend()**, **csendrecv()**, **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 'fnx.h'

SUBROUTINE CSEND(type, buf, count, node, ptype)

INTEGER type
INTEGER buf(*)
INTEGER count
INTEGER node
INTEGER ptype
```

Parameters

type	Type of the message to send. Refer to Appendix A of the *Paragon™ System Fortran Calls Reference Manual* for more information about message types.
buf	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 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 (that is, the receiving process).

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

Description

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

The 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.

Examples

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

```
      include 'fnx.h'

      integer      iam, msg_type
      integer      count, node, pid, type
      character*80 msg, smsg, rmsg
      parameter    (msg_type = 10)

c  Identify self.

      iam = mynode()

c  If node 0, then ...

      if(iam .eq. 0) then
         print *, 'Starting ...'

c    Build message.

         msg = 'Hello from node '
         write(smsg, 100) msg, iam
100      format(a16, i3, '.')

c    Send message.

         call csend(msg_type, smsg, len(smsg), -1, mypid())

         write(*, 200) iam, smsg
200      format('Node ', i3, ' sent: ', a20)
```

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

```
c  if not node 0, then ...

      else

c     Probe for message.

         call cprobe(msg_type)

c     Receive message.

         if(infocount() .le. len(rmsg)) then
            call crecv(msg_type, rmsg, len(rmsg))
            count = infocount()
            type  = infotype()
            pid   = infopid()
            node  = infonode()

c     Report receipt of message.

            write(*, 300) iam, count, type, pid, node
300         format('Node ', i3,
     1         ' reports ', i3
     2         '-byte message of type ', i2,
     3         ' received from PID ', i2,
     4         ' on node ', i3, '.')

            write(*, 400) iam, rmsg
400         format('Node ', i3, ' received: ', a30)

         endif

      endif

      end
```

Limitations and Workarounds

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

CSEND() *(cont.)*

CSEND() *(cont.)*

See Also

cprobe(), crecv(), csendrecv(), 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 'fnx.h'

INTEGER FUNCTION CSENDRECV(type, sbuf, scount, node, ptype, typesel,
      rbuf, rcount)

INTEGER type
INTEGER sbuf(*)
INTEGER scount
INTEGER node
INTEGER ptype
INTEGER typesel
INTEGER rbuf(*)
INTEGER rcount
```

Parameters

type	Type of the message to send. Refer to Appendix A of the *Paragon™ System Fortran Calls Reference Manual* for information on message types.
sbuf	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 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).

CSENDRECV() *(cont.)*

CSENDRECV() *(cont.)*

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

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

rcount Number of bytes in the *rbuf* parameter.

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 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 an **isendrecv()** or **hsendrecv()** call (asynchronous calls) instead of a **csendrecv()** call.

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

This call does not affect the information returned by the **info...()** 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 Fortran Calls Reference Manual* for more information on force-type messages.

Return Values

Length (in bytes) of the received message.

Errors

`Invalid argument`

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(), 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 'fnx.h'

SUBROUTINE CWRITE(unit, buffer, nbytes)

INTEGER unit
INTEGER buffer(*)
INTEGER nbytes

SUBROUTINE CWRITEV(unit, iov, iovcnt)

INTEGER unit
INTEGER iov(*)
INTEGER iovcnt
```

Parameters

unit	Unit number (an integer between 1 and 100) assigned when the file was opened.
buffer	Buffer containing data to be written. The buffer can be of any valid data type.
nbytes	Number of bytes to write. The size is limited only by the memory available for the buffer.
iov	Array of iovec entries, which identifies the buffers containing the data to be written.The iovec entry is defined to be a pair of integers. The first integer contains the address of the buffer. The second integer contains the number of bytes in the buffer.
iovcnt	Number of iovec entries in the *iov* array.

CWRITE() *(cont.)*

CWRITE() *(cont.)*

Description

The **cwrite()** and **cwritev()** subroutines perform high-speed, synchronous data writes to a file. The **cwritev()** subroutine performs the same actions as the **cwrite()** subroutine, but gathers the output data from the buffers specified by the *iov* parameter.

These are synchronous calls. The calling process waits (blocks) until the calling process completes. To write a file without blocking the calling process, use the corresponding asynchronous write call (**iwrite()** or **iwritev()**).

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.

To open a file for writing, use the Fortran **open()** statement with the *form* parameter set to `'unformatted'` or use the **gopen()** subroutine.

For a given file, mixing the **cwrite()** or **cwritev()** subroutines with the Fortran **read()** and **write()** statements causes an error.

You can automatically create files using a Fortran **read()** or **write()** statement without an **open()** statement. These kind of files are named with the form *fnode.unit*, where *node* is the node number and *unit* is the value of the *unit* parameter. You can write these kind of files using a **write()** statement. However, these kind of files do not have the correct format for high-speed system writes using the **cwrite()** or **cwritev()** subroutines.

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

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

Errors

NOTE

The majority of the Fortran I/O errors that you are likely to receive are described in the "Runtime Error Messages" appendix of the *Paragon™ System Fortran Compiler User's Guide*. This section describes additional errors that you may receive.

`Attempt to mix standard and PFS I/O calls`

You cannot mix the **cwrite()** subroutine with Fortran **read()** and **write()** statements on the same file.

`Attempt to write to READONLY file`

Check file attributes.

`Bad file descriptor`

The *unit* parameter does not specify a valid file unit that is open for writing.

`Invalid argument`

Check arguments.

`Mixed file operations`

In I/O mode **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.

`No space left on device`

Not enough space on device to which you are writing. Create more space in file system.

`No such unit`

The *unit* parameter must be a positive integer no larger than 100.

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

`Too many open files`

Only 64 files can be open at one time for any process.

`Unformatted I/O to FORMATTED file`

Use the Fortran **open()** statement to open the file, setting proper format.

Examples

The following example globally opens a file with the **gopen()** subroutine and uses the **cwrite()** subroutine to do a synchronous write to the file.

```
      include 'fnx.h'

      integer      iam
      character*13 buf

c  Identify self.

      iam = mynode()

c  Globally open file with the M_UNIX I/O mode

      call gopen(12, '/tmp/mydata', M_UNIX)

c  Write and close the file.

      buf = 'Hello, world!'
      call cwrite(12, buf, len(buf))

      write(*, 100) iam, buf
100   format('Node ', i3, ' wrote: ', a13)

      close(12)

      end
```

Limitations and Workarounds

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

See Also

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

DCLOCK() DCLOCK()

Gets elapsed time in double precision seconds since the node was booted.

Synopsis

INCLUDE 'fnx.h'

DOUBLE PRECISION FUNCTION **DCLOCK**()

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

Elapsed time (in seconds) since booting the node.

Examples

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

```
      include 'fnx.h'

      integer          iam
      double precision stime, etime, dclock

c  Identify self.

      iam = mynode()

c  Get starting time.

      stime = dclock()

c  A delay loop.

      do 10 i=1,1000000
10    continue
```

DCLOCK() *(cont.)* DCLOCK() *(cont.)*

```
c  Calculate elapsed execution time.

      etime = dclock() - stime

c  Display elapsed execution time.

      write(*,100) iam, etime
100   format('Node ', i3, ' elapsed execution time = ', D15.6,
     1       ' seconds.')

      end
```

Limitations and Workarounds

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

See Also

rpm

EADD() EADD()

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

Synopsis

INCLUDE 'fnx.h'

SUBROUTINE **EADD**(*e1*, *e2*, *eresult*)

INTEGER *e1*(2)
INTEGER *e2*(2)
INTEGER *eresult*(2)

INTEGER FUNCTION **ECMP**(*e1*, *e2*)

INTEGER *e1*(2)
INTEGER *e2*(2)

SUBROUTINE **EDIV**(*e*, *n*, *result*)

INTEGER *e*(2)
INTEGER *n*
INTEGER *result*

SUBROUTINE **EMOD**(*e*, *n*, *result*)

INTEGER *e*(2)
INTEGER *n*
INTEGER *result*

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

SUBROUTINE **EMUL**(*e*, *n*, *eresult*)

INTEGER *e*(2)
INTEGER *n*
INTEGER *eresult*(2)

SUBROUTINE **ESUB**(*e1*, *e2*, *eresult*)

INTEGER *e1*(2)
INTEGER *e2*(2)
INTEGER *eresult*(2)

Parameters

e, *e1*, *e2*	Extended integer values, implemented as an array of two integers.
n	Integer value by which an extended integer is multiplied or divided.
eresult	Resulting extended integer.
result	Resulting integer.

Description

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

Use the subprograms to perform mathematical operations on extended integers:

eadd()	Adds an extended integer to another extended integer.
ecmp()	Compares two extended integers.
ediv()	Divides an extended integer by an integer.
emod()	Gets remainder of an extended integer divided by an integer.
emul()	Multiplies an extended integer by an integer.

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

esub()	Subtracts an extended integer by an extended integer.

Return Values

The **ecmp()** system call returns:

-1	If *e1* < *e2*
0	If *e1* = *e2*
1	If *e1* > *e2*

The **eadd()**, **ediv()**, **emod()**, **emul()**, and **esub()** system calls have no return values.

Errors

`Arithmetic overflow`

Quotient should fit in an integer or division by zero for **ediv()** or **emod()** system calls. Result does not fit in an extended integer for the **eadd()**, **emul()**, and **esub()** system calls.

Examples

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

```
      include 'fnx.h'

      integer     n, result
      integer     e1(2), e2(2), eresult(2)
      character*5 string1, string2, string3, string4

c  Identify self.

      iam = mynode()

c  If node 0, then ...

      if(iam .eq. 0) then
         print *,'Starting ...'
```

EADD() *(cont.)*

EADD() *(cont.)*

```
c     Initialize.

          string1 = '3'//char(0)
          string2 = '4'//char(0)
          string3 = '5'//char(0)
          n       =  5

c     Convert strings to extended numbers.

          call stoe(string1, e1)
          call stoe(string2, e2)
          call stoe(string3, eresult)

c     Add e1 and e2.

          call eadd(e1, e2, eresult)
          call etos(eresult, string4)
          write(*, 200) string1, string2, string4
200       format(a10, ' + ', a10, ' = ', a10)

c     Subtract e1 from e2.

          call esub(e1, e2, eresult)
          call etos(eresult, string4)
          write(*, 300) string1, string2, string4
300       format(a10, ' - ', a10, ' = ', a10)

c     Multiply e1 and n.

          call emul(e1, n, eresult)
          call etos(eresult, string4)
          write(*, 400) string1, string3, string4
400       format(a10, ' * ', a10, ' = ', a10)

c     Divide e1 by n.

          call ediv(e1, n, result)
          write(*, 500) string1, string3, result
500       format(a10, ' / ', a10, ' = ', i10)

c     Remainder of e1 divided by n.

          call emod(e1, n, result)
          write(*, 600) string1, string3, result
600       format(a10, 'MOD', a10, ' = ', i10)
```

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

```
c     Compare e1 and e2.

          n = ecmp(e1, e2)
          if(n .lt. 0) then
             write(*, 700) string1, string2
700          format(a10, ' is less than ', a10)
          else if(n .eq. 0) then
             write(*, 710) string1, string2
710          format(a10, ' is equal to ', a10)
          else
             write(*, 720) string1, string2
720          format(a10, ' is greater than ', a10)
          endif

       endif

       end
```

Limitations and Workarounds

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

See Also

eseek(), esize(), etos(), stoe()

ESEEK() ESEEK()

Moves a file's read-write file pointer.

Synopsis

```
INCLUDE 'fnx.h'

SUBROUTINE ESEEK(unit, offset, whence, newpos)

INTEGER unit
INTEGER offset(2)
INTEGER whence
INTEGER newpos(2)
```

Parameters

unit Unit number of an extended or standard OSF/1 file that is open for reading or writing. The unit number is an integer between 1 and 100 that was assigned to the file when it was opened.

offset An extended integer (64-bits) that is the value, in bytes, used by the *whence* parameter to set the file pointer.

whence Specifies how to interpret the *offset* parameter in setting the file pointer associated with the *unit* parameter. Values for the *whence* parameter are as follows:

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 *offset* bytes beyond the end of the file.

newpos An extended integer (64-bits) that is the value, in bytes, for the new position of the file pointer as measured from the beginning of the file.

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

Description

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

The **eseek()** subroutine moves the file pointer in an open extended or standard OSF/1 file specified by the *unit* parameter. You can also use the **lseek()** system call to move the file pointer in a standard OSF/1 file.

Upon successful completion, the **eseek()** subroutine returns an extended integer (*newpos*) that is the new position of the file pointer measured in bytes from the beginning of the file. 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()** subroutine does not successfully complete, it writes an error message on the standard error output and causes the calling process to terminate.

The **eseek()** subroutine allows a file pointer to be set beyond the end of existing data in the file. If data is later written at this point, reading data in the gap returns bytes with the value 0 (zero) until data is actually written into the gap.

The **eseek()** subroutine does not extend the size of the file by itself.

The **eseek()** subroutine may block while asynchronous I/O requests queued by the same process to the same file complete.

Errors

NOTE

The majority of the Fortran I/O errors that you are likely to receive are described in the "Runtime Error Messages" appendix of the *Paragon™ System Fortran Compiler User's Guide*. This section describes additional errors that you may receive.

`File too large`

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.

`Bad file descriptor`

Invalid file unit number.

`Fortran runtime error: Unit not open`

A file must be open to perform a seek operation.

`No such unit`

The *unit* parameter must be a positive integer no larger than 100.

`Mixed file operations`

In I/O mode **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.

`Seek to different file pointers`

In I/O mode **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 modes to the same file position.

Examples

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

```
      include 'fnx.h'

      double precision etime, stime
      integer          newpos(2), newsize(2), offset(2)
      character*10     position, size

c     Open the file /tmp/mydata.

      call gopen(12, '/tmp/mydata', M_UNIX)
```

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

```
c     Set file size to 4,096 bytes (4k bytes).

      call stoe('4096'//char(0), offset)
      call esize(12, offset, SIZE_SET, newsize)

      call etos(newsize, size)
      write(*, 100) size
100   format('New file size is:        ', a10)

c     Move read/write pointer to a location in the file.

      call stoe('500'//char(0), offset)
      call eseek(12, offset, SEEK_SET, newpos)

      call etos(newpos, position)
      write(*, 200) position
200   format('New pointer position is: ', a10)

c     Close the file /tmp/mydata.

      close(12)

      end
```

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(), lseek(), lsize(), setiomode()

ESIZE() ESIZE()

Increases the size of an file.

Synopsis

```
INCLUDE 'fnx.h'

SUBROUTINE ESIZE(unit, offset, whence, newsize)

INTEGER unit
INTEGER offset(2)
INTEGER whence
INTEGER newsize(2)
```

Parameters

unit Unit number of an extended file or standard OSF/1 files open for writing. The unit number is an integer between 1 and 100 that was assigned to the file when it was opened. A standard OSF/1 file cannot have a resulting size greater than 2G - 1 bytes.

offset An extended integer (64-bits) that is the value, in bytes, used by 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 *unit* parameter. Values for the *whence* parameter are as follows:

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.)*

newsize An extended integer (64-bits) that is the new size, in bytes, of the file. If the new size specified by *offset* and *whence* is greater than the available disk space, the **esize()** subroutine allocates all of the available space and returns the new size of the file in the *newsize* parameter.

Description

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

You can use the **esize()** subroutine 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()** subroutine 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 contents of file space allocated by the **esize()** subroutine is undefined.

Using the **esize()** subroutine does not affect the position of the file pointer; use the **eseek()** system call to move the file pointer.

The **esize()** subroutine 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.

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

NOTE

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

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

Errors

NOTE

The majority of the Fortran I/O errors that you are likely to receive are described in the "Runtime Error Messages" appendix of the *Paragon™ System Fortran Compiler User's Guide*. This section describes additional errors that you may receive.

`Bad file descriptor`

Invalid file unit number.

`File too large`

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

`Fortran runtime error: Unit not open`

A file must be open to perform a size operation.

`Invalid argument`

The file is not an extended file.

`No space left on device`

The new size specified by *offset* and *whence* is greater than the available disk space. Create more space in file system.

`Operation not supported for this file system`

The *unit* parameter refers to a file that resides in a file system that does not support this operation. The **esize()** subroutine does not support files that reside in remote file systems and have been NFS mounted.

`Permission denied`

Write access permission to the file was denied.

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

`Read-only file system`

The file resides on a read-only file system.

`Resource temporarily unavailable`

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

Examples

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

```
      include 'fnx.h'

      double precision etime, stime
      integer          newpos(2), newsize(2), offset(2)
      character*10     position, size

c     Open the file /tmp/mydata.

      call gopen(12, '/tmp/mydata', M_UNIX)

c     Set file size to 4,096 bytes (4k bytes).

      call stoe('4096'//char(0), offset)
      call esize(12, offset, SIZE_SET, newsize)

      call etos(newsize, size)
      write(*, 100) size
100   format('New file size is:        ', a10)

c     Move read/write pointer to a location in the file.

      call stoe('500'//char(0), offset)
      call eseek(12, offset, SEEK_SET, newpos)

      call etos(newpos, position)
      write(*, 200) position
200   format('New pointer position is: ', a10)

c     Close the file /tmp/mydata.

      close(12)

      end
```

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

Limitations and Workarounds

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

See Also

cread(), **cwrite()**, **eseek()**, **etos()**, **iread()**, **iwrite()**, **lseek()**, **lsize()**, **stoe()**

ETOS() ETOS()

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

Synopsis

```
INCLUDE 'fnx.h'

SUBROUTINE ETOS(e, s)

INTEGER e(2)
CHARACTER*(*) s

SUBROUTINE STOE(s, e)

CHARACTER*(*) s
INTEGER e(2)
```

Parameters

e	An extended integer.
s	A character string. (For the **stoe()** subroutine, the *s* parameter must be a null terminated.)

Description

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

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

ETOS() *(cont.)* ETOS() *(cont.)*

Errors

`Arithmetic overflow`

Size of the extended integer must be less than 2^{63} -1.

Examples

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

```
      include 'fnx.h'

      integer     n, result
      integer     e1(2), e2(2), eresult(2)
      character*5 string1, string2, string3, string4

c  Identify self.

      iam = mynode()

c  If node 0, then ...

      if(iam .eq. 0) then
         print *,'Starting ...'

c     Initialize.

         string1 = '3'//char(0)
         string2 = '4'//char(0)
         string3 = '5'//char(0)
         n       =  5

c     Convert strings to extended numbers.

         call stoe(string1, e1)
         call stoe(string2, e2)
         call stoe(string3, eresult)
```

ETOS() *(cont.)* ETOS() *(cont.)*

```
c     Add e1 and e2.

          call eadd(e1, e2, eresult)
          call etos(eresult, string4)
          write(*, 200) string1, string2, string4
200       format(a10, ' + ', a10, ' = ', a10)


      endif

      end
```

Limitations and Workarounds

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

See Also

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

FCNTL() FCNTL()

Controls open file descriptors.

Synopsis

INCLUDE 'fnx.h'

CALL **FCNTL** (*unit*, *request*, *<argument>*)

INTEGER *unit*
INTEGER *request*
INTEGER *<argument>*

Parameters

unit Unit number of an extended or standard OSF/1 file that is open for reading or writing. The unit number is an integer between 1 and 100 that was assigned to the file when it was opened.

request Specifies the operation to be performed.

argument Specifies a variable that depends on the value of the *request* parameter.

The following are values for the *request* parameter:

F_GETSATTR Gets the PFS stripe attributes of the file referred to by the *unit* parameter. The *argument* parameter references the structure (*sattr*) in which the stripe attributes are returned. The structure has the following form:

```
STRUCTURE /sattr/
   INTEGER*4 s_sunitsize
   INTEGER*4 s_sfactor
   INTEGER*4 s_start_sdir
END STRUCTURE
```

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

- The file's *stripe unit size*, in bytes. This is the unit of data interleaving used in the PFS file.

FCNTL() *(cont.)*

FCNTL() *(cont.)*

- 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, and 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 <sfactor>-1, inclusive, where <sfactor> 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.

F_SETSATTR Sets the PFS stripe attributes of the file referred to by the *unit* parameter. The *argument* parameter references the structure (*sattr*) which must contain 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 <sfactor>-1 and the stripe factor must be less than or equal to <sfactor>, where <sfactor> is the current stripe factor of the PFS file.

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.)*

Description

The **fcntl()** subroutine gets and sets PFS striping attributes for a supplied file descriptor (*unit*). Calling the subroutine suspends only the calling thread until the request is completed

When used to permanently set the stripe attributes of a file, the **F_SETSATTR** request can only be used 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 *unit* is reset 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 *unit* 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.

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. Note also that for performance reasons, it is advisable to issue the **F_SETSATTR** from only one node, rather than all nodes running the application.

FCNTL() *(cont.)*

FCNTL() *(cont.)*

Errors

NOTE

The majority of the Fortran I/O errors that you are likely to receive are described in the "Runtime Error Messages" appendix of the *Paragon™ System Fortran Compiler User's Guide*. This section describes additional errors that you may receive.

If the **fcntl()** subroutine fails, one of the following error messages appears:

`Bad file number`

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

`File exists`

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

`Not a PFS file`

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

`Invalid Argument`

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.

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

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 'fnx.h'

      INTEGER unit

      STRUCTURE /sattr/
         INTEGER*4 s_sunitsize
         INTEGER*4 s_sfactor
         INTEGER*4 s_start_sdir
      END STRUCTURE

      RECORD /sattr/ sattr

      unit = 12

c Create unformatted file.

      OPEN(unit,
     &     FILE='/pfs/my_file',
     &     STATUS='NEW',
     &     FORM='UNFORMATTED',
     &     ERR=6100)
c Get the default striping attributes and print them

      CALL fcntl(unit, F_GETSATTR, sattr)

      PRINT *,'Here are the default striping attributes.'
      WRITE(*, 100) sattr.s_sunitsize
      WRITE(*, 200) sattr.s_sfactor
      WRITE(*, 300) sattr.s_start_sdir

c Update the sattr structure with the new striping attributes
c so they can be written later.

      sattr.s_sunitsize = 65536
      sattr.s_sfactor = 2
      sattr.s_start_sdir = 2
```

FCNTL() *(cont.)*

FCNTL() *(cont.)*

```
c Set the new stripe attributes

CALL fcntl(unit, F_SETSATTR, sattr)

c Close the file.

      CLOSE(unit,
     &     ERR=6200)

c Re-open the file

      OPEN(unit,
     &     FILE='/pfs/my_file',
     &     STATUS='OLD',
     &     FORM='UNFORMATTED',
     &     ERR=6100)

c Get the new striping attributes

      CALL fcntl(unit, F_GETSATTR, sattr)

      PRINT *,'Here are the new striping attributes.'
      WRITE(*, 100) sattr.s_sunitsize
      WRITE(*, 200) sattr.s_sfactor
      WRITE(*, 300) sattr.s_start_sdir
c Close the file

      CLOSE(unit,
     &     ERR=6200)

      STOP

c Error handling and format statements

 6100 CONTINUE
      PRINT *,'Opening file failed.'

 6200 CONTINUE
      PRINT *,'Closing file failed.'

  100 FORMAT('Stripe Unit size = ',I8.0)
  200 FORMAT('Stripe Factor    = ',I8.0)
  300 FORMAT('Stripe Index     = ',I8.0)

      STOP
      END
```

FCNTL() *(cont.)*

FCNTL() *(cont.)*

See Also

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

FLICK() FLICK()

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

Synopsis

INCLUDE 'fnx.h'

SUBROUTINE **FLICK**()

Description

The **flick()** subroutine 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 **flick()**, control returns to the operating system. For example, if your node program has set up a number of **hrecv()** operations and has nothing else to do, it should issue **flick()**. The operating system can then more efficiently respond to an incoming message and wake up your process.

The **flick()** subroutine does not have any effect on rollin and rollout of the application.

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

- If the calling process is the only process on the node, **flick()** 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, **flick()** 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.

Limitations and Workarounds

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

FORCEFLUSH() FORCEFLUSH()

Flushes all buffered I/O when an exception occurs.

Synopsis

INCLUDE 'fnx.h'

SUBROUTINE **FORCEFLUSH**()

Description

The **forceflush()** subroutine enables a signal handler that flushes all buffered I/O before an application terminates with an exception. Use this subroutine with **fpsetmask()** to flush all I/O in the event of a floating-point exception.

A program must use **forceflush()** *before* an exception occurs.

Writes to the terminal are not buffered.

Examples

The following example shows how to use the **forceflush()** subroutine to flush all buffered I/O for an application:

```
      include 'fnx.h'

      integer   a, newmask, oldmask, r
      character   list(100)

c  Set floating-point exception mask:
c    1     Enables invalid operation exceptions.
c    2     Enables denormalization exceptions.
c    4     Enables divide-by-zero exceptions.
c    8     Enables overflow exceptions.

      newmask = 1 + 2 + 4 + 8
      oldmask = fpsetmask(newmask)
```

FORCEFLUSH() *(cont.)* FORCEFLUSH() *(cont.)*

```
c  Display old and new floating-point exception masks.

      write(*, 100) oldmask, newmask
100   format('Old mask was: ', i10, /, 'New mask is:  ', i10)

c  Ensure that all I/O is flushed if an exception occurs.

      call forceflush()

c  Build list of 100 charcters.

      do 1, i = 1, 100
         list(i) = 'b'
1     continue

c  Open the file named "/tmp/mydata"

      call gopen(12, '/tmp/mydata', M_UNIX)
      end file(12)

c  Write to the file.

      write(12) (list(i), i = 1, 100)

c  The next statement causes a divide-by-zero exception.

      r = 0
      a = 1/r

      close(12)

      end
```

Limitations and Workarounds

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

See Also

fpsetmask(), forflush()

FORFLUSH() FORFLUSH()

Forces completion of all buffered I/O to a specified file.

Synopsis

INCLUDE 'fnx.h'

SUBROUTINE **FORFLUSH**(*unit*)

INTEGER *unit*

Parameters

unit	The unit number (an integer between 1 and 100) assigned when the file was opened.

Description

The **forflush()** subroutine forces all buffered I/O to the file identified by *unit*.

Errors

NOTE

The majority of the Fortran I/O errors that you are likely to receive are described in the "Runtime Error Messages" appendix of the *Paragon™ System Fortran Compiler User's Guide*. This section describes additional errors that you may receive.

`Bad file descriptor`

Invalid file unit number.

FORFLUSH() *(cont.)* FORFLUSH() *(cont.)*

Examples

The following example shows how to use the **forceflush()** subroutine to force completion of all buffered I/O for an application:

```
      include 'fnx.h'

      character list(100)

c  Build list of 100 characters.

      do 1, i = 1, 100
         list(i) = 'a'
1     continue

c  Open the file named "/tmp/mydata"

      call gopen(12, '/tmp/mydata', M_UNIX)
      end file(12)

c  Write to the file.

      write(12) (list(i), i = 1, 100)

c  Flush any buffered I/O.

      call forflush(12)

      close(12)

      end
```

Limitations and Workarounds

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

See Also

forceflush()

FPSETMASK() FPSETMASK()

Sets the floating-point exception mask.

Synopsis

INCLUDE 'fnx.h'

INTEGER FUNCTION **FPSETMASK**(*mask*)

INTEGER *mask*

Parameters

mask An arithmetic value that enables or disables floating-point exceptions:

0	Disable all floating-point exceptions (default).
1	Enable invalid operation exception.
2	Enable divide-by-zero exception.
4	Enable overflow exception.
8	Enable underflow exception.
16	Enable imprecise (loss of precision) exception.

Description

The **fpsetmask()** function sets the floating-point exception mask.

Return Values

The previous value of the *mask* parameter.

FPSETMASK() *(cont.)* FPSETMASK() *(cont.)*

Examples

The following example shows how to use the **fpsetmask()** function to set a floating-point exception mask:

```
      include 'fnx.h'

      integer   a, newmask, oldmask, r
      character   list(100)

c  Set floating-point exception mask:
c     1     Enables invalid operation exceptions.
c     2     Enables divide-by-zero exceptions.
c     4     Enables overflow exceptions.
c     8     Enables underflow exceptions.

      newmask = 1 + 2 + 4 + 8
      oldmask = fpsetmask(newmask)

c  Display old and new floating-point exception masks.

      write(*, 100) oldmask, newmask
100   format('Old mask was: ', i10, /, 'New mask is:  ', i10)

c  Ensure that all I/O is flushed if an exception occurs.

      call forceflush()

c  Build list of 100 charcters.

      do 1, i = 1, 100
         list(i) = 'b'
1     continue

c  Open the file named "/tmp/mydata"

      call gopen(12, '/tmp/mydata', M_UNIX)
      end file(12)

c  Write to the file.

      write(12) (list(i), i = 1, 100)
```

```
c  The next statement causes a divide-by-zero exception.

      r = 0
      a = 1/r

      close(12)

      end
```

Limitations and Workarounds

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

See Also

forceflush()

GCOL() GCOL()

Collects contributions from all nodes. (Global concatenation operation)

Synopsis

INCLUDE 'fnx.h'

SUBROUTINE **GCOL**(*x*, *xlen*, *y*, *ylen*, *ncnt*)

INTEGER *x*(*)
INTEGER *xlen*
INTEGER *y*(*)
INTEGER *ylen*
INTEGER *ncnt*

Parameters

x	Input buffer to be used in the operation. Note that *x* may be of any type.
xlen	Length (in bytes) of *x*.
y	Output buffer to be used in the operation. Note that *y* must be of the same data type as *x*.
ylen	Length (in bytes) of *y*.
ncnt	Number of bytes returned in *y*.

Description

The **gcol()** subroutine 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.

GCOL() *(cont.)* GCOL() *(cont.)*

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

This is a "global" operation. 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.

Examples

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

```
      include 'fnx.h'

      integer          count, dpsize, iam, xsize, ysize, nbrnodes
      parameter          (xsize = 4)
      parameter          (ysize = 16)
      double precision x(xsize), y(ysize), dot, norm, work
      character*80     msg

c  Initialize.

      count    = 0
      dpsize   = 8
      dot      = 0.0
      nbrnodes = numnodes()

c  Identify self.

      iam = mynode()

      if(iam .eq. 0) print *, 'Starting ...'

c  Each node creates and displays its four-element vector.

      do 1, i = 1, xsize
         x(i) = iam * (xsize) + i-1
         write(*, 100) iam, i, x(i)
100      format('Node ', i1, ' x(', i1, ') = ', f3.1)
1     continue
```

GCOL() *(cont.)* GCOL() *(cont.)*

```
c  Each node calculates and displays its dot product.

      do 2, i = 1, xsize
         dot = dot + x(i)*x(i)
2     continue
      write(*, 200) iam, dot
200   format('Node ', i1, ' dot = ', f10.6)

c  Each node sums the dot products of all nodes.

      call gdsum(dot, 1, work)

c  Node 0 displays the resulting dot product.

      if(iam .eq. 0) write(*, 300) dot
300   format('dot = ', f10.6)

c  Each node normalizes its dot products.

      norm = dsqrt(dot)
      do 3, i = 1, xsize
         x(i) = x(i)/norm
3     continue

c  Each node collects contributions from other node.

     call gcol(x, xsize*dpsize, y, nbrnodes*xsize*dpsize, count)

c  Node 0 displays the resulting vector.

      if(iam .eq. 0) then
         do 4, i = 1, nbrnodes*xsize
            write(*, 400) i, y(i)
400         format('y(', i1, ') = ', f3.1)
4        continue

      endif

      end
```

Limitations and Workarounds

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

GCOL() *(cont.)*

GCOL() *(cont.)*

See Also

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

GCOLX() GCOLX()

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

Synopsis

```
INCLUDE 'fnx.h'

SUBROUTINE GCOLX(x, xlens, y)

INTEGER x(*)
INTEGER xlens(*)
INTEGER y(*)
```

Parameters

x	Input buffer to be used in the operation. This parameter may be of any type.
xlens	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	Output buffer to be used in the operation. This parameter must be of the same data type as *x*.

Description

The **gcolx()** subroutine 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()**. This is 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. 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.)*

Examples

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

```
      include 'fnx.h'

      integer          count, dpsize, iam, xsize, ysize, nbrnodes
      parameter          (xsize = 4)
      parameter          (ysize = 16)
      integer            xlen(xsize)
      double precision x(xsize), y(ysize), dot, norm, work
      character*80       msg

c  Initialize.

      count    = 0
      dpsize   = 8
      dot      = 0.0
      nbrnodes = numnodes()
      do 1, i = 1, nbrnodes
         xlen(i) = xsize*dpsize
1     continue

c  Identify self.

      iam = mynode()
      if(iam .eq. 0) print *, 'Starting ...'

c  Each node creates and displays its four-element vector.

      do 2, i = 1, xsize
         x(i) = iam * (xsize) + i-1
         write(*, 100) iam, i, x(i)
100      format('Node ', i1, ' x(', i1, ') = ', f3.1)
2     continue

c  Each node calculates and displays its dot product.

      do 3, i = 1, xsize
         dot = dot + x(i)*x(i)
3     continue
      write(*, 200) iam, dot
200   format('Node ', i1, ' dot = ', f10.6)
```

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

```
c  Each node sums the dot products of all nodes.

      call gdsum(dot, 1, work)

c  Node 0 displays the resulting dot product.

      if(iam .eq. 0) write(*, 300) dot
300   format('dot = ', f10.6)

c  Each node normalizes its dot products.

      norm = dsqrt(dot)
      do 4, i = 1, xsize
         x(i) = x(i)/norm
4     continue

c  Each node collects contributions from other node.

      call gcolx(x, xlen, y)

c  Node 0 displays the resulting vector.

      if(iam .eq. 0) then
         do 5, i = 1, nbrnodes*xsize
            write(*, 400) i, y(i)
400         format('y(', i1, ') = ', f3.1)
5        continue

      endif

      end
```

Limitations and Workarounds

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

See Also

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 'fnx.h'

SUBROUTINE GDHIGH(x, n, work)

DOUBLE PRECISION x(*)
INTEGER n
DOUBLE PRECISION work(*)

SUBROUTINE GIHIGH(x, n, work)

INTEGER x(*)
INTEGER n
INTEGER work(*)

SUBROUTINE GSHIGH(x, n, work)

REAL x(*)
INTEGER n
REAL work(*)
```

Parameters

x	Array to use in the operation. When the operation completes, *x* contains the final result.
n	Number of elements in *x*.
work	Array that receives the contributions from other nodes. The number of elements in *work* must be at least *n*.

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

Description

Use one of the following subroutines to determine a maximum value across all 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 real 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. 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.

Examples

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

```
      include 'fnx.h'

      integer*4 iam, max, maxval, min, minval, size, seed, work
      parameter (size = 10)
      integer   list(size)

c  Identify self.

      iam = mynode()
      if(iam .eq. 0) print *, 'Starting ...'

c  Each node creates list of random integers in the range 0..100.

      seed = (2 * mclock()/(iam+1)) + 1
      do 1, i = 1, size
         list(i) = int(100*ran(seed))
1     continue

      write(*, 100) iam, (list(i), i = 1, size)
100   format('List for node ', i3, ' is: ', 10i4)
```

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

```
c  Each node finds smallest and largest values in its list.

      min = 1
      max = 1
      do 2, i = 2, size
         if(list(i) .gt. list(max)) max = i
         if(list(i) .lt. list(min)) min = i
2     continue
      minval = list(min)
      maxval = list(max)

c  Each node finds smallest and largest values across all nodes.

      call gilow(minval, 1, work)
      call gihigh(maxval, 1, work)

c  Node 0 displays global minimum and maximum values.

      if(iam .eq. 0) write(*, 200) minval, maxval
200   format('Minimum value is ', i4, /, 'Maximum value is ', i4)

      end
```

Limitations and Workarounds

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

See Also

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 'fnx.h'

SUBROUTINE GDLOW(x, n, work)

DOUBLE PRECISION x(*)
INTEGER n
DOUBLE PRECISION work(*)

SUBROUTINE GILOW(x, n, work)

INTEGER x(*)
INTEGER n
INTEGER work(*)

SUBROUTINE GSLOW(x, n, work)

REAL x(*)
INTEGER n
REAL work(*)
```

Parameters

x	Array to use in the operation. When the operation completes, *x* contains the final result.
n	Number of elements in *x*.
work	Array that receives the contributions from other nodes. The number of elements in *work* must be at least *n*.

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

Description

Use one of the following subroutines to determine a minimum value across all 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 real 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. 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.

Examples

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

```
      include 'fnx.h'

      integer*4 iam, max, maxval, min, minval, size, seed, work
      parameter (size = 10)
      integer   list(size)

c  Identify self.

      iam = mynode()
      if(iam .eq. 0) print *, 'Starting ...'

c  Each node creates list of random integers in the range 0..100.

      seed = (2 * mclock()/(iam+1)) + 1
      do 1, i = 1, size
         list(i) = int(100*ran(seed))
1     continue

      write(*, 100) iam, (list(i), i = 1, size)
100   format('List for node ', i3, ' is: ', 10i4)
```

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

```
c  Each node finds smallest and largest values in its list.

      min = 1
      max = 1
      do 2, i = 2, size
         if(list(i) .gt. list(max)) max = i
         if(list(i) .lt. list(min)) min = i
2     continue
      minval = list(min)
      maxval = list(max)

c  Each node finds smallest and largest values across all nodes.

      call gilow(minval, 1, work)
      call gihigh(maxval, 1, work)

c  Node 0 displays global minimum and maximum values.

      if(iam .eq. 0) write(*, 200) minval, maxval
200   format('Minimum value is ', i4, /, 'Maximum value is ', i4)

      end
```

Limitations and Workarounds

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

See Also

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 'fnx.h'

SUBROUTINE GDPROD(x, n, work)

DOUBLE PRECISION x(*)
INTEGER n
DOUBLE PRECISION work(*)

SUBROUTINE GIPROD(x, n, work)

INTEGER x(*)
INTEGER n
INTEGER work(*)

SUBROUTINE GSPROD(x, n, work)

REAL x(*)
INTEGER n
REAL work(*)
```

Parameters

x	Array to use in the operation. When the operation completes, *x* contains the final result.
n	Number of elements in the array the *x* parameter specifies.
work	Array that receives the contributions from other nodes. The number of elements in *work* must be at least the value of the *n* parameter.

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

Description

Use one of the following subroutines to calculate a product across all 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 real 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. 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.

Examples

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

```
      include 'fnx.h'

      integer*4 iam, size, seed
      parameter (size = 5)
      integer   list(size), work(size)

c  Identify self.

      iam = mynode()
      if(iam .eq. 0) print *, 'Starting ...'

c  Create list of random integers in the range 0..10.

      seed = (2 * mclock()/(iam+1)) + 1
      do 1, i = 1, size
         list(i) = int(10 * ran(seed))
1     continue

      write(*, 100) iam, (list(i), i = 1, size)
100   format('List for node ', i3, ' is: ', 5i6)
```

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

```
c  Perform multiplication across all nodes.

      call giprod(list, size, work)

c  If node 0, display resulting vector.

      if(iam .eq. 0) write(*, 200) (list(i), i = 1, size)
200   format('Resulting list is:    ', 5i6)

      end
```

Limitations and Workarounds

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

See Also

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 'fnx.h'

SUBROUTINE GDSUM(x, n, work)

DOUBLE PRECISION x(*)
INTEGER n
DOUBLE PRECISION work(*)

SUBROUTINE GISUM(x, n, work)

INTEGER x(*)
INTEGER n
INTEGER work(*)

SUBROUTINE GSSUM(x, n, work)

REAL x(*)
INTEGER n
REAL work(*)
```

Parameters

x	Array to use in the operation. When the operation completes, *x* contains the final result.
n	Number of elements in *x*.
work	Array that receives the contributions from other nodes. The number of elements in *work* must be at least *n*.

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

Description

Use one of the following subroutines to calculate a sum across all 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 real 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. 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.

Examples

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

```
      include 'fnx.h'

      integer          count, dpsize, iam, xsize, ysize, nbrnodes
      parameter          (xsize = 4)
      parameter          (ysize = 16)
      double precision x(xsize), y(ysize), dot, norm, work
      character*80     msg

c     Initialize.

      count    = 0
      dpsize   = 8
      dot      = 0.0
      nbrnodes = numnodes()

c     Identify self.

      iam = mynode()

      if(iam .eq. 0) print *, 'Starting ...'
```

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

```
c  Each node creates and displays its four-element vector.

      do 1, i = 1, xsize
         x(i) = iam * (xsize) + i-1
         write(*, 100) iam, i, x(i)
100      format('Node ', i1, ' x(', i1, ') = ', f3.1)
1     continue

c  Each node calculates and displays its dot product.

      do 2, i = 1, xsize
         dot = dot + x(i)*x(i)
2     continue
      write(*, 200) iam, dot
200   format('Node ', i1, ' dot = ', f10.6)

c  Each node sums the dot products of all nodes.

      call gdsum(dot, 1, work)

c  Node 0 displays the resulting dot product.

      if(iam .eq. 0) write(*, 300) dot
300   format('dot = ', f10.6)

c  Each node normalizes its dot products.

      norm = dsqrt(dot)
      do 3, i = 1, xsize
         x(i) = x(i)/norm
3     continue

c  Each node collects contributions from other node.

     call gcol(x, xsize*dpsize, y, nbrnodes*xsize*dpsize, count)
```

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

```
c  Node 0 displays the resulting vector.

      if(iam .eq. 0) then
         do 4, i = 1, nbrnodes*xsize
            write(*, 400) i, y(i)
400         format('y(', i1, ') = ', f3.1)
4        continue

      endif

      end
```

Limitations and Workarounds

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

See Also

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

GIAND() GIAND()

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

Synopsis

INCLUDE 'fnx.h'

SUBROUTINE **GIAND**(*x*, *n*, *work*)

INTEGER *x*(*)
INTEGER *n*
INTEGER *work*(*)

SUBROUTINE **GLAND**(*x*, *n*, *work*)

LOGICAL *x*(*)
INTEGER *n*
LOGICAL *work*(*)

Parameters

x	Array to use in the operation. When the operation completes, *x* contains the final result.
n	Number of elements in *x*.
work	Array that receives the contributions from other nodes. The number of elements in *work* must be at least *n*.

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

Description

Use one of the following subroutines to perform an AND operation 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. 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.

Examples

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

```
      include 'fnx.h'

      integer iam
      integer x(5), work(5)

c  Identify self.

      iam = mynode()

c  Node 0 builds simple vector.

      if(iam .eq. 0) then

         print *, 'Starting ...'

         do 1, i = 1, 5
            x(i) = i
1        continue

         write(*, 100) (x(i), i = 1, 5)
100      format('Vector is:                              ', 5i6)
```

```
      else

c  Node 1 builds bitwise-complement of Node 0's vector.

         do 2, i = 1, 5
            x(i) = inot(i)
2        continue

         write(*, 300) (x(i), i = 1, 5)
300      format('Complement Vector is:                ', 5i6)

      endif

c  Perform bitwise AND.

      call giand(x, 5, work)

c  Node 0 displays resulting vector.

      if (iam .eq. 0 ) write(*, 200) (x(i), i = 1, 5)
200   format('Vector ANDed with its complement is: ', 5i6)

      end
```

Limitations and Workarounds

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

See Also

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

GIOR() GIOR()

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

Synopsis

```
INCLUDE 'fnx.h'

SUBROUTINE GIOR(x, n, work)

INTEGER x(*)
INTEGER n
INTEGER work(*)

SUBROUTINE GLOR(x, n, work)

LOGICAL x(*)
INTEGER n
LOGICAL work(*)
```

Parameters

x	Array to use in the operation. When the operation completes, *x* contains the final result.
n	Number of elements in *x*.
work	Array that receives the contributions from other nodes. The number of elements in *work* must be at least *n*.

Description

Use one of the following subroutines to perform an OR operation 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.

GIOR() *(cont.)* GIOR() *(cont.)*

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. 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.

Examples

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

```
      include 'fnx.h'

      integer iam
      integer x(5), work(5)

c  Identify self.

      iam = mynode()

c  Node 0 builds simple vector.

      if(iam .eq. 0) then

         print *, 'Starting ...'

         do 1, i = 1, 5
            x(i) = i
1        continue

         write(*, 100) (x(i), i = 1, 5)
100      format('Original Vector:                              ', 5i6)

      else

c  Node 1 builds vector containing all ones.

         do 2, i = 1, 5
            x(i) = inot(0)
2        continue
```

```
         write(*, 300) (x(i), i = 1, 5)
300     format('Vector containing all ones:                ', 5i6)

      endif

c  Perform exclusive OR.

      call gior(x, 5, work)

c  Display resulting vector.

      if (iam .eq. 0) write(*, 200) (x(i), i = 1, 5)
200   format('Vector exclusive ORed with all-ones vector: ', 5i6)

      end
```

Limitations and Workarounds

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

See Also

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 operation.

Synopsis

```
INCLUDE 'fnx.h'

SUBROUTINE GOPEN(unit, path, iomode)

INTEGER unit
CHARACTER *(*) path
INTEGER iomode
```

Parameters

unit Unit number (an integer between 1 and 100) to be assigned to the file being opened. An existing file must be a Fortran unformatted file.

path String containing the pathname for the file being opened or created. If the path parameter refers to a symbolic link, the **gopen()** subroutine opens the file pointed to by the symbolic link.

iomode I/O mode for the file to be opened. This parameter can have the following values:

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.

GOPEN() *(cont.)*

GOPEN() *(cont.)*

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.

Description

The **gopen()** subroutine optimizes the standard Fortran **open()** statement so all nodes can open and share a file. The **gopen()** subroutine performs a *global* open; all nodes can open the same file without issuing multiple I/O requests.

You can use the **gopen()** subroutine 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()** subroutine to change a unit's I/O mode after the unit is opened. Use the **iomode()** function to return a unit's current I/O mode.

This call performs a global synchronization of all nodes in the application's partition. That is, all nodes must call the **gopen()** subroutine 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.

The **gopen()** subroutine opens a new file as a Fortran unformatted file. Using the **gopen()** subroutine to open an existing Fortran formatted file causes a format-conflict error when you try to write to the file.

Errors

NOTE

The majority of the Fortran I/O errors that you are likely to receive are described in the "Runtime Error Messages" appendix of the *Paragon™ System Fortran Compiler User's Guide*. This section describes additional errors that you may receive.

`Bad I/O mode number`

The *iomode* parameter is set to a invalid I/O mode number.

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

`File pathname not consistent.`

The pathname is not consistent between the nodes.

`Formatted/unformatted file conflict`

You opened an existing Fortran formatted file and tried to write to the file.

`I/O mode value not consistent.`

The I/O mode value is not consistent between the nodes.

`Invalid argument`

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

Examples

The following example globally opens a file with the **gopen()** subroutine and writes to the file.

```
      include 'fnx.h'

      integer      iam
      character*13 buf

c  Identify self.

      iam = mynode()

c  Globally open file with the M_UNIX I/O mode

      call gopen(12, '/tmp/mydata', M_UNIX)
```

```
c  Write and close the file.

      buf = 'Hello, world!'
      call cwrite(12, buf, len(buf))

      write(*, 100) iam, buf
100   format('Node ', i3, ' wrote: ', a13)

      close(12)

      end
```

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()

Paragon™ System Fortran Compiler User's Guide: **open()**

GOPF() GOPF()

Makes a global operation of a user-defined function.

Synopsis

```
INCLUDE 'fnx.h'

SUBROUTINE GOPF(x, xlen, work, name)

INTEGER x(*)
INTEGER xlen
INTEGER work(*)
EXTERNAL name
```

Parameters

x	Array to use in the operation. Note that *x* can be any type. When the operation completes, *x* contains the final result.
xlen	Length (in bytes) of *x*.
work	Array that receives the contributions from other nodes. The length of *work* must be at least *xlen*.
name	The user-defined function to be called. The function is defined separately. The function *name* 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()** subroutine gives a user-defined function the same global properties as system-defined global communications routines (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.

GOPF() *(cont.)*

GOPF() *(cont.)*

- Each node calculates the result and stores it in the *x* buffer.
- The *work* array receives contributions from other nodes.
- The result is returned in *x* to all nodes.

Examples

In the following example, each node constructs a short list of random integers. After constructing the lists, the global function *max_node* is called. Each node executes this function which determines the largest integer from all lists. The code follows:

```
program test

      include 'fnx.h'

      external max_node

      integer*4 seed, size, iam, max_node
      parameter (size = 5)
      integer*4 list(size)
      integer*4 mine(3), work(3)

c  Identify self.

      iam = mynode()
      if(iam .eq. 0) print *, 'Starting ...'

c  Create list of random integers in the range 0...100.


      seed = (2 * mclock()/(iam+1)) + 1
      do 1, i = 1, size
         list(i) = int(100 * ran(seed))
1     continue

      write(*, 100) iam, (list(i), i = 1, size)
100   format('List for Node ', i3, ' is: ', 10i4)

c  Determine: mine(1) -- maximum absolute value in list.
c             mine(2) -- associated element number.
c             mine(3) -- associated node number.
```

GOPF() *(cont.)*

GOPF() *(cont.)*

```
      mine(1) = abs(list(1))
      mine(2) = 1
      mine(3) = iam
      do 2, i = 2, size
         if(mine(1) .lt. abs(list(i))) then
             mine(1) = abs(list(i))
             mine(2) = i
         endif
2     continue

c  Call gopf() to determine maximum value across all nodes.

      call gopf(mine, 12, work, max_node)

c  Display maximum value and associated element and node numbers.

      if(iam .eq. 0) write(*, 200) mine(1), mine(2), mine(3)
200   format(/, 'Maximum value is ', i3, ' in element ', i3,
     1       ' on Node ', i3)

      end


      integer function max_node(mine, work)

      integer*4 mine(3), work(3)

      if(mine(1) .lt. work(1) .or.
     1  (mine(1) .eq. work(1) .and. mine(3) .gt. work(3)) )
     2  then
          mine(1) = work(1)
          mine(2) = work(2)
          mine(3) = work(3)
      endif

      return

      end
```

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

Limitations and Workarounds

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

See Also

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

GSENDX() GSENDX()

Sends a message to a list of nodes.

Synopsis

```
INCLUDE 'fnx.h'

SUBROUTINE GSENDX(type, buf, count, nodes, nodecount)

INTEGER type
INTEGER buf(*)
INTEGER count
INTEGER nodes(*)
INTEGER nodecount
```

Parameters

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

Description

The **gsendx()** subroutine 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()**. All participating processes must have the same process type.

GSENDX() *(cont.)*

GSENDX() *(cont.)*

Examples

The following example shows how to use the **gsendx()** subroutine to send a message to a list of nodes in an application:

```
      include 'fnx.h'

      integer*4 iam
      integer*4 nodenums(2), x(10), y(10)

c  Initialize.

      do 1, i = 1, 10
         x(i) = 0
         y(i) = 0
1     continue

c  Identify self.

      iam = mynode()

c  If node 0, then ...

      if (iam .eq. 0) then

         print *, 'Starting ...'

c    Build list to send.

         do 2, i = 1, 10
            x(i) = i
2        continue

c    Specify receiving node numbers.

         nodenums(1) = 1
         nodenums(2) = 3

c    Send list to receiving nodes.

         call gsendx(100, x, 10*4, nodenums, 2)

         write(*, 100) iam, (x(i), i = 1, 10)
100      format('List sent by     Node ', i3, ' is: ', 10i4)
```

GSENDX() *(cont.)* GSENDX() *(cont.)*

```
c  If not node 0, then ...

      else

c    Receive the list.

         call crecv(100, y, 10*4)
c         if (iam .ne. 2) call crecv(100, y, 10*4)

c    Display the received list.

         write(*, 200) iam, (y(i), i = 1, 10)
200      format('List received by Node ', i3, ' is: ', 10i4)

      endif

      end
```

Limitations and Workarounds

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

See Also

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

GSYNC() GSYNC()

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

Synopsis

INCLUDE 'fnx.h'

SUBROUTINE **GSYNC**()

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.

Examples

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

```
      include 'fnx.h'

      parameter (MAX_IDS = 900)
      integer   n, node, my_node, num_nodes
      character*10 sbuf(10), rbuf(10)
      integer   rmid(0:MAX_IDS-1)

      my_node = mynode()
      num_nodes = numnodes()

      if(my_node .eq. 0) then
         print *,'Starting ...'
      endif

c  Post receives.

      do 1 n = 0,num_nodes - 1
        rmid(n) = irecv(1, rbuf, MBUF_LEN)
    1 continue
```

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

```
c  Send a message to each node.

      do 2 node = 0,num_nodes - 1
        call csend(1, sbuf, MBUF_LEN, node, 0)
    2 continue

c  Check received messages.

      do 3 n = 0,num_nodes - 1
        call msgwait(rmid(n))
    3 continue

c    Wait for all nodes to complete.

      call gsync()


      if (my_node .eq. 0) then
         print *,'Finished!'
      endif

      end
```

Limitations and Workarounds

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

See Also

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

HRECV() HRECV()

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

Synopsis

```
INCLUDE 'fnx.h'

SUBROUTINE HRECV (typesel, buf, count, handler)

INTEGER typesel
INTEGER buf(*)
INTEGER count
EXTERNAL handler

SUBROUTINE HRECVX(typesel, buf, count, nodesel, ptypesel, handler, hparam)

INTEGER typesel
INTEGER buf(*)
INTEGER count
INTEGER nodesel
INTEGER ptypesel
EXTERNAL handler
INTEGER 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 Fortran Calls Reference Manual* for more information about message type selectors.

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

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

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

nodesel Node number of the message source (the sending node). 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.

handler Handler to execute when the receive completes after an **hrecv()** call. 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()** subroutine.

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

hparam Integer that is passed directly to *handler*. Typically, *hparam* is used by the handler to identify the request that invoked the handler, thus 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.

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()** subroutines 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.

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

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

A handler for the **hrecvx()** subroutine requires a fifth argument, *hparam*. The *hparam* parameter is an integer that is passed by the call to the handler and identifies the request that invoked the handler.

NOTE

The handler function must be written in C. The handler function's parameters all must be of type **long**.

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

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

An example handler for the **hrecvx()** subroutine 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).

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

NOTE

There are a limited number of message IDs available for applications. Therefore, applications need to release unused message IDs. The **hrecv()** and **hrecvx()** subroutines use message IDs internally, but do not return message IDs, like the **irecv()**, and **irecvx()** functions do. The handlers associated with **hrecv()** and **hrecvx()** subroutines release these message IDs.

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.

Limitations and Workarounds

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

HRECV() *(cont.)*

HRECV() *(cont.)*

See Also

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

HSEND() HSEND()

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

Synopsis

```
INCLUDE 'fnx.h'

SUBROUTINE HSEND(type, buf, count, node, ptype, handler)

INTEGER type
INTEGER buf(*)
INTEGER count
INTEGER node
INTEGER ptype
EXTERNAL handler

SUBROUTINE HSENDX(type, buf, count, node, ptype, handler, hparam)

INTEGER type
INTEGER buf(*)
INTEGER count
INTEGER node
INTEGER ptype
EXTERNAL handler
INTEGER hparam
```

Parameters

type	Type of the message to send. Refer to Appendix A of the *Paragon™ System Fortran Calls Reference Manual* for information on message types.
buf	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.

HSEND() *(cont.)*

HSEND() *(cont.)*

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).

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

handler Handler to execute when the send completes, after an **hsend()** call. You must provide the handler and the handler must have four parameters only. See the "Description" section for a description of the handler for the **hsend()** subroutine.

xhandler Handler to execute when the send completes, after an **hsendx()** call. You must provide the handler and the handler must have five parameters only. See the "Description" section for a description of the handler for the **hsendx()** subroutine.

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()** subroutines are asynchronous message-passing calls. After calling one of these subroutines, the call 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. (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.

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

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

A handler for the **hsend()** and **hsendx()** subroutines 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()** subroutine requires a fifth parameter, *hparam*. The *hparam* parameter is an integer passed by the call to the handler that identifies the request invoking the handler.

NOTE

The handler function must be written in C. The handler function's parameters all must be of type **long**.

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

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

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

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

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

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

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.

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()**, **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 'fnx.h'

SUBROUTINE HSENDRECV(type, sbuf, scount, node, ptype, typesel, rbuf,
        rcount, handler)

INTEGER type
INTEGER sbuf(*)
INTEGER scount
INTEGER node
INTEGER ptype
INTEGER typesel
INTEGER rbuf(*)
INTEGER rcount
EXTERNAL handler
```

Parameters

type	Type of the message to send. Refer to Appendix A of the *Paragon™ System Fortran Calls Reference Manual* for information on message types.
sbuf	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 sends and receives a message of any type. Refer to Appendix A of the *Paragon™ System Fortran Calls Reference Manual* for information on message type selectors.

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

rbuf Buffer for storing the reply. The buffer can be of any valid data type, but should match the data type of the buffer in the corresponding send operation.

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

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

Description

The **hsendrecv()** subroutine is an asynchronous system call. The subroutine 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.

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

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.

NOTE

The handler function must be written in C. The handler function's parameters all must be of type **long**.

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()** subprogram to protect that section of code.

NOTE

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

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()**, **hrecv()**, **hsend()**, **iprobe()**, **irecv()**, **isend()**, **isendrecv()**, **masktrap()**

INFOCOUNT() INFOCOUNT()

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

Synopsis

INCLUDE 'fnx.h'

INTEGER FUNCTION **INFOCOUNT**()

INTEGER FUNCTION **INFONODE**()

INTEGER FUNCTION **INFOPTYPE**()

INTEGER FUNCTION **INFOTYPE**()

Description

Use the information calls to return information about a pending or received message. Information calls are used immediately after completion of one of the following calls and the conditions indicated:

- A **cprobe()**, **crecv()**, or **msgwait()** call.
- A **cprobex()** or **crecvx()** call whose *info* parameter was set to the global array *msginfo*.
- An **iprobe()** or **msgdone()** call that returns 1.

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

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

Return Values

The requested information about a pending or received message:

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.

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

Examples

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

```
      include 'fnx.h'

      integer      iam, msg_type
      integer      count, node, pid, type
      character*80 msg, smsg, rmsg
      parameter    (msg_type = 10)

c  Identify self.

      iam = mynode()

c  If node 0, then ...

      if(iam .eq. 0) then
         print *, 'Starting ...'

c    Build message.

         msg = 'Hello from node '
         write(smsg, 100) msg, iam
100      format(a16, i2, '.')
```

INFOCOUNT() *(cont.)*

INFOCOUNT() *(cont.)*

```
c     Send message.

          call csend(msg_type, smsg, len(smsg), -1, mypid())

          write(*, 200) iam, smsg
200       format('Node ', i2, ' sent: ', a20)

c   if not node 0, then ...

       else

c     Probe for message.

          call cprobe(msg_type)

c     Receive message.

          if(infocount() .le. 80) then
             call crecv(msg_type, rmsg, len(rmsg))
             count = infocount()
             type  = infotype()
             pid   = infopid()
             node  = infonode()

c     Report receipt of message.

             write(*, 300) iam, count, type, pid, node
300          format('Node ', i2,
     1          ' reports ', i3
     2          '-byte message of type ', i2,
     3          ' received from PID ', i2,
     4          ' on node ', i2, '.')

             write(*, 400) iam, rmsg
400          format('Node ', i2, ' received: ', a30)

          endif

       endif

       end
```

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

Limitations and Workarounds

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

See Also

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

IODONE() IODONE()

Determine whether an asynchronous read or write operation is complete.

Synopsis

INCLUDE 'fnx.h'

INTEGER FUNCTION **IODONE**(*id*)

INTEGER *id*

Parameters

id	Non-negative I/O ID returned by an asynchronous read or write 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.

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

NOTE

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

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

Return Values

0 Read or write is not yet complete.

1 Read or write is complete.

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

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

Errors

NOTE

The majority of the Fortran I/O errors that you are likely to receive are described in the "Runtime Error Messages" appendix of the *Paragon™ System Fortran Compiler User's Guide*. This section describes additional errors that you may receive.

`Invalid message id`

Use the I/O ID returned by the asynchronous read or write call (for example, the **iread()** or **iwrite()** call).

Examples

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

```
      include 'fnx.h'

      integer   iam, msgid, size
      parameter (size=10000)
      integer   rbuf(size)

c  Identify self.

      iam = mynode()

c  Open existing file.

      call gopen(12, '/tmp/mydata', M_UNIX)

      end file(12)

      msgid = iread(12, rbuf, 4*size)

c     Loop until the read completes.

      do while (iodone(msgid) .eq. 0)
         write(*, 100) iam
100      format('Node ', i3, ' looping ....')
      end do

c     Display a message when the read is finished.

      write(*, 200) iam
200   format('Node ',i3, ' finished reading.')

      close(12)

      end
```

Limitations and Workarounds

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

IODONE() *(cont.)*

IODONE() *(cont.)*

See Also

iowait(), iread(), iwrite()

IOMODE() IOMODE()

Return the I/O mode of a file.

Synopsis

INCLUDE 'fnx.h'

INTEGER FUNCTION **IOMODE**(*unit*)

INTEGER *unit*

Parameters

unit	Unit number (an integer between 1 and 100) assigned when the file was opened.

Description

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

Return Values

Current I/O mode of the file identified by the *unit* 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.

IOMODE() *(cont.)*

IOMODE() *(cont.)*

Errors

NOTE

The majority of the Fortran I/O errors that you are likely to receive are described in the "Runtime Error Messages" appendix of the *Paragon™ System Fortran Compiler User's Guide*. This section describes additional errors that you may receive.

`Bad file number`

Invalid file unit number.

`No such unit`

The *unit* must be a value no larger than 100.

`Fortran runtime error: Unit not open`

A file must be open to get its I/O mode.

Examples

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

```
      include 'fnx.h'

      integer   mode

c  Open existing file.

      call gopen(12, '/tmp/mydata', M_UNIX)
```

```
c  Verify I/O mode.

      mode = iomode(12)
      write(*, 100) mode
100   format('I/O mode set to: ', i2, '.')

      close(12)

      end
```

Limitations and Workarounds

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

See Also

gopen(), **setiomode()**

Paragon™ System Fortran Compiler User's Guide: **open()**

IOWAIT() IOWAIT()

Wait for an asynchronous read or write operation to complete.

Synopsis

INCLUDE 'fnx.h'

SUBROUTINE **IOWAIT**(*id*)

INTEGER *id*

Parameters

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

Description

The **iowait**() subroutine waits until the asynchronous read or write operation (for example, **iread**() or **iwrite**()) identified by *id* completes. When the **iowait**() subroutine returns, the following is true:

- The buffer specified with an **iread**() call that contains valid data (if the *id* parameter identifies the **iread**() call).
- The buffer specified with an **iwrite**() call is available for reuse (if the *id* parameter identifies an **iwrite**() call).
- 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 subroutine.

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.)*

Errors

NOTE

The majority of the Fortran I/O errors that you are likely to receive are described in the "Runtime Error Messages" appendix of the *Paragon™ System Fortran Compiler User's Guide*. This section describes additional errors that you may receive.

`Invalid message id`

Use the I/O ID returned by the asynchronous read or write call (for example, **iread()** or **iwrite()**).

Examples

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

```
      include 'fnx.h'

      integer   iam, mode, msgid, size
      parameter (size=10000)
      integer   sbuf(size), rbuf(size)

c  Open existing file.

      call gopen(12, '/tmp/mydata', M_UNIX)

c  Identify self.

      iam = mynode()

c  Create the list.

      do 1 i = 1, size
         sbuf(i) = i
1     continue
```

IOWAIT() *(cont.)* IOWAIT() *(cont.)*

```
c  Write to the file.

      end file(12)
      msgid = iwrite(12, sbuf, 4*size)

c  Wait until the write completes.

      call iowait(msgid)

      write(*, 200) iam
200   format('Node ',i3, ' finished writing.')

      close(12)

      end
```

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 'fnx.h'

INTEGER FUNCTION **IPROBE**(*typesel*)

INTEGER *typesel*

INTEGER FUNCTION **IPROBEX**(*typesel, nodesel, ptypesel, info*)

INTEGER *typesel*
INTEGER *nodesel*
INTEGER *ptypesel*
INTEGER *info*(8)

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 Fortran 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 accepts a message from any process type.

info Eight-element array of 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.

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

Return Values

0	If the specified message is not available.
1	If the specified message is available.

Description

Use the appropriate asynchronous probe call 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...()** calls to get more information about the message. Otherwise, the **info...()** 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 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 calls (for example, **cprobe()**).

Examples

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

```
      include 'fnx.h'

      integer      done1, done2, iam, id1, id2
      integer      msgid1, msgid2, mtime
      character*80 buf1, buf2

c  Identify self.

      iam = mynode()
```

IPROBE() *(cont.)*

IPROBE() *(cont.)*

```
c  If node 0, then ...

      if(iam  .eq.  0) then
         print *,'Starting ...'

c    Waste some time.

         mtime = mclock()
1        continue
         if(mclock() .lt. mtime+50) goto 1

c    Create and send message type 1.

         buf1   = 'This is message type 1'
         msgid1 = isend(1, buf1, len(buf1), -1, mypid())

c    Wait until message is sent and then report same.

         call msgwait(msgid1)
         write(*, 100) iam
100      format('Node ', i3, ' sent message type 1.')

c    Waste some more time.

         mtime = mclock()
2        continue
         if(mclock() .lt. mtime+100) goto 2

c    Create and send message type 2.

         buf2 = 'This is message type 2'
         msgid2 = isend(2, buf2, len(buf2), -1, mypid())

c    Wait until message is sent and then report same.

         call msgwait(msgid2)
         write(*, 200) iam
200      format('Node ', i3, ' sent message type 2.')

c  If not node 0, then ...

      else
```

```
c     Set up loop exit condition.

          done1 = 0
          done2 = 0
          do while ((done1 .eq. 0) .or. (done2 .eq. 0))

c     Receive message type 1 if available.

          if(iprobe(1) .eq. 1) then
             call crecv(1, buf1, len(buf1))
             write(*, 300) iam
300          format('Node ', i3, ' received message type 1.')
             done1 = -1

c     Receive message type 2 if available.

          else if(iprobe(2) .eq. 1) then
             call crecv(2, buf2, len(buf2))
             write(*, 400) iam
400          format('Node ', i3, ' received message type 2.')
             done2 = -1

c     Flick if neither message is available.

          else
             write(*, 500) iam
500          format('Node ', i3, ' calling flick() ...')
             call flick()
          endif

          end do

        endif

        end
```

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(), infocount(), infonode(), infoptype(), infotype()

IREAD() IREAD()

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

Synopsis

INCLUDE 'fnx.h'

INTEGER FUNCTION **IREAD**(*unit*, *buffer*, *nbytes*)

INTEGER *unit*
INTEGER *buffer*(*)
INTEGER *nbytes*

INTEGER FUNCTION **IREADV**(*unit*, *iov*, *iovcnt*)

INTEGER *unit*
INTEGER *iov*(*)
INTEGER *iovcnt*

Parameters

unit	Unit number (an integer between 1 and 100) assigned when the file was opened.
buffer	Buffer in which the data is stored after it is read. The buffer can be of any valid data type.
nbytes	Size (in bytes) of *buffer*.
iov	Array of iovec entries that identifies the buffers into which the data is to be placed. An iovec entry is a pair of integers. The first integer contains the address of the buffer. The second integer contains the number of bytes in the buffer.
iovcnt	Number of iovec entries in the *iov* array.

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

Description

The **iread**() and **ireadv**() subroutines perform high-speed, asynchronous data reads from a file. The **ireadv**() subroutine performs the same actions as the **iread**() subroutine, but scatters the input data into the buffer specified by the *iov* parameter. An **iread**() or **ireadv**() function call returns to the calling process immediately; the calling process continues to run while the read is being done. If the calling process needs the data for further processing, it must do one of the following:

- Use the **cread**() or **creadv**() subroutine for synchronous reads, instead of the **iread**() or **ireadv**() function.
- Use the **iowait**() subroutine to wait until the read completes.
- Loop until the **iodone**() function 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.

To open a file before using the **iread**() or **ireadv**() function, use the Fortran **open**() statement with the *form* parameter set to `'unformatted'` or use the **gopen**() subroutine.

Mixing the **iread**() or **ireadv**() subroutines with the Fortran **read**() or **write**() statements causes an error.

You can automatically create files using a Fortran **read**() or **write**() statement without an **open**() statement. These kind of files are named with the form *fnode.unit*, where *node* is the node number and *unit* is the value of the *unit* parameter. You can read these kind of files with a **read**() statement. However, these kind of files do not have the correct format for high-speed system reads using the **iread**() or **ireadv**() functions.

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.

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

Return Values

Non-negative I/O ID for use in **iodone()** and **iowait()** calls.

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

NOTE

The majority of the Fortran I/O errors that you are likely to receive are described in the "Runtime Error Messages" appendix of the *Paragon™ System Fortran Compiler User's Guide*. This section describes additional errors that you may receive.

`Attempt to mix standard and PFS I/O call`

You cannot mix the **iread()** function with Fortran **read()** and **write()** statements on the same file.

`Bad file descriptor`

Invalid file unit number.

`Invalid argument`

Check arguments.

`Mixed file operations`

In I/O mode **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.

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

`I/O error`

A disk error occurred.

`No such unit`

The *unit* must be a positive value no larger than 100.

`Too many open files`

Only 64 files can be open at one time for any process.

`Unformatted I/O to FORMATTED file`

Use the Fortran **open()** statement to open the file, setting the proper format.

`Too many I/O requests outstanding`

No available I/O file descriptors. Use the **iowait()** or **iodone()** subprogram for outstanding **iread()** or **iwrite()** requests.

`Tries to read past EOF`

Attempt was made to read past the end-of-file.

Examples

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

```
      include 'fnx.h'

      integer   iam, msgid, size
      parameter (size=10000)
      integer   rbuf(size)

c  Identify self.

      iam = mynode()
```

```
c  Open existing file.

      call gopen(12, '/tmp/mydata', M_UNIX

      msgid = iread(12, rbuf, 4*size)

c  Loop until the read completes.

      do while (iodone(msgid) .eq. 0)
         write(*, 100) iam
100      format('Node ', i3, ' looping ....')
      end do

c  Display a message when the read is finished.

      write(*, 200) iam
200   format('Node ',i3, ' finished reading.')

      close(12)

      end
```

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(), iomode(), iowait(), iseof(), iwrite(), setiomode()

IRECV() IRECV()

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

Synopsis

```
INCLUDE 'fnx.h'

INTEGER FUNCTION IRECV(typesel, buf, count)

INTEGER typesel
INTEGER buf(*)
INTEGER count

INTEGER FUNCTION IRECVX(typesel, buf, count, nodesel, ptypesel, info)

INTEGER typesel
INTEGER buf(*)
INTEGER count
INTEGER nodesel
INTEGER ptypesel
INTEGER info(8)
```

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 Fortran Calls Reference Manual* for more information about message type selectors.
buf	Buffer in which to store the received message. The buffer can be of any valid data type, but should match the data type of the buffer in the corresponding send operation.
count	Length (in bytes) of the *buf* parameter.
nodesel	Node number of the sender. Setting the *nodesel* parameter to -1 receives a message from any node.
ptypesel	Process type of the sender. Setting the *ptypesel* parameter to -1 receives a message from any process type.

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

info Eight-element array of 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.

Description

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

For the **irecv()** function, you can use the **info...()** calls to get more information about the message after it is received. For the **irecv()** function, the same message information is returned in the *info* array. Note, however, that until the receive completes, neither the **info...()** 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 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()** subroutine to wait until the receive completes.
- Loop until the **msgdone()** function returns 1, indicating that the receive is complete.
- Use one of the synchronous calls (for example, the **crecv()** subroutine) instead.

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

Return Values

A message ID (*mid*) for use in **msgcancel()**, **msgdone()**, **msgignore()**, **msgmerge()**, or **msgwait()** system calls.

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

NOTE

The majority of the Fortran I/O errors that you are likely to receive are described in the "Runtime Error Messages" appendix of the *Paragon™ System Fortran Compiler User's Guide*. This section describes additional errors that you may receive.

`Too many requests`

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.

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

Examples

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

```
      include 'fnx.h'

      integer      iam, gotmsg1, gotmsg2, id1, id2, mtime
      character*80 buf1, buf2

c  Identify self.

      iam = mynode()

c  If node 0, then ...

      if(iam  .eq.  0) then
         print *,'Starting ...'

c    Waste some time.

         mtime = mclock()
1        continue
         if(mclock() .lt. mtime+50) goto 1

c    Create and send message type 1.

         buf1   = 'This is message type 1'
         id1 = isend(1, buf1, len(buf1), -1, mypid())

c    Wait until message is sent and then report same.

         call msgwait(id1)
         write(*, 100) iam
100      format('Node ', i3, ' sent message type 1.')

c    Waste some more time.

         mtime = mclock()
2        continue
         if(mclock() .lt. mtime+100) goto 2
```

IRECV() *(cont.)*

IRECV() *(cont.)*

```
c     Create and send message type 2.

          buf2 = 'This is message type 2'
          id2 = isend(2, buf2, len(buf2), -1, mypid())

c     Wait until message is sent and then report same.

          call msgwait(id2)
          write(*, 200) iam
200       format('Node ', i3, ' sent message type 2.')

c  If not node 0, then ...

       else

c     Post receives for the messages.

          id1 = irecv(1, buf1, len(buf1))
          id2 = irecv(2, buf2, len(buf2))

c     Set up loop exit condition.

          gotmsg1 = 0
          gotmsg2 = 0
10        if(gotmsg1 .eq. 1 .and. gotmsg2 .eq. 1) goto 20

c     Receive message type 1 if available.

          if(msgdone(id1) .eq. 1) then
             write(*, 300) iam
300          format('Node ', i3, ' received message type 1.')
             id1 = irecv(1, buf1, len(buf1))
             gotmsg1 = 1
c     Receive message type 2 if available.

          else if(msgdone(id2) .eq. 1) then
             write(*, 400) iam
400          format('Node ', i3, ' received message type 2.')
             id2 = irecv(2, buf2, len(buf2))
             gotmsg2 = 1
```

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

```
c     Flick if neither message is available.

          else
             write(*, 500) iam
500          format('Node ', i3, ' calling flick() ...')
             call flick()
          endif

          goto 10

       endif

20     end
```

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(), hrecv(), hsend(), hsendrecv(), infocount(), infonode(), infoptype(), infotype(), iprobe(), isend(), isendrecv(), msgcancel(), msgdone(), msgignore(), msgmerge(), msgwait()

ISEND() ISEND()

Sends a message and returns immediately. (Asynchronous send)

Synopsis

```
INCLUDE 'fnx.h'

INTEGER FUNCTION ISEND(type, buf, count, node, ptype)

INTEGER type
INTEGER buf(*)
INTEGER count
INTEGER node
INTEGER ptype
```

Parameters

type	Type of the message to send. Refer to Appendix A of the *Paragon™ System Fortran Calls Reference Manual* for information on message types.
buf	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 asynchronous send calls return a message ID. You can use with the **msgdone()** and **msgwait()** system calls 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.

These are asynchronous calls. 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 one of the synchronous send calls (for example, **csend()**) instead.

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

Return Values

A message ID (*mid*) for use in **msgcancel()**, **msgdone()**, **msgignore()**, **msgmerge()**, or **msgwait()** system calls.

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

NOTE

The majority of the Fortran I/O errors that you are likely to receive are described in the "Runtime Error Messages" appendix of the *Paragon™ System Fortran Compiler User's Guide*. This section describes additional errors that you may receive.

`Too many requests`

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.

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

Examples

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

```
      include 'fnx.h'

      integer      iam, gotmsg1, gotmsg2, id1, id2, mtime
      character*80 buf1, buf2

c  Identify self.

      iam = mynode()

c  If node 0, then ...

      if(iam  .eq.  0) then
         print *,'Starting ...'

c    Waste some time.

         mtime = mclock()
1        continue
         if(mclock() .lt. mtime+50) goto 1

c    Announce asynchronous message

         write(*,85) iam
85       format('Node ', i3, ' sending message type 1.')

c    Create and send message type 1.

         buf1   = 'This is message type 1'
         id1 = isend(1, buf1, len(buf1), -1, mypid())

c    Perform concurrent processing

         write(*,90) iam
90       format('Node ', i3, ' waiting for send to complete...')

c    Wait until message is sent and then report same.

         call msgwait(id1)
         write(*, 100) iam
100      format('Node ', i3, ' send of type 1 complete.')
```

ISEND() *(cont.)*

ISEND() *(cont.)*

```
c     Waste some more time.

          mtime = mclock()
2         continue
          if(mclock() .lt. mtime+100) goto 2

c     Announce asynchronous message

          write(*,105) iam
105       format('Node ', i3, ' sending message type 2.')

c     Create and send message type 2.

          buf2 = 'This is message type 2'
          id2 = isend(2, buf2, len(buf2), -1, mypid())

c     Perform concurrent processing

                                                write(*,190) iam
190       format('Node ', i3, ' waiting for send to complete...')

c     Wait until message is sent and then report same.

          call msgwait(id2)
          write(*, 200) iam
200       format('Node ', i3, ' send of type 2 complete.')

c  If not node 0, then ...

       else

c     Post receives for the messages.

          id1 = irecv(1, buf1, len(buf1))
          id2 = irecv(2, buf2, len(buf2))

c     Set up loop exit condition.

          gotmsg1 = 0
          gotmsg2 = 0
10        if(gotmsg1 .eq. 1 .and. gotmsg2 .eq. 1) goto 20
```

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

```
c     Receive message type 1 if available.

          if(msgdone(id1) .eq. 1) then
             write(*, 300) iam
300          format('Node ', i3, ' received message type 1.')
             id1 = irecv(1, buf1, len(buf1))
             gotmsg1 = 1

c     Receive message type 2 if available.

          else if(msgdone(id2) .eq. 1) then
             write(*, 400) iam
400          format('Node ', i3, ' received message type 2.')
             id2 = irecv(2, buf2, len(buf2))
             gotmsg2 = 1

c     Flick if neither message is available.

          else
             write(*, 500) iam
500          format('Node ', i3, ' calling flick() ...')
             call flick()
          endif

          goto 10

       endif

20     end
```

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(), 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 'fnx.h'

INTEGER FUNCTION ISENDRECV(type, sbuf, scount, node, ptype, typesel,
      rbuf, rcount)

INTEGER type
INTEGER sbuf(*)
INTEGER scount
INTEGER node
INTEGER ptype
INTEGER typesel
INTEGER rbuf(*)
INTEGER rcount
```

Parameters

type	Type of the message to send. Refer to Appendix A of the *Paragon™ System Fortran Calls Reference Manual* for more information about message types.
sbuf	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. Refer to Appendix A of the *Paragon™ System Fortran Calls Reference Manual* for information on message type selectors.
rbuf	Buffer for storing the reply. The buffer can be of any valid data type, but should match the data type of the buffer in the corresponding send operation.
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 the **msgdone()** and **msgwait()** system calls to determine when the send-receive completes (that is, the reply arrives). When the reply arrives, the calling process receives the message and stores it in the *rbuf* buffer.

If the reply is too long for the *rbuf* buffer, 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()** subroutine 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

A message ID (*mid*) for use in **msgcancel()**, **msgdone()**, **msgignore()**, **msgmerge()**, or **msgwait()** system calls

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()**.

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

Errors

NOTE

The majority of the Fortran I/O errors that you are likely to receive are described in the "Runtime Error Messages" appendix of the *Paragon™ System Fortran Compiler User's Guide*. This section describes additional errors that you may receive.

`Too many requests`

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.

Examples

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

```
      include 'fnx.h'

      integer          msgid, inode, ipid
      real             lbuf, value
      double precision sbuf0, rbuf0, sbuf1, rbuf1

      sbuf0 = 0.0
      rbuf0 = 0.0
      sbuf1 = 0.0
      rbuf1 = 0.0
      lbuf  = 0.0
      value = 3.14

c  Identify self.

      iam = mynode()

c  If node 0, then ...

      if(iam .eq. 0) then
         print *,'Starting ...'
```

ISENDRECV() *(cont.)*

ISENDRECV() *(cont.)*

```
c     Send message to other node(s) and post receive for result.

          sbuf0 = 0.21381
          msgid = isendrecv(10,sbuf0,8,-1,mypid(),11,rbuf0,8)

c     Do some processing while waiting for results.

          lbuf = sqrt(value)
          write (*, 50) value, lbuf
50        format('The square root of ',F5.2, ' is ', F5.2)

c     Wait for result.

          call msgwait(msgid)

c     Display result.

          write(*, 100) sbuf0, rbuf0
100       format('Arcsin of ', d15.5, ' is ', d15.5)

c  If not node 0, then ...

       else

c     Receive message.

          call crecv(10, rbuf1, 8)

c     Get sending node and pid (for returning result).

          inode = infonode()
          ipid  = infopid()

c     Get value of result.

          sbuf1 = dasind(rbuf1)

c     Send result to calling node.

          call csend(11, sbuf1, 8, inode, ipid)

       endif

       end
```

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(), hrecv(), hsend(), hsendrecv(), iprobe(), irecv(), isend(), isendrecv(), msgcancel(), msgdone(), msgignore(), msgmerge(), msgwait()

ISEOF() ISEOF()

Determine whether specified file pointer is at end-of-file.

Synopsis

```
INCLUDE 'fnx.h'

INTEGER FUNCTION ISEOF(unit)

INTEGER unit
```

Parameters

unit	The unit number (an integer between 1 and 100) assigned when the file was opened.

Description

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

Return Values

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

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

Errors

NOTE

The majority of the Fortran I/O errors that you are likely to receive are described in the "Runtime Error Messages" appendix of the *Paragon™ System Fortran Compiler User's Guide*. This section describes additional errors that you may receive.

`Attempt to mix standard and PFS I/O calls`

Fortran **read()** and **write()** and operating system I/O calls cannot be used in the same file.

`Bad file number`

Use the *unit* assigned with **open()**.

`Fortran runtime error: Unit not open`

A file must be open to check the position of its file pointer.

`No such unit`

The *unit* number must be a value no larger than 100.

Examples

The following example shows how to use the **iseof()** function to determine end-of-file:

```
      include 'fnx.h'

      character*20 sbuf
      character*1  char

c  Open existing file.

      call gopen(12, '/tmp/mydata', M_UNIX)
```

ISEOF() *(cont.)*

ISEOF() *(cont.)*

```
c  Identify self.

      iam = mynode()

c  Read and display the file.

      do while(iseof(12) .eq. 0)

         call cread(12, char, 1)

         write(*, 100) iam, char
100      format('Node ', i3, ' read:  ', a1)

      end do

      close(12)

      end
```

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()

IWRITE() IWRITE()

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

Synopsis

```
INCLUDE 'fnx.h'

INTEGER FUNCTION IWRITE(unit, buffer, nbytes)

INTEGER unit
INTEGER buffer(*)
INTEGER nbytes

INTEGER FUNCTION IWRITEV(unit, iov, iovcnt)

INTEGER unit
INTEGER iov(*)
INTEGER iovcnt
```

Parameters

unit	Unit number of the file (an integer between 1 and 100) assigned when the file was opened.
buffer	Buffer containing data to be written. The buffer can be of any valid data type.
nbytes	Number of bytes to write. The size is limited only by the memory available for the buffer.
iov	Array of iovec entries, which identifies the buffers containing the data to be written. An iovec entry is defined to be a pair of integers, the first integer contains the address of the buffer, the second contains the number of bytes in the buffer.
iovcnt	Number of iovec entries in the **iov** array.

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

Description

The **iwrite()** and **iwritev()** subroutines perform high-speed, asynchronous data writes to a file. The **iwritev()** subroutine performs the same actions as the **iwrite()** subroutine, but gathers the output data from the buffers specified by the *iov* parameter. A call to the **iwrite()** or **iwritev()** function returns immediately to the calling process. The calling process continues to run while the write is being done. If the calling process needs the write buffer for further processing, it must do one of the following:

- Use the **cwrite()** or **cwritev()** subroutine (synchronous write) 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.

To open a file before using asynchronous write calls, use the Fortran **open()** statement with the *form* parameter set to `'unformatted'` or use the **gopen()** subroutine.

For a given file, mixing the operating system read and write calls (for example, **iread()** or **iwrite()**) with the Fortran **read()** and **write()** statements causes an error.

You can automatically create files using a Fortran **read()** or **write()** statement without an **open()** statement. Automatically-created files are named with the form f*node.unit*, where *node* is the node number and *unit* is the value of the *unit* parameter. You can write these files with a **write()** statement, however, these files do not have the correct format for high-speed system writes using the **iwrite()** or **iwritev()** function.

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

Return Values

Non-negative I/O ID for use in **iodone()** and **iowait()** calls.

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

NOTE

The majority of the Fortran I/O errors that you are likely to receive are described in the "Runtime Error Messages" appendix of the *Paragon™ System Fortran Compiler User's Guide*. This section describes additional errors that you may receive.

`Attempt to mix standard and PFS I/O call`

You cannot mix the **iwrite()** function with Fortran **read()** and **write()** statements on the same file.

`Attempt to write to READONLY file`

Check file attributes.

`Bad file descriptor`

Invalid file unit number.

`Invalid argument`

Check arguments.

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

`I/O error`

A disk error occurred. If error persists, run the diagnostics.

`Mixed file operations`

In I/O mode **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.

`No space left on device`

Not enough space on device to which you are writing. Create more space in file system.

`Too many I/O requests outstanding`

Use **iowait()** or **iodone()** for outstanding **iwrite()** requests.

`Too many open files`

Only 64 files can be open at one time for any process.

`Unformatted I/O to FORMATTED file`

Use Fortran **open()** to open the file, setting proper format.

Examples

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

```
      include 'fnx.h'

      integer   iam, mode, msgid, size
      parameter (size=10000)
      integer   sbuf(size), rbuf(size)

c  Open existing file.

      call gopen(12, '/tmp/mydata', M_UNIX)
```

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

```
c  Identify self.

      iam = mynode()

c  Create the list.

      do 1 i = 1, size
         sbuf(i) = i
1     continue

c  Write to the file.

      end file(12)
      msgid = iwrite(12, sbuf, 4*size)

c  Wait until the write completes.

      call iowait(msgid)

      write(*, 200) iam
200   format('Node ',i3, ' finished writing.')

      close(12)

      end
```

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(), iomode(), iowait(), iread(), iseof(), setiomode()

LSEEK() LSEEK()

Move the read/write file pointer in a normal (non-extended) file.

Synopsis

```
INCLUDE 'fnx.h'

INTEGER FUNCTION LSEEK(unit, offset, whence)

INTEGER unit
INTEGER offset
INTEGER whence
```

Parameters

unit Identifies the file whose pointer is to be moved. The unit number is an integer between 1 and 100 that was assigned to the file when it was opened.

offset Affects the file pointer in the manner specified by the *whence* parameter.

whence Specifies how *offset* affects the file pointer:

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 *offset* bytes beyond the end of the file.

Description

The **lseek()** subroutine moves the file pointer in an open normal file specified by the *unit* parameter. To move the pointer in an extended file, use the **eseek()** system call. The *whence* parameter determines how the file pointer is interpreted.

The **lseek()** function can be used on extended files if the offset result is less than 2G bytes.

LSEEK() *(cont.)* LSEEK() *(cont.)*

The **lseek()** subroutine allows a file pointer to be set beyond the end of existing data in the file. If data is later written at this point, reading data in the gap returns bytes with the value 0 (zero) until data is actually written into the gap.

The **lseek()** subroutine does not extend the size of the file by itself.

Return Values

The new position of the file pointer (measured in bytes from the beginning of the file)

Errors

NOTE

The majority of the Fortran I/O errors that you are likely to receive are described in the "Runtime Error Messages" appendix of the *Paragon™ System Fortran Compiler User's Guide*. This section describes additional errors that you may receive.

`Bad file number`

Use the *unit* number assigned when the file was opened.

`No such unit`

The *unit* number must be a value no larger than 100.

`Mixed file operations`

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

`Seek to different file pointers`

Two or more application processes are calling **lseek()** with different shared I/O modes (**M_SYNC** or **M_RECORD**).

LSEEK() *(cont.)* LSEEK() *(cont.)*

```
Fortran runtime error: Unit not open
```

A file must be open to perform a seek operation.

Examples

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

```
      include 'fnx.h'

      integer newpos, newsize

c  Open the file /tmp/mydata.

      call gopen(12, '/tmp/mydata', M_UNIX)

c  Set file size to 1000 bytes.

      newsize = lsize(12, 1000, SIZE_SET)

      write(*, 100) newsize
100   format('New file size is:        ', i10)

c  Move read/write pointer to the end of the file.

      newpos = lseek(12, 0, SEEK_END)

      write(*, 200) newpos
200   format('New pointer position is: ', i10)

c  Close the file /tmp/mydata.

      close(12)

      end
```

Limitations and Workarounds

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

LSEEK() *(cont.)*

LSEEK() *(cont.)*

See Also

cread(), cwrite(), eseek(), esize(), iread(), iseof(), iwrite(), lsize()

LSIZE() LSIZE()

Change the size of a normal (non-extended) file.

Synopsis

```
INCLUDE 'fnx.h'

INTEGER FUNCTION LSIZE(unit, offset, whence)

INTEGER unit
INTEGER offset
INTEGER whence
```

Parameters

unit Unit number of a regular file opened for writing. The unit number is an integer between 1 and 100 that was assigned to the file when it was opened.

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

whence Specifies how *offset* affects the file size:

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*.

NOTE

If the new size specified by *offset* and *whence* is greater than the available disk space, **lsize()** allocates all of the available space, prints an error message, and terminates your program.

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

Description

The **lsize()** function increases the size of a file according to the *offset* and *whence* parameters.

Use the **lsize()** function to preallocates 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.

The **lsize()** function increases the size of a file. If the file is 2G bytes or more in size, use the **esize()** system call. This function cannot decrease the size of a file.

The **lsize()** function can be used on extended files if the offset result is less than 2G bytes.

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.

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.

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

NOTE

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

Return Values

>= 0	The call is successful (returns the new file size in bytes).
-1	The call is not successful.

NOTE

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

LSIZE() *(cont.)*

LSIZE() *(cont.)*

Errors

NOTE

The majority of the Fortran I/O errors that you are likely to receive are described in the "Runtime Error Messages" appendix of the *Paragon™ System Fortran Compiler User's Guide*. This section describes additional errors that you may receive.

`Bad file number`

Use the *unit* number assigned when the file was opened.

`Fortran runtime error: unit not open`

A file must be open to perform a size operation.

`Invalid argument`

The value for the *whence* parameter is invalid or the resulting file size would be invalid.

`Invalid size`

Cannot be used in extended file sizes. Use **esize()**.

`No space left on device`

The new size specified by *offset* and *whence* is greater than the available disk space. Create more space in file system.

`Operation not supported on this file system`

The *unit* 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.

`Read-only file system`

The unit number refers to a file that resides in a read-only file system.

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

Examples

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

```
      include 'fnx.h'

      integer newpos, newsize

c  Open the file /tmp/mydata.

      call gopen(12, '/tmp/mydata', M_UNIX)

c  Set file size to 1000 bytes.

      newsize = lsize(12, 1000, SIZE_SET)

      write(*, 100) newsize
100   format('New file size is:        ', i10)

c  Move read/write pointer to the end of the file.

      newpos = lseek(12, 0, SEEK_END)

      write(*, 200) newpos
200   format('New pointer position is: ', i10)

c  Close the file /tmp/mydata.

      close(12)

      end
```

Limitations and Workarounds

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

See Also

eseek(), esize(), lsize

MASKTRAP() MASKTRAP()

Enables or disables send and receive handlers.

Synopsis

INCLUDE 'fnx.h'

INTEGER FUNCTION **MASKTRAP**(*state*)

INTEGER *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.

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

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.

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

The previous value of *state*.

Limitations and Workarounds

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

See Also

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

MSGCANCEL() MSGCANCEL()

Cancels an asynchronous send or receive operation.

Synopsis

INCLUDE 'fnx.h'

SUBROUTINE **MSGCANCEL**(*mid*)

INTEGER *mid*

Description

The **msgcancel()** subroutine 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()**.

Parameters

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

Limitations and Workarounds

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

See Also

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

MSGDONE() MSGDONE()

Determines whether an asynchronous send or receive operation is complete.

Synopsis

INCLUDE 'fnx.h'

INTEGER FUNCTION **MSGDONE**(*mid*)

INTEGER *mid*

Parameters

mid	Message ID returned by one of the asynchronous send or receive calls (for example, **isend()**, **irecv()**, or **isendrecv()**) or by the **msgmerge()** 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 calls) contains valid information.
- The **info...()** 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

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

Limitations and Workarounds

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

See Also

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 is complete.

Synopsis

INCLUDE 'fnx.h'

SUBROUTINE **MSGIGNORE**(*mid*)

INTEGER *mid*

Parameters

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

Description

The **msgignore()** subroutine releases a message ID as soon as its asynchronous send or receive operation completes.

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:

- Your application must have some alternate means to determine when it can reuse a send or receive buffer.
- For applications that have a natural synchronization, **msgignore()** is more convenient and has better performance than **msgmerge()**.
- Do not use **msgignore()** as a substitute for **msgwait()**.

Limitations and Workarounds

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

See Also

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

MSGMERGE() MSGMERGE()

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

Synopsis

INCLUDE 'fnx.h'

INTEGER FUNCTION **MSGMERGE**(*mid1*, *mid2*)

INTEGER *mid1*
INTEGER *mid2*

Parameters

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

Description

The **msgmerge**() function groups *mid2* with *mid1* and returns a *mid* 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 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...**() calls after a **msgmerge**() call; the information returned is unpredictable.

Return Values

A message ID (*mid*) for use (instead of *mid1* or *mid2*) in **msgcancel()**, **msgdone()**, **msgignore()**, **msgmerge()**, or **msgwait()** calls.

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()**.

Limitations and Workarounds

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

See Also

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

MSGWAIT() MSGWAIT()

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

Synopsis

INCLUDE 'fnx.h'

SUBROUTINE **MSGWAIT**(*mid*)

INTEGER *mid*

Parameters

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

Description

The **msgwait()** subroutine causes a node process to wait until the asynchronous send or receive operation identified by *mid* completes. When **msgwait()** 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 calls) contains valid information.
- The **info...()** 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()**.

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

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

Limitations and Workarounds

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

See Also

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

MYHOST() MYHOST()

Gets the node number of the controlling process.

Synopsis

INCLUDE 'fnx.h'

INTEGER FUNCTION **MYHOST**()

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

The node number of the controlling process.

Limitations and Workarounds

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

See Also

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

MYNODE() MYNODE()

Gets the node number of the calling process.

Synopsis

INCLUDE 'fnx.h'

INTEGER FUNCTION **MYNODE**()

Description

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

Return Values

The node number of the calling process.

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:

```
      include 'fnx.h'

      integer      iam, msg_type
      integer      count, node, pid, type
      character*80 msg, smsg, rmsg
      parameter    (msg_type = 10)

c  Identify self.

      iam = mynode()

c  If node 0, then ...

      if(iam .eq. 0) then
         print *, 'Starting ...'
```

MYNODE() *(cont.)*

MYNODE() *(cont.)*

```
c     Build message.

          msg = 'Hello from node '
          write(smsg, 100) msg, iam
100       format(a16, i3, '.')

c     Send message.

          call csend(msg_type, smsg, len(smsg), -1, mypid())

          write(*, 200) iam, smsg
200       format('Node ', i3, ' sent: ', a20)

c  if not node 0, then ...

       else

c     Probe for message.

          call cprobe(msg_type)

c     Receive message.

          if(infocount() .le. len(rmsg)) then
             call crecv(msg_type, rmsg, len(rmsg))
             count = infocount()
             type  = infotype()
             pid   = infopid()
             node  = infonode()

c     Report receipt of message.

             write(*, 300) iam, count, type, pid, node
300          format('Node ', i3,
     1          ' reports ', i3
     2          '-byte message of type ', i2,
     3          ' received from PID ', i2,
     4          ' on node ', i3, '.')

             write(*, 400) iam, rmsg
400          format('Node ', i3, ' received: ', a30)

          endif
       endif
       end
```

Limitations and Workarounds

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

See Also

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

MYPTYPE() MYPTYPE()

Gets the process type of the calling process.

Synopsis

INCLUDE 'fnx.h'

INTEGER FUNCTION **MYPTYPE**()

Description

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

Return Values

The process type (*ptype*) of the calling process.

Examples

The following example shows how to use the **myptype**() function to get the process type of the calling process:

```
      include 'fnx.h'

      integer iam, msg_type
      integer count, node, pid, type
      character*80 msg, smsg, rmsg
      parameter (msg_type = 10)

c Identify self.

      iam = mynode()

c If node 0, then ...

      if(iam .eq. 0) then
         print *, 'Starting ...'
```

MYPTYPE() *(cont.)* MYPTYPE() *(cont.)*

```
c Build message.

         msg = 'Hello from node '
         write(smsg, 100) msg, iam
  100    format(a16, i3, '.')

c Send message.

         call csend(msg_type, smsg, len(smsg), -1, myptype())

         write(*, 200) iam, smsg
  200    format('Node ', i3, ' sent: ', a20)

c if not node 0, then ...

      else

c Probe for message.

         call cprobe(msg_type)

c Receive message.

         if(infocount() .le. len(rmsg)) then
            call crecv(msg_type, rmsg, len(rmsg))
            count = infocount()
            type = infotype()
            pid = infopid()
            node = infonode()

c Report receipt of message.

            write(*, 300) iam, count, type, pid, node
300         format('Node ', i3,
     1             ' reports ', i3
     2             '-byte message of type ', i2,
     3             ' received from PID ', i2,
     4             ' on node ', i3, '.')

            write(*, 400) iam, rmsg
  400       format('Node ', i3, ' received: ', a30)

         endif
       endif
       end
```

Limitations and Workarounds

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

See Also

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

NUMNODES() NUMNODES()

Gets the number of nodes in the application.

Synopsis

INCLUDE 'fnx.h'

INTEGER FUNCTION **NUMNODES**()

Description

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

Return Values

The number of nodes in the application.

Examples

The following example shows how **numnodes()** can be used to determine the number of nodes in the application:

```
      include 'fnx.h'

      integer          count, dpsize, iam, xsize, ysize, nbrnodes
      parameter          (xsize = 4)
      parameter          (ysize = 16)
      double precision x(xsize), y(ysize), dot, norm, work
      character*80     msg

c     Initialize.

      count    = 0
      dpsize   = 8
      dot      = 0.0
      nbrnodes = numnodes()

c     Identify self.

      iam = mynode()
```

NUMNODES() *(cont.)*

NUMNODES() *(cont.)*

```
      if(iam .eq. 0) print *, 'Starting ...'

c  Each node creates and displays its four-element vector.

      do 1, i = 1, xsize
         x(i) = iam * (xsize) + i-1
         write(*, 100) iam, i, x(i)
100      format('Node ', i1, ' x(', i1, ') = ', f3.1)
1     continue

c  Each node calculates and displays its dot product.

      do 2, i = 1, xsize
         dot = dot + x(i)*x(i)
2     continue
      write(*, 200) iam, dot
200   format('Node ', i1, ' dot = ', f10.6)

c  Each node sums the dot products of all nodes.

      call gdsum(dot, 1, work)

c  Node 0 displays the resulting dot product.

      if(iam .eq. 0) write(*, 300) dot
300   format('dot = ', f10.6)

c  Each node normalizes its dot products.

      norm = dsqrt(dot)
      do 3, i = 1, xsize
         x(i) = x(i)/norm
3     continue

c  Each node collects contributions from other node.

     call gcol(x, xsize*dpsize, y, nbrnodes*xsize*dpsize, count)
```

```
c  Node 0 displays the resulting vector.

      if(iam .eq. 0) then
         do 4, i = 1, nbrnodes*xsize
            write(*, 400) i, y(i)
400         format('y(', i1, ') = ', f3.1)
4        continue

      endif
      end
```

Limitations and Workarounds

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

See Also

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

NX_APP_NODES() NX_APP_NODES()

Returns the list of nodes allocated to an application.

Synopsis

INCLUDE 'fnx.h'

INTEGER **NX_APP_NODES**(*pgroup, ptr, list_size*)

INTEGER *pgroup*
POINTER *(ptr, node_list(1))*
INTEGER *list_size*

Parameters

pgroup	Process group ID for the application, 0 (zero) to specify the calling application. If the specified process group ID is not the 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.
ptr	Pointer variable that specifies the address of the list of nodes for the application. The node numbers are root-partition node numbers. The call allocates memory for this parameter. Free this memory using the **free()** subroutine.
list_size	Variable into which the **nx_app_nodes()** function stores the number of elements in the *node_list* 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.

NX_APP_NODES() *(cont.)* NX_APP_NODES() *(cont.)*

Examples

The following example prints the list of nodes for an application:

```
      include 'fnx.h'

      pointer    (ptr, mynodes(1))
      integer    nnodes
      integer    i, status

      status = nx_app_nodes(0, ptr, nnodes)

      if(status .ne. 0) then
         call nx_perror("nx_app_nodes()")
         stop
      end if

      do 2, i = 1, nnodes
         print *, mynodes(i)
2     continue

      call free(ptr)

      end
```

Errors

`Application does not exist for process group`

The specified process group does not exist.

`Partition permission denied`

Insufficient access permission for this operation on a partition.

Limitations and Workarounds

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

NX_APP_NODES() *(cont.)*

NX_APP_NODES() *(cont.)*

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 'fnx.h'

INTEGER FUNCTION NX_APP_RECT(rows, cols)

INTEGER rows
INTEGER cols

INTEGER FUNCTION MYPART(rows, cols)

INTEGER rows
INTEGER cols
```

Parameters

rows	Number of rows in the node set for the application. If the node set is not a rectangle, the value pointed to by *rows* is set to 1.
cols	Number of columns in the node set 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.)*

Examples

This example returns the number of rows and columns used by the application.

```
include 'fnx.h'

integer*4  rows, cols, result

if (mynode() .eq. 0) then
   status = nx_app_rect(rows, cols)
   if(status .ne. 0) then
      call nx_perror("nx_app_rect()")
      stop
   end if
   print *, "Number of rows    = ", rows
   print *, "Number of columns = ", cols
end if
end
```

Limitations and Workarounds

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

See Also

Paragon™ System Fortran Calls Reference Manual: **nx_app_nodes()**, **nx_initve_rect()**, **nx_mkpart()**, **nx_part_attr()**, **nx_root_nodes()**

Paragon™ XP/S System Commands Reference Manual: ***application***, **mkpart**

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 'fnx.h'

INTEGER FUNCTION **NX_CHPART_EPL**(*partition*, *priority*)

CHARACTER *(*) *partition*
INTEGER *priority*

INTEGER FUNCTION **NX_CHPART_MOD**(*partition*, *mode*)

CHARACTER*(*) *partition*
INTEGER *mode*

INTEGER FUNCTION **NX_CHPART_NAME**(*partition*, *name*)

CHARACTER*(*) *partition*
CHARACTER *name*(*)

INTEGER FUNCTION **NX_CHPART_OWNER**(*partition*, *owner*, *group*)

CHARACTER*(*) *partition*
INTEGER *owner*
INTEGER *group*

INTEGER FUNCTION **NX_CHPART_RQ**(*partition*, *rollin_quantum*)

CHARACTER*(*) *partition*
INTEGER *rollin_quantum*

NX_CHPART_EPL() *(cont.)* NX_CHPART_EPL() *(cont.)*

INTEGER FUNCTION **NX_CHPART_SCHED**(*partition*, *sched_type*)

CHARACTER*(*) *partition*
INTEGER *sched_type*

Parameters

partition 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 owner name is not changed. See the *Paragon™ System Fortran Compiler User's Guide* for information about using the **getuid()** system call 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.

NX_CHPART_EPL() *(cont.)* NX_CHPART_EPL() *(cont.)*

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 name is not changed. See the *Paragon™ System Fortran Compiler User's Guide* for information about using the **getgid**() system call to convert a group name to a numeric user 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 -1 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 *fnx.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 **nx_chpart...**() functions change specific characteristics of a partition. Each of these calls specifically changes a partition characteristic as follows:

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

nx_chpart_mod()
Changes the partition's permission modes.

NX_CHPART_EPL() *(cont.)*

NX_CHPART_EPL() *(cont.)*

nx_chpart_name()
Changes the partition's name.

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

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 only. You cannot use this function to change the partition's parent partition or the partition's relationship in a partition hierarchy.

Each partition has an owner, a group, and protection modes that determine who can perform what operations on a partition. When you create a partition, you become the partition's owner and the partition's group is set to your current group. The **nx_chpart_owner()** function changes the owner and group of a partition. The owner and group must be specified as a numeric ID, not as a name.

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 limits 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

0	Partition's characteristic was successfully changed.
-1	Error. Use the **nx_perror()** subroutine to display the error message for the current error.

Errors

`Allocator internal error`

An internal error occurred in the node allocation server.

`Exceeded allocator configuration parameters`

The application exceeded the configuration parameters for the partition. See the **allocator** manual page.

`Exceeds partition resources`

Request exceeds the partition's resources.

NX_CHPART_EPL() *(cont.)*

NX_CHPART_EPL() *(cont.)*

`Invalid group`

An invalid group ID was specified.

`Invalid partition rename`

You specified a partition name that was not a simple name. You cannot change a partition's relationship within a partition hierarchy.

`Invalid priority`

An invalid priority level was specified.

`Invalid user`

An invalid user ID was specified.

`Partition not found`

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

`Partition permission denied`

The application has insufficient access permission on a partition.

`Scheduling parameters conflict with allocator configuration parameters`

The scheduling parameters conflict with the allocator configuration. See the **allocator** manual page.

Limitations and Workarounds

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

NX_CHPART_EPL() *(cont.)*

NX_CHPART_EPL() *(cont.)*

See Also

Paragon™ System Fortran 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)**

NX_EMPTY_NODES() NX_EMPTY_NODES()

Returns the list of empty nodes in the root partition.

Synopsis

INCLUDE 'fnx.h'

INTEGER **NX_EMPTY_NODES**(*ptr, list_size*)

POINTER *(ptr, node_list*(1))
INTEGER *list_size*

Parameters

ptr Pointer variable to the *node_list* array into which the **nx_empty_nodes()** function stores the list of empty nodes found in the root partition. The node numbers are root-partition node numbers. The call allocates memory for this parameter. Free this memory using the **free()** function.

list_size 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.

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.

NX_EMPTY_NODES() *(cont.)* NX_EMPTY_NODES() *(cont.)*

Examples

The following example prints the list of the empty nodes in the root partition:

```
      include 'fnx.h'

      integer*4  empty(1)
      pointer    (ptr, empty)
      integer    nempty
      integer    i, status

      status = nx_empty_nodes(ptr, nempty)

      if(status .ne. 0) then
         call nx_perror("nx_empty_nodes()")
         stop
      end if

      do 2, i = 1, nempty
         print *, empty(i)
2     continue

      call free(ptr)

      end
```

Limitations and Workarounds

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

See Also

Paragon™ System Fortran 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 'fnx.h'

INTEGER FUNCTION **NX_FAILED_NODES**(*ptr*, *list_size*)

POINTER (*ptr*, *node_list*(1))
INTEGER *list_size*

Parameters

ptr Pointer variable to the array *node_list* into which the **nx_failed_nodes()** function stores the list of failed nodes found in the root partition. The node numbers are root-partition node numbers. 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 for the **showpart** command.

NOTE

Do not call the **nx_failed_nodes()** function on more than a few nodes at once.

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.

NX_FAILED_NODES() *(cont.)* NX_FAILED_NODES() *(cont.)*

Examples

The following example prints the list of the failed nodes in the root partition:

```
      include 'fnx.h'

      integer*4  failed(1)
      pointer    (ptr, failed)
      integer    nfailed
      integer    i, status

      status = nx_failed_nodes(ptr, nfailed)

      if(status .ne. 0) then
         call nx_perror("nx_failed_nodes()")
         stop
      end if

      do 2, i = 1, nfailed
         print *, failed(i)
2     continue

      call free(ptr)

      end
```

Limitations and Workarounds

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

See Also

Paragon™ System Fortran Calls Reference Manual: **mynode()**, **nx_app_nodes()**, **nx_empty_nodes()**

Paragon™ XP/S System Commands Reference Manual: **showpart**

NX_INITVE() NX_INITVE()

nx_initve(), nx_initve_rect(): Initializes an application.

Synopsis

INCLUDE 'fnx.h'

INTEGER FUNCTION **NX_INITVE**(*partition*, *size*, *account*, *argc*, *argv*)

CHARACTER*(*) *partition*
INTEGER *size*
CHARACTER*(*) *account*
INTEGER *argc*
INTEGER *argv*

INTEGER FUNCTION **NX_INITVE_RECT**(*partition*, *anchor_node*, *rows*, *columns*, *account*, *argc*, *argv*)

CHARACTER*(*) *partition*
INTEGER *anchor_node*
INTEGER *row*
INTEGER *cols*
CHARACTER*(*) *account*
INTEGER *argc*
INTEGER *argv*

Parameters

partition Relative or absolute pathname of the partition in which to run the application, or a null string ("") to use 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).

NX_INITVE() *(cont.)*

NX_INITVE() *(cont.)*

size Size of the application (number of nodes to run the application on), or 0 (zero) to use the default size. The default size is the size specified by the *NX_DEFAULT_SIZE* environment variable, or is all nodes of the partition if the *NX_DEFAULT_SIZE* environment variable is not set.

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 the null string.

argc Number of arguments on the command line (including the application name). If the value is 0 (zero), the command line is ignored. If the value is greater than 0 (zero), the command line is parsed.

argv Command line argument values. Fortran does not support this parameter, so the value must be 0 (zero).

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

- Call either **nx_initve()** or **nx_initve_rect()** before any other Paragon system calls.
- Call either **nx_initve()** or **nx_initve_rect()** only once.
- Use the **-lnx** switch to link a program that calls either **nx_initve()** or **nx_initve_rect()**. 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 node set 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 initialized.
-1	Error. Use the **nx_perror()** subroutine to display the error message for the current error.

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

Errors

`Application exists for a process group`

An application has already been established for the process group.

`Packet size invalid or out of range`

The packet size is invalid or out of range.

`Memory buffer invalid or out of range`

The memory buffer size is invalid or out of range.

`Memory each invalid or out of range`

The memory each size is invalid or out of range.

`Memory export invalid or out of range`

The memory export size is invalid or out of range.

`Send threshold invalid or out of range`

The send threshold size is invalid or out of range.

`Give threshold invalid or out of range`

The give threshold size is invalid or out of range.

`Request overlaps with nodes in use`

A partition or application overlaps with another partition or application.

`Use of -plk not allowed in gang-scheduled partition`

An application cannot use the **-plk** switch in a gang-scheduled partition.

`The application and the OS are of incompatible revisions`

Your application's code is no longer up to date with the current release of the installed operating system. You must relink your application.

`Allocator internal error`

An internal error occurred in the node allocation server.

`Partition permission denied`

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

`Bad node specification`

A bad node was specified.

`Invalid priority`

An invalid priority value was specified.

`Partition not found`

The specified partition was not found.

`Exceeds partition resources`

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

Paragon™ System Fortran Calls Reference Manual: **nx_nfork()**, **nx_load()**

Paragon™ XP/S System Commands Reference Manual: **allocator**, ***application***

NX_INITVE_ATTR() NX_INITVE_ATTR()

Initializes a new application with specified attributes.

Synopsis

INCLUDE 'fnx.h'

INTEGER FUNCTION **NX_INITVE_ATTR**(*partition, argc, argv [attribute, value]* ... **NX_ATTR_END**)

CHARACTER*(*) *partition*
INTEGER *argc*
INTEGER *argv*
INTEGER *attribute*
CHARACTER* | INTEGER *value*

Parameters

partition Relative or absolute pathname of the partition in which to run the application, or a null string ("") to use 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).

argc Number of arguments on the command line (including the application name). If the value is 0 (zero), the command line is ignored. If the value is greater than 0 (zero), the command line is parsed.

argv Command line argument values. Fortran does not support this parameter, so the value must be 0 (zero).

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.

NX_INITVE_ATTR() *(cont.)* NX_INITVE_ATTR() *(cont.)*

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

NOTE

If you call **nx_initve_attr()** in a subprogram, you must include *fnx.h* after the subprogran declaration and before the call. This is required for the call to recognise the pre-defined attribute constants (for example, **NX_ATTR_SZ**).

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_INITVE_ATTR() *(cont.)* NX_INITVE_ATTR() *(cont.)*

- **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."

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.

Errors

`Application exists for process group`

An application has already been established for the process group.

`Memory buffer invalid or out of range`

The memory buffer size is invalid or out of range.

`Memory each invalid or out of range`

The memory each size is invalid or out of range.

`Memory export invalid or out of range`

The memory export size is invalid or out of range.

`Packet size invalid or out of range`

The packet size is invalid or out of range.

`Send threshold invalid or out of range`

The send threshold size is invalid or out of range.

NX_INITVE_ATTR() *(cont.)* NX_INITVE_ATTR() *(cont.)*

`Give threshold invalid or out of range`

The give threshold size is invalid or out of range.

`Request overlaps with nodes in use`

A partition or application overlaps with another partition or application.

`Use of -plk not allowed in gang-scheduled partition`

An application cannot use the **-plk** switch in a gang-scheduled partition.

`The application and the OS are incompatible revisions`

Your application's code is no longer up to date with the current release of the installed operating system. You must relink your application.

`Allocator internal error`

An internal error occurred in the node allocation server.

`Partition permission denied`

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

`Bad node specification`

A bad node was specified.

`Invalid priority`

An invalid priority value was specified.

`Partition not found`

The specified partition was not found.

`Attributes do not match`

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.

`Exceeds partition resources`

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 'fnx.h'

INTEGER FUNCTION NX_LOAD(node_list, numnodes, ptype, pid_list,
      pathname)

INTEGER node_list(*)
INTEGER numnodes
INTEGER ptype
INTEGER pid_list(*)
CHARACTER*(*) pathname

INTEGER FUNCTION NX_LOADVE(node_list, numnodes, ptype, pid_list,
      pathname, argv, envp)

INTEGER node_list(*)
INTEGER numnodes
INTEGER ptype
INTEGER pid_list(*)
CHARACTER*(*) pathname
INTEGER argv
INTEGER envp
```

NOTE

It is possible that after loading and starting an executable on multiple nodes, the executable can fail on more than one node. Each failure on each node can be for a different reason. The value of the error number reflects only one of the failures. In such cases, it may not be possible to determine which failure the error number value is for.

NX_LOAD() *(cont.)*

NX_LOAD() *(cont.)*

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* array. If the *numnodes* parameter is set to -1, the application is loaded onto all the application's nodes (the *node_list* parameter is ignored).

ptype Process type of the new process(es).

pathname Pathname of the executable image to load and start.

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 the *node_list* parameter. An entry of 0 (zero) indicates that the process on the corresponding node was not started successfully. The *pid_list* array needs to 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 Must be 0 (zero). Any other value causes an error.

envp Must be 0 (zero). Any other value causes an error.

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

Description

The **nx_load()** and **nx_loadve()** functions load and start an executable image on the nodes specified by *node_list*. These calls can only be made after the calling process makes an initial **nx_initve()** call.

The **nx_loadve()** function is provided for compatibility with the corresponding Paragon™ OSF/1 C system call. See the *Paragon™ System C Calls Reference Manual*. The **nx_loadve()** function is identical to the **nx_load()** function, except for the *argv* and *envp* parameters. These parameters are not supported in the Fortran version of the **nx_loadve()** function. You specify a 0 (zero) value for these parameters. The **nx_load()** function is preferable for Fortran applications.

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

Return Values

> 0	Number of child processes started successfully.
-1	Error. Use the **nx_perror()** subroutine to display the error message for the current error.

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()**

NX_MKPART() NX_MKPART()

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

Synopsis

INCLUDE 'fnx.h'

INTEGER FUNCTION **NX_MKPART**(*partition*, *size*, *type*)

CHARACTER*(*) *partition*
INTEGER *size*
INTEGER *type*

INTEGER FUNCTION **NX_MKPART_RECT**(*partition*, *rows*, *cols*, *type*)

CHARACTER*(*) *partition*
INTEGER *rows*
INTEGER *cols*
INTEGER *type*

INTEGER FUNCTION **NX_MKPART_MAP**(*partition*, *numnodes*, *node_list*, *type*)

CHARACTER*(*) *partition*
INTEGER *numnodes*
INTEGER *node_list*(*)
INTEGER *type*

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

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.

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 in the parent partition available to the new partition.

Description

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

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

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

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

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 or the **chpart** command.

Return Values

> 0	Number of nodes allocated for the partition.
-1	Error. Use the **nx_perror()** subroutine to display the error message for the current error.

Errors

`Partition permission denied`

The application has insufficient access permission on a partition.

`Allocator internal error`

An internal error occurred in the node allocation server.

`Bad node specification`

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.

`Partition not found`

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

`Partition exists`

The specified partition already exists.

`Exceeds partition resources`

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.

Synopsis

INCLUDE 'fnx.h'

INTEGER FUNCTION **NX_MKPART_ATTR**(*partition*, *[attribute, value,]*...
NX_ATTR_END)

CHARACTER*(*) *partition*
INTEGER *attribute*
CHARACTER* | INTEGER *value*

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.

NX_MKPART_ATTR() *(cont.)*

NX_MKPART_ATTR() *(cont.)*

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.

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

NOTE

If you call **nx_mkpart_attr()** in a subprogram, you must include *fnx.h* after the subprogran declaration and before the call. This is required for the call to recognise the pre-defined attribute constants (for example, **NX_ATTR_SZ**).

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_MKPART_ATTR() *(cont.)*

NX_MKPART_ATTR() *(cont.)*

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_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_MKPART_ATTR() *(cont.)*

NX_MKPART_ATTR() *(cont.)*

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_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_MKPART_ATTR() *(cont.)*

NX_MKPART_ATTR() *(cont.)*

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*.
[*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 *ntypes*, 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.)*

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

`Invalid argument`

Invalid attribute specified in the *attribute* parameter, including error in the Some nodes in the map or rectangle do not qualify attribute selector.

`Partition permission denied`

The application has insufficient access permission on a partition.

`Allocator internal error`

An internal error occurred in the node allocation server.

`Bad node specification`

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

`Bad partition request`

Partition request contains bad or missing nodes.

`Partition not found`

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

`Partition lock denied`

Partition is currently in use or being updated.

`Attributes do not match`

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.

`Partition exists`

The specified partition already exists.

`Exceeds partition resources`

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 'fnx.h'

INTEGER FUNCTION NX_NFORK(node_list, num_nodes, ptype, pid_list)

INTEGER node_list(*)
INTEGER num_nodes
INTEGER ptype
INTEGER 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`".

num_nodes Number of nodes in the *node_list* array. If you set *num_nodes* to -1, **nx_nfork()** assumes all nodes of the application and ignores the value of 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 *num_nodes* parameter equals -1, the first element of the *pid_list* array equals the PID of node 0, the second element of the *pid_list* array equals the PID of node 1, and so on for all the nodes in the system.

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. 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).
- Use the **nx_perror()** subroutine to display the error message for the current error.

NOTE

It is possible that the process could not be successfully forked on more than one node, and that each failure could be for a different reason. In this case, the value of error number reflects only one of the failures. It may not be possible to determine which failure the error number value is for.

Errors

`Allocator internal error`

An internal error occurred in the node allocation server.

`Bad node specification`

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()**

NX_PART_ATTR() NX_PART_ATTR()

Returns information about a partition.

Synopsis

INCLUDE 'fnx.h'

INTEGER FUNCTION **NX_PART_ATTR**(*partition*, *attributes*)

CHARACTER*(*) *partition*
RECORD /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 *fnx.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 *fnx.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.

Examples

The following example prints the rollin quantum and effective priority limit for the partition *mypart*:

```
include 'fnx.h'

record /nx_part_info_t/ info
integer status

status = nx_part_attr("mypart", info)

if(status .ne. 0) then
   call nx_perror("nx_part_attr()")
   stop
end if

print *, "rq =",info.rq,", epl =",info.epl

end
```

NX_PART_ATTR() *(cont.)* NX_PART_ATTR() *(cont.)*

Errors

`Partition permission denied`

The application has insufficient access permission on a partition.

`Partition not found`

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

Paragon™ System Fortran Calls Reference Manual: **nx_chpart_epl()**, **nx_pspart()**, **nx_part_nodes()**

Paragon™ XP/S System Commands Reference Manual: **chpart**, **lspart**, **pspart**, **showpart**

NX_PART_NODES() NX_PART_NODES()

Returns the root partition node numbers for a partition.

Synopsis

INCLUDE 'fnx.h'

INTEGER FUNCTION **NX_PART_NODES**(*partition*, *ptr*, *list_size*)

CHARACTER*(*) *partition*,
POINTER *(ptr, node_list(1))*
INTEGER *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.
ptr	Pointer variable to the integer array *node_list* into which the **nx_part_nodes()** function stores the address of a list of nodes in *partition*. The call allocates memory for this parameter. Free this memory using the **free()** function.
list_size	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 'fnx.h'

      integer*4  mynodes(1)
      pointer    (ptr, mynodes)
      integer    nnodes
      integer    i, status

      status = nx_part_nodes("mypart", ptr, nnodes)

      if(status .ne. 0) then
         call nx_perror("nx_part_nodes()")
         stop
      end if

      do 2, i = 1, nnodes
         print *, mynodes(i)
2     continue

      call free(ptr)

      end
```

Errors

`Partition permission denied`

The application has insufficient access permission on a partition.

`Partition not found`

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*.

NX_PART_NODES() *(cont.)*

NX_PART_NODES() *(cont.)*

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 'fnx.h'

SUBROUTINE **NX_PERROR**(*string*)

CHARACTER*(*) *string*

Parameters

string	Error message you want to display.

Description

Use the **nx_perror()** subroutine with **nx_...** calls to display the error message for the current error. The **nx_perror()** subroutine prints its argument (any string), the current node number and process type, and the error message associated with the current error number to the standard error output in the following format:

```
(node n, ptype p) string: error_message
```

Limitations and Workarounds

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

See Also

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

NX_PRI() NX_PRI()

Sets the priority of an application.

Synopsis

```
INCLUDE 'fnx.h'

INTEGER FUNCTION NX_PRI(pgroup, priority)

INTEGER pgroup
INTEGER 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.

NX_PRI() *(cont.)* NX_PRI() *(cont.)*

Return Values

> 0	No errors; priority successfully set.
-1	Error.Use the **nx_perror()** subroutine to display the error message for the current error.

Errors

`Allocator internal error`

An internal error occurred in the node allocation server.

`Application does not exist for process group`

The specified process group does not exist.

`Not owner`

The calling process does not have permissions to change the application's priority.

`No such process`

The specified process group does not exist.

`Priority out of range`

The specified priority is out of the range of priority values.

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 'fnx.h'

INTEGER FUNCTION **NX_PSPART**(*partition*, **NX_PSPART_T**)

INTEGER *partition*
INTEGER *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.

NX_PSPART_T
Structure into which the **nx_pspart()** function stores information about an application or active partition in the partition specified by the *partition* parameter. The **nx_pspart_t** structure is defined in the include file *allocsys.h*, which is included by the include file *fnx.h*.

list_size Variable into which the **nx_pspart()** function stores the number of elements in the **NX_PSPART_T** structure.

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**).

NX_PSPART() *(cont.)* NX_PSPART() *(cont.)*

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 owner.

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.)*

Errors

`Partition permission denied`

The application has insufficient access permission on a partition.

`Partition not found`

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 'fnx.h'

INTEGER FUNCTION NX_RMPART( pathname, force, recursive)

CHARACTER*(*) partition
INTEGER force
INTEGER 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 e (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 removes 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 a 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. Use the **nx_perror()** subroutine to display the error message for the current error.

Errors

`Allocator internal error`

An internal error occurred in the node allocation server.

`Partition lock denied`

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

NX_RMPART() *(cont.)* NX_RMPART() *(cont.)*

`Partition not empty`

The specified partition contains one or more subpartitions or running applications.

`Partition not found`

The specified partition does not exist.

`Partition permission denied`

Insufficient access permission for this operation on a permission.

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 'fnx.h'

INTEGER FUNCTION NX_WAITALL()
```

Description

The **nx_waitall()** function suspends the application's calling process until all the application's child processes stop or terminate. An application can start child processes can with the **nx_nfork()** or **nx_load()** 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

`Interrupted system call`

The function was terminated by receipt of a signal.

`No child processes`

The calling process has no existing child processes to wait for.

NX_WAITALL() *(cont.)* NX_WAITALL() *(cont.)*

Limitations and Workarounds

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

See Also

nx_nfork(), **nx_load()**

SETIOMODE() SETIOMODE()

Sets a file's I/O mode and performs a global synchronization operation.

Synopsis

INCLUDE 'fnx.h'

SUBROUTINE **SETIOMODE**(*unit*, *iomode*)

INTEGER *unit*
INTEGER *iomode*

Parameters

unit Unit number (an integer between 1 and 100) assigned when the file was opened.

iomode I/O mode to be assigned to the file associated with *unit*. 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()** subroutine 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.

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 *unit* number representing the opened file, and the file pointer must be in the same position in the file for each node at the time the call to **setiomode()** is made.

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 **iomode()** 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.

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()** subprograms 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()** subprograms 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()** subprograms 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 used when opening PFS files by setting the *PFS_ASYNC_DFLT* bootmagic string to 1.

SETIOMODE() *(cont.)* SETIOMODE() *(cont.)*

Errors

NOTE

The majority of the Fortran I/O errors that you are likely to receive are described in the "Runtime Error Messages" appendix of the *Paragon™ System Fortran Compiler User's Guide*. This section describes additional errors that you may receive.

`Bad file number`

Use the unit assigned when the file was opened.

`Bad I/O mode number`

I/O mode must be set to **M_UNIX**, **M_LOG**, **M_SYNC**, **M_RECORD**, or **M_GLOBAL**.

`File is not synchronized`

In the I/O modes **M_LOG**, **M_SYNC**, **M_RECORD**, or **M_GLOBAL**, all nodes must set the file pointer to the same location.

`Fortran runtime error: Unit not open`

A file must be open to set its I/O mode.

`Invalid argument`

The given value for *iomode* is not valid.

`Invalid argument`

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

`No such unit`

The *unit* number must be a value no larger than 100.

SETIOMODE() *(cont.)* SETIOMODE() *(cont.)*

Examples

The following example shows how to use the **setiomode()** subroutine to set the I/O mode after opening a file, but before writing to the file.

```
      include 'fnx.h'

      integer      iam
      character*14 buf

c  Identify self.

      iam = mynode()

c  Globally open file with the M_UNIX I/O mode.

      call gopen(12, '/tmp/mydata', M_UNIX)

c  You can read the file and do some computation before
c  changing the I/O mode.

      call setiomode(12, M_RECORD)

c  Write and close the file.

      buf = 'Hello, world!\n'
      call cwrite(12, buf, len(buf))

      write(*, 100) iam, buf
100   format('Node ', i3, ' wrote: ', a13)

      close(12)

      end
```

Limitations and Workarounds

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

See Also

cread(), **cwrite()**, **fopen()**, **fork()**, **gopen()**, **iomode()**, **iread()**, **iwrite()**, **open()**

SETPTYPE() SETPTYPE()

Sets the process type of the calling process.

Synopsis

INCLUDE 'fnx.h'

SUBROUTINE **SETPTYPE**(*ptype*)

INTEGER *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()** subroutine 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()** subroutine has the following restrictions:

- Do not use the **setptype()** subroutine 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()** subroutine in processes created with the **nx_nfork()**, **nx_load()**, or **nx_loadve()** subprograms. These subprograms have a *ptype* parameter for specifying the process type of newly created processes in an application.
- Do not use the **setptype()** subroutine in controlling processes that do not use message passing, because the **setptype()** subroutine 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()** subroutine.)

The **fork()** system call creates a new process on the same node as the process that calls it. The **fork()** system call 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()** subroutine 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.

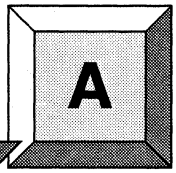
Limitations and Workarounds

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

See Also

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

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()** calls. Force types have three special properties:

1. A message with a force type bypasses the normal flow control mechanisms and is not delayed by clogged message buffers on the 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 hexadecimal numbers associated with the *types* you want to select to the constant, `0x80000000`. For example, if you want to receive message types `1`, `2`, `5`, and `12`, add the following hex numbers:

```
0'2'x, 0'4'x, 0'20'x, 0'1000'x + 0'80000000'x = 0'80001026'x
```

then enter

```
crecv (0'80001026'x, buf, len);
```

Or, if you want to receive any message except type `0` use:

```
crecv (0'FFFFFFFE'x, 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	0'00000001'x
1	0'00000002'x
2	0'00000004'x
3	0'00000008'x
4	0'00000010'x
5	0'00000020'x
6	0'00000040'x
7	0'00000080'x
8	0'00000100'x
9	0'00000200'x
10	0'00000400'x
11	0'00000800'x
12	0'00001000'x
13	0'00002000'x
14	0'00004000'x
15	0'00008000'x
16	0'00010000'x
17	0'00020000'x
18	0'00040000'x
19	0'00080000'x
20	0'00100000'x
21	0'00200000'x
22	0'00400000'x
23	0'00800000'x
24	0'01000000'x
25	0'02000000'x

Table A-1. Typesel Mask List (2 of 2)

Type	Hex Number
26	0'04000000'x
27	0'08000000'x
28	0'10000000'x
29	0'20000000'x
Other types	0'40000000'x

Index

H

I

L

M

N

S