August 1991

Change Notice Order Number: 312208-001

Change Notice

for the

iPSC®/2 and iPSC®/860 Programmer's Reference Manual

Manual Order Number: 311708-004

intel® Corporation

Section or Chapter	Remove These Pages	Insert These Pages
Disclaimer	iii, iv	iii, iv
Table of Contents	ix through xviii	ix through xviii
Chapter 1	all of Chapter 1	the new Chapter 1

REV.	REVISION HISTORY	DATE
-001	Original Issue	03/89
-002	Revision	10/89
-003	Revision	06/90
-004	Revision	04/91
	Revised by Change Notice 312208-001	08/91

RESTRICTED RIGHTS

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the rights in Technical Data and Computer Software clause at 52.227-7013. Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051.

TABLE OF CONTENTS

CHAPTER 1
iPSC®/2 and iPSC®/860 COMMANDS

INTRODUCTION 1-1

APROPOS 1-3

ARCHCUBE 1-4

ATTACHCUBE 1-6

BOOTCUBE 1-8

CBACKUP 1-14

CPTAPE 1-17

CRESTORE 1-18

CUBEINFO 1-21

FSPLIT 1-24

GETCUBE 1-25

HEAD 1-31

KILLCUBE 1-32

LESS 1-34

LOAD 1-40

LSIZE 1-43
MAKEWHATIS 1-45
MAN 1-46
MANPATH 1-50
MKCFS 1-52
MKDEV 1-55
MT 1-57
NEWSERVER 1-59
NSH 1-61
PLOGON, PLOGOFF 1-63
PROF860 1-67
REBOOTCUBE 1-69
RELCUBE 1-70
REMOTE HOST 1-72
SHOWVOL 1-76
STAR 1-78
STARTCUBE 1-81
STREAM 1-83
SYSLOG 1-84
TAPEMODE 1-86
WAITCUBE 1-90
WHATIS 1-93

CHAPTER 2
iPSC®/2 and iPSC®/860 C LIBRARY CALLS

INTRODUCTION 2-1

ATTACHCUBE() 2-3

CPROBE() 2-6

CREAD() 2-9

CREATESTRUC(), RELSTRUC() 2-11

CRECV() 2-15

CSEND() 2-18

CSENDRECV() 2-22

CTOHx(), HTOCx() 2-28

CUBEINFO() 2-31

CWRITE() 2-35

DCLOCK() 2-38

EADD(), ECMP(), EDIV(), EMOD(), EMUL(), ESUB() 2-40

ESEEK() 2-44

ESIZE() 2-47

ESTAT(), FESTAT() 2-50

ETOS(), STOE() 2-53

FLICK() 2-55

FLUSHMSG() 2-58

GCOL() 2-61

GCOLX() 2-64

GETCUBE() 2-68

GETIPHOSTS() 2-74
GINV() 2-77
GOPF() 2-79
GRAY() 2-83
GSENDX() 2-85
GSYNC() 2-87
GxAND() 2-88
GxHIGH() 2-91
GxLOW() 2-94
GxOR() 2-97
GxPROD() 2-100
GxSUM() 2-103
GxXOR() 2-106
HANDLER() 2-109
HRECV() 2-113
HSEND() 2-117
HSENDRECV() 2-121
HWCLOCK() 2-127
INFO: INFOCOUNT(), INFONODE(), INFOPID(), INFOTYPE() 2-129
IODONE() 2-132
IOMODE() 2-134
IOWAIT() 2-136
IPROBE() 2-138
IREAD() 2-141

IRECV() 2-144
ISEND() 2-148
ISENDRECV() 2-152
ISEOF() 2-158
IWRITE() 2-160
KILLCUBE() 2-163
KILLPROC() 2-165
KILLSYSLOG() 2-167
LED() 2-169
LOAD() 2-171
LSIZE() 2-175
MASKTRAP() 2-178
MCLOCK() 2-180
MSGCANCEL() 2-182
MSGDONE() 2-184
MSGWAIT() 2-187
MYHOST() 2-189
MYNODE() 2-191
MYPID() 2-193
NEWSERVER() 2-195
NODEDIM() 2-198
NUMNODES() 2-200
PLOGON(), PLOGOFF() 2-202
RELCUBE() 2-205

RESTRICTVOL() 2-207
SETIOMODE() 2-211
SETIPHOST() 2-215
SETPID() 2-217
SETSYSLOG() 2-220
WAITALL() 2-223
WAITONE() 2-225

CHAPTER 3
iPSC®/2 and iPSC®/860 FORTRAN ROUTINES

ATTACHCUBE() 3-3
CPROBE() 3-6
CREAD() 3-9
CRECV() 3-12
CSEND() 3-16
CSENDRECV() 3-20
CTOHx(), HTOCx() 3-25
CUBEINFO() 3-28
CWRITE() 3-32
DCLOCK() 3-36
EADD(), ECMP(), EDIV(), EMOD(), EMUL(), ESUB() 3-38
ESEEK() 3-42
ESIZE() 3-45
ETOS(), STOE() 3-48

FLICK() 3-50
FLUSHMSG() 3-53
FORCEFLUSH() 3-56
FORFLUSH() 3-58
FPSETMASK() 3-60
GCOL() 3-62
GCOLX() 3-65
GETCUBE() 3-68
GINV() 3-73
GOPF() 3-75
GRAY() 3-79
GSENDX() 3-81
GSYNC() 3-83
GxAND() 3-84
GxHIGH() 3-86
GxLOW() 3-89
GxOR() 3-92
GxPROD() 3-94
GxSUM() 3-97
GxXOR() 3-100
HANDLER() 3-102
HRECV() 3-106
HSEND() 3-110
HSENDRECV() 3-115

HWCLOCK() 3-122

INFO: INFOCOUNT(), INFONODE(), INFOPID(), INFOTYPE() 3-124

IODONE() 3-128

IOMODE() 3-130

IOWAIT() 3-132

IPROBE() 3-134

IREAD() 3-137

IRECV() 3-141

ISEND() 3-145

ISENDRECV() 3-149

ISEOF() 3-155

IWRITE() 3-158

KILLCUBE() 3-161

KILLPROC() 3-163

KILLSYSLOG() 3-165

LED() 3-167

LOAD() 3-169

LSEEK() 3-172

LSIZE() 3-175

MASKTRAP() 3-178

MCLOCK() 3-180

MSGCANCEL() 3-182

MSGDONE() 3-184

MSGWAIT() 3-187

MYHOST() 3-190

MYNODE() 3-192

MYPID() 3-194

NEWSERVER() 3-196

NODEDIM() 3-199

NUMNODES() 3-201

PLOGON(), PLOGOFF() 3-203

RELCUBE() 3-206

RESTRICTVOL() 3-208

SETIOMODE() 3-212

SETPID() 3-216

SETSYSLOG() 3-219

WAITALL() 3-221

WAITONE() 3-223

APPENDIX A
iPSC® SYSTEM COMMANDS, SYSTEM CALLS, and ROUTINES

APPENDIX B
iPSC® SYSTEM TYPES and TYPESEL MASKS

TYPES B-1

TYPESEL MASKS B-2

APPENDIX C
ERROR HANDLING IN C

LIST OF TABLES

Table A-1. SRM Cube Command Summary A-1

Table A-2. Summary of System Calls for Cube Control (C Version) A-3

Table A-3. Summary of Routines for Cube Control (Fortran Version) A-5

Table A-4. Summary of System Calls for Message Passing (C Version) A-7

Table A-5. Summary of Routines for Message Passing (Fortran Version) A-10

Table A-6. Global Operations (C Version) A-13

Table A-7. Global Operations (Fortran Version) A-16

Table A-8. Byte-Swapping System Calls for Remote Host/Cube Communication (C Version) A-19

Table A-9. Byte-Swapping Routines for Remote Host/Cube Communication (Fortran Version) A-20

Table A-10. Miscellaneous System Calls (C Version) A-21

Table A-11. Miscellaneous Routines (Fortran Version) A-22

Table A-12. Summary of System Calls for Concurrent File I/O (C Version) A-23

Table A-13. Summary of Routines for Concurrent File I/O (Fortran Version) A-25

Table A-14. Summary of Mathematical System Calls (C Version) A-27

Table A-15. Summary of Mathematical Routines (Fortran Version) A-28

Table A-16. C-Shell Built-Ins and UNIX Utilities That Run Under the Node's C-Shell A-29

Table A-17. UNIX-Compatible System Calls (C Version) A-30

Table A-18. UNIX Compatible I/O Library Calls (C Only Version) A-31

Table A-19. Summary of SRM UNIX Extensions A-32

Table A-20. Node's C-Shell Cube Command Summary A-33

Table A-21. Unique Node TCP/IP System Calls A-34

Table C-1. iPSC®/2 and iPSC®/860 Error Messages C-2

iPSC®/2 and iPSC®/860 COMMANDS 1

INTRODUCTION

This chapter documents all of the iPSC commands. These commands allow you to perform tasks such as accessing and releasing cubes, and loading, starting, and killing cube processes. You can invoke these commands from your terminal or from a shell script.

Most of the commands have corresponding system library calls so you can initiate these functions from user programs. The C and Fortran versions of the system library calls are described in the chapters following this one. Refer to Appendix A for a tabular description of all commands, routines, and calls.

NOTE

The following five commands are UNIX extensions and do not affect operation of the cube: **cptape**, **head**, **less**, **star**, and **stream**. For user ease, these commands have been included and alphabetized within this chapter.

The following SRM commands are located in the directory */usr/bin*:

archcube	**killcube**	**plogon**	**rld**
attachcube	**less**	**rar**	**showvol**
bootcube	**load**	**ras**	**star**
cptape	**mkcfs**	**rcc**	**startcube**
cubeinfo	**newserver**	**rebootcube**	**stream**
getcube	**nsh**	**relcube**	**syslog**
head	**plogoff**	**rf77**	**waitcube**

The following NSH commands are located either in the directory */usr/bin* (Intel386™ versions) or in the directory */usr/i860/ipsc/bin* (i860™ versions):

archcube	**killcube**	**mt**	**tapemode**
cbackup	**load**	**showvol**	**waitcube**
crestore	**mkdev**		

Note that the **archcube, killcube, load, showvol,** and **waitcube** commands are located in all three directories. They run on both the SRM and the nodes, although they have different switches. The following descriptions note the applicable differences.

APROPOS APROPOS

Search the *whatis* database for strings. This command describes the **man** implementation on the SRM only.

Syntax

apropos *keyword ...*

Arguments

keyword	One or more words separated by spaces.

Description

apropos searches a set of database files containing short descriptions of system commands for keywords and displays the result on the standard output.

Examples

```
% apropos flush
FLUSHMSG (3)          - Flush specified messages from the system.
FLUSHMSG (3f)         - Flush specified messages from the system.
FORCEFLUSH (3f)          - Forces all buffered I/O to be flushed
when an exception occurs.
FORFLUSH (3f)            - Forces completion of all buffered I/O
on a file specified by its unit number.
```

See Also

man, manpath, whatis

ARCHCUBE ARCHCUBE

Display the architecture of the currently attached or specified cubename.

SRM Syntax

archcube [**-c** *cubename*]

NSH Syntax

archcube

Arguments

-c *cubename*	Allows you to get the architecture of the specified cube. This argument is not valid from the NSH version of **archcube.**

Description

The **archcube** command displays the architecture of either the currently attached cube or the cube specified by **-c** *cubename*. The command outputs to *stdout* the string `i860` if architecture is an 80860 or the string `i386` if the architecture is an 80386.

This command is available from either the SRM or NSH. The NSH version does not allow the **-c** *cubename* argument however.

Example

This example uses **archcube** to determine which version of *ps* to load on a node:

```
CUBEARCH = 'archcube'
if [$CUBEARCH = "i860"]
then
     load 0 /usr/i860/ipsc/bin/ps
else
     load 0 /usr/ipsc/bin/ps
fi
```

ARCHCUBE *(cont.)*

ARCHCUBE *(cont.)*

Errors

`setpid: Bootcube in progress`

A **bootcube** is in progress. Try again when the **bootcube** is completed.

ATTACHCUBE ATTACHCUBE

Make the default or specified *cubename* the current attached cube.

Syntax

attachcube [**-c** *cubename*]

Arguments

-c *cubename* Allows you to attach to a specific cube of the given name.

Description

Use the **attachcube** command when you own more than one cube (allocated with the **getcube** command) and you want to change the current cube to another of your cubes. The cube most recently allocated is the current cube. You can be attached to only one cube at a time. If you specify multiple names in the command, an error is returned.

The **-c** option requires a *cubename* defined with the **getcube** command, which requires an ASCII character string, 15 characters or less in length. If you do not use this argument, the cube with the default name "defaultname", is attached. If the specified cube does not exist, an error is returned.

Subsequent message-passing and load operations (such as **csend, crecv, load, killcube**) apply to the current cube only. For a list of the cube names you have allocated, use the **cubeinfo -a** command.

The effect of this command differs from the effect of the **attachcube** routine in that the command changes the shell's current cube, while the routine changes only a host process's current cube, leaving the shell unaffected.

ATTACHCUBE *(cont.)*

ATTACHCUBE *(cont.)*

Example

Suppose that in a 32-node system you used the **getcube** command as follows to allocate three 8-node cubes:

```
getcube -t8
getcube -c alpha -t8
getcube -c beta -t8
```

The first cube is assigned the default name "defaultname." The current cube is the last cube you got (in this case, the one named "beta"). To attach the first cube allocated, enter the command:

```
attachcube
```

This automatically looks for a cube named "defaultname." To attach the cube named "alpha," enter the command:

```
attachcube -c alpha
```

Errors

`attachcube: Cubename does not exist`

Use **cubeinfo** to determine cube names.

`attachcube: Commser not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with the **bootcube** or **rebootcube** command.

`attachcube: Lifeline not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with the **bootcube** or **rebootcube** command.

See Also

getcube, cubeinfo, relcube

BOOTCUBE BOOTCUBE

Load the node operating system onto the nodes of the cube. This command is for use by the system administrator only.

Syntax

bootcube [**-B***file*] [**-b***file*] [**-C**] [**-D***n* | **-D***m-n*] [**-d** *dim* | **-n** *nodes*] [**-f**] [**-K***file*][**-k***file*] [**-L**] [**-l**] [**-o**] [**-Q***file*] [**-R***file*] [**-r***file*] [**-S***file*] [**-s***file*] [**-T***file*] [**-U***file*]

Arguments

-B*file* Reads an alternative hardware definition file; the default is */usr/ipsc/lib/hbits.x*, where x is an extension consisting of a single alphabetic character, indicating a revision level. This is a binary file.

CAUTION

Loading bad bits can destroy the DCM hardware.

-b*file* Loads an alternative CX node bootloader; the default is */usr/ipsc/lib/bootld.*

-C Causes configuration information to be displayed during the bootcube process.

-D*n*; **-D***m-n* Runs the specified steps. **-D***n* stops execution after step *n*. **-D***m-n* runs steps *m* to *n* (for example, **-D3-4** skips steps 1 and 2 and runs steps 3 and 4). Use only one **-D** on the invocation line. To list the steps without actually executing them, use **-L**. This switch is valid on the remote host.

CAUTION

Running **bootcube** without beginning at step 1 (for example, ***bootcube -D3-24***) will generally not produce a fully operational system.

BOOTCUBE *(cont.)*

BOOTCUBE *(cont.)*

-d *dim*; **-n** *nodes* Resets and reloads the NX/2 operating system into a cube the size of **-d** *dim*ension or **-n** *nodes*. If the size is not specified, the entire cube is reloaded. If you enter a size that is larger than the one specified in the cube configuration file, an error is returned. If the number of nodes you specify is not a power of two, the next higher power of two is used. All other nodes will be nonfunctional.

-f Forces **bootcube** to proceed even though all cubes have not been released. Use of this option may leave file server processes running which must be killed manually. Use of this option may also leave open **bootcube** lock files that must be deleted. This switch is valid on the remote host.

-K *file* Loads an alternative CX node operating system; the default is */usr/ipsc/lib/nx.b*.

-k *file* Loads an alternative RX node operating system; the default is */usr/i860/ipsc/lib/nx.b*.

-L Lists the **bootcube** steps without executing them. This switch is valid on the remote host.

-l Disables use of the cube from remote development machines.

-o Boots all except the I/O nodes. I/O nodes interface to disks, tapes, Ethernet controllers, and custom devices.

-Q *file* Uses an alternative cube configuration file; the default is */usr/ipsc/conf/cubeconf*.

-R *file* Uses an alternative shutdown shell file; the default is */usr/ipsc/lib/rc0*. This switch is valid only on the remote host.

-r *file* Loads an alternative RX node bootloader; the default is */usr/i860/ipsc/lib/bootld*.

-S *file* Uses an alternative start-up shell file; the default is */usr/ipsc/lib/rc1*. This switch is valid on the remote host.

-s *file* Loads alternative socket driver; the default is */usr/ipsc/lib/sockproc*.

-T *file* Uses an alternative start-up shell file; the default is */usr/ipsc/lib/rc2*.

-U *file* Uses an alternative start-up shell file; the default is */usr/ipsc/lib/rc3*.

BOOTCUBE *(cont.)*

BOOTCUBE *(cont.)*

Description

The **bootcube** command loads the NX/2 operating system into the cube and starts up system daemons. Only root can run **bootcube**. This command returns an error unless the entire cube is available (no users can have cubes allocated) unless you use the **-f** option. The **rebootcube** command is available to users who are not root, but need to reload the NX/2 operating system onto the nodes.There is a separate version of **bootcube** specifically for starting the appropriate daemons on the remote host machine. You must be root on the remote workstation to run the command. The only flags that can be used with this version are **-L**, **-D**, **-f**, **-R**, and **-S**.

The **bootcube** command affects all of the nodes in an iPSC system. Therefore, if more than one user has cubes on the system, and one user requires that the cube be rebooted, all cubes are affected.

The **bootcube** disables the **getcube** command while it is executing. If any of the cube utility commands or library calls (for example, **getcube**, **getcube()**, **showvol**, etc.) is attempted while a **bootcube** is in progress, a message to that effect is printed and the cube utility command is aborted. In addition, host programs that call **setpid()** while a **bootcube** is in progress return with an error message similar to that for cube utility commands and library calls.

Examples

1. Load the NX/2 operating system into entire cube:

```
# bootcube
```

2. Reset and load NX/2 into a 8-node cube:

```
# bootcube -n8
```

3. Load an alternative operating system:

```
# bootcube -K /usr/joelc/my_os
```

BOOTCUBE *(cont.)*

BOOTCUBE *(cont.)*

Errors

`bootcube must be run by superuser`

Only the superuser can run **bootcube**.

`You must release all cubes before running bootcube`

Unless you use the **-f** switch, **bootcube** will not run if there are allocated cubes.

`Steps specified for -D out of order`

If you specify **-D***m-n*, *m* must be less than *n*.

xxx `cannot open`

bootcube cannot open the file *xxx*.

`File` *xxx* `cannot be found`

bootcube cannot find the file *xxx*.

`Bootcube aborted. Another bootcube in progress`

The presence of lock file */usr/spool/locks/LCK..bootcube* was detected. This lock file will still exist after interrupting and terminating a **bootcube** in progress. You can use the **-f** switch to force **bootcube** to ignore the existence of a lock file and proceed with the **bootcube**.

`Invalid number of slots:` *xxx* `maximum:` *xxx*

The **bootcube -d** switch specifies more slots than the configuration file.

`Failure trying to download operating system`

The system hung while **bootcube** was downloading the NX/2 operating system. Retry at least once before calling SSD Customer Support.

`Node 0 failed: Cannot load operating system`

Node 0 must be working if **bootcube** is to successfully load the NX/2 operating system on the nodes.

BOOTCUBE *(cont.)*

BOOTCUBE *(cont.)*

`Error occurred while reading a port on the SRM direct connect channel`

The SRM Direct-Connect Module has a problem. Call SSD Customer Support.

`Error occurred while performing DCM reset` *x* `on the SRM direct connect channel`

The SRM Direct-Connect Module has a problem. Call SSD Customer Support.

`Error occurred while initializing hardware on the SRM direct connect channel`

The SRM Direct-Connect Module has a problem. Call SSD Customer Support.

`diagnostic link:` *xxxx*

Internal **bootcube** error. Call SSD Customer Support.

`diagnostic link: Timeout`

Nodes are unresponsive. Check cable connections.

`loadboot` *xxxx*

Internal **bootcube** error. Call SSD Customer Support.

`querycube` *xxxx*

Internal **bootcube** error. Call SSD Customer Support.

`resetcube Invalid diagnostic channel mode`

Internal **bootcube** error. Call SSD Customer Support.

`Reset timed out`

Nodes are unresponsive. Check cable connections.

`starting socket process: No such device`

The ENET address or the ENET board has a problem.

BOOTCUBE *(cont.)*

BOOTCUBE *(cont.)*

`setavail xxxx`

Internal **bootcube** error. Call SSD Customer Support.

`testcube xxxx`

Internal **bootcube** error. Call SSD Customer Support.

See Also

rebootcube

CBACKUP CBACKUP

Back up one or more CFS disk drives. This command is for use by the system administrator only.

NSH Syntax

cbackup [**-s** *mediasize*] [**-a**] [*volume...*] [*filename*]

Arguments

-s *mediasize*	Allows you to specify the size in bytes of the media used for backup.
-a	Backs up all of the blocks in each volume, whether the blocks contain data or not. Without this parameter, only those blocks containing data are backed up.
volume...	A space-separated list of the numbers of the disk drives (volumes) that you want to back up. If no volume numbers are listed, the command will back up all volumes.
filename	The name of the file or device to which the backup is to be written. If no file or device is named, the backup is written to *stdout*, allowing you to pipe the output to other commands, such as **tar**.

Description

NOTE

This command only operates within the **nsh** command.

The **cbackup** command lets you back up the volumes (disk drives) of the concurrent file system. The *filename* parameter is the name of a file or device (such as */cfs/tape*) to which you are backing up the data. Without the *filename* parameter, the backup goes to *stdout*. Use the **crestore** command to restore the backup. It is good practice to issue a **getcube** to reserve an entire cube before performing backup procedures.

By default, the **cbackup** command backs up only those blocks that contain data. However, if you use the **-a** option, the entire disk image will be backed up, regardless of whether blocks contain data. Data passed to the backup device are buffered up to 256K bytes.

CBACKUP *(cont.)* CBACKUP *(cont.)*

The *volume* parameter lets you specify the volume number(s) that you want to back up. With no volume numbers specified, all disk drives are backed up.

The **-s** parameter lets you specify the size (in bytes) of the media to which you are backing up the information. If you have more data to back up than the tape can hold, specify the size to avoid losing data. A "K" after the number specifies kilobytes (1024 bytes). For example, 600K bytes is interpreted as 600 kilobytes. Similarly, an "M" after the number specifies megabytes (1048576 bytes). A "G" after the number specifies gigabytes. The number before the magnitude specifier must be a whole number. The standard cartridge tape size is 45M bytes. The 2-hour, 8mm tape is 2.2G bytes.

Example

To back up the data on volumes 1 and 2 to a tape, designated as the device */cfs/tape*, specifying a maximum backup space of 500M bytes, enter the following:

```
# nsh
# cbackup -s 500M 1 2 /cfs/tape
```

Errors

`cbackup: Media size too small`

Ensure that you specify the media size correctly or use larger backup media.

`cbackup: Bad disk number`

Use **showvol** to find valid disk numbers.

`cbackup: Cannot create` *nnn*

File name is invalid. Use the UNIX filename conventions.

`cbackup: Disk Process not initialized`

Reboot the cube and initialize the disk processes.

`cbackup: Not enough memory for bitmap`

Not enough room for the buffer on the SRM disk. Obtain more space on SRM disk.

CBACKUP *(cont.)*

CBACKUP *(cont.)*

`cbackup: Bitmap error`

Error in the disk process. Try rebooting with a **bootcube** or **rebootcube** command.

`cbackup: Write error`

Problem with the backup media. Check to see that the end of the tape is not overwritten.

See Also

crestore

CPTAPE CPTAPE

Provides a simplified user interface to the **cpio** command.

Syntax

cptape {-i|-o|-t} [-f *devicename*] *filename...*

Arguments

-i	reads a tape (passes **-iducvm** to **cpio**).
-o	writes a tape (passes **-ocu** to **cpio**).
-t	displays what is available on tape (passes **-ictu** to **cpio**).
-f *devicename*	name of the device that is the target of the **cpio** command.
filename	the directory or files to archive (**-o**) or extract (**-i**).

Description

The **cptape** command is a shell script located at */usr/bin/cptape* that archives or extracts files from a **cpio** format device or file.

Examples

The following example extracts files from a **cpio** format tape on device */dev/tape*:

```
cptape -i
```

The following example archives the files located in */usr/tmp* on the floppy disk in device */dev/fdhi*.

```
cptape -o -f /dev/fdhi /usr/tmp
```

CRESTORE CRESTORE

Restore a saved CFS image to the volume(s) it came from. This command is for use by the system administrator only.

NSH Syntax

crestore [*volume...*] [*filename*]

Arguments

volume...	A space-separated list of the volume numbers of the disk drives to which the backup data are to be restored. Only volumes that have been backed up may be included in the list. If you do not list any *volume* numbers, all volumes backed up to that file will be restored.
filename	The name of the file or device that contains the backup. If no file or device is named, *stdin* is read.

Description

NOTE

This command only operates within the **nsh** command.

The **crestore** command lets you restore previously backed-up disk images to the disks from which they were backed up. The *filename* parameter is the name of a file or device (such as */cfs/tape*) from which the backed-up data are to be read. If you do not use the *filename* parameter, *stdin* is read.

If you do not list any *volume* numbers, **crestore** will automatically restore all volumes backed up to the specified file or device. You can only restore data to a volume from which the data have been previously backed up with the **cbackup** command. Your volume list may be a subset of the volumes previously backed up; if so, only the listed volumes will be restored. Use the **showvol** command to get a list of valid *volume* numbers.

It is good practice to issue a **getcube** to reserve an entire cube before performing backup procedures.

CRESTORE *(cont.)*

CRESTORE *(cont.)*

Example

1. To restore from */cfs/tape* the backup previously done of drives 1 and 2, enter the command (notice that since all backed-up data on */cfs/tape* are to be restored in this example, it is not necessary to specify the volume numbers):

```
# crestore /cfs/tape
```

2. To restore backed-up data to volumes 0, 1, and 3 only from a file named *bak.mar1*:

```
# crestore 0 1 3 bak.mar1
```

Errors

`crestore: Bad disk number`

Use **showvol** to find valid volume numbers.

`crestore: Cannot open` *nnn*

File name is invalid. Make sure file name is correct.

`crestore: Disk process not initialized`

Reboot the cube and initialize the disk processes.

`crestore: Cannot open tty`

To avoid conflicting with *stdio*, **crestore** opens another device during the transfer. Make sure the tty device is in */dev*.

`crestore: Invalid backup format`

Error in backup media.

`crestore: Incorrect media label number`

Error in media. Make sure you are restoring to the same drives backed up.

CRESTORE *(cont.)*

CRESTORE *(cont.)*

`crestore: Invalid block number`

Error in backup media.

`crestore: Read error`

Problem with backup media. Check to see that the end of the tape is not overwritten.

See Also

cbackup, showvol

CUBEINFO CUBEINFO

Displays cube ownership information.

Syntax

cubeinfo [**-a** | **-s** | **-n** | **-h** *srmname*]

Arguments

-a	Returns information about all the cubes that you own on the system from which you invoked the command.
-s	Gets information on all the cubes on the system from which the command was executed. If executed on an SRM, it returns information on all cubes allocated from that SRM either directly or via remote workstation. If executed on a remote development machine, it returns information on all cubes that have been allocated from that workstation.
-n	Executed from a remote development system, this option gets information on all cubes allocated on all SRMs to which that workstation has access. It is equivalent to executing **cubeinfo -s** on each SRM on the network. This argument may not be used on the SRM.
-h *srmname*	Executed from a remote development system, returns for the named SRM the same information as **cubeinfo -s**.

Description

With no arguments, the **cubeinfo** command returns information about the current attached cube. The other options provide information about other allocated cubes as described. The cube information returned for each cube is as follows:

CUBENAME	the name given to the cube when it was allocated with **getcube**.
USER	the user id of the owner of the cube.
SRM	the name of the System Resource Manager.
HOST	the name of the system from which **getcube** was invoked.
TYPE	the size and type of cube.

CUBEINFO *(cont.)*

CUBEINFO *(cont.)*

TTYS the terminal from which **getcube** was invoked. A user could have multiple TTY's if an SRM has multiple terminals or if **getcube** was invoked from more than one window on a multi-window workstation. REMOTE in this column means that **getcube** was invoked from a remote development machine, not a System Resource Manager. TTY's are only displayed for attached terminals.

If executed by the superuser, **cubeinfo** also returns the cube ID.

Examples

1. Get information about your current cube (named "defaultname") on an SRM named "yosemite":

```
cubeinfo
CUBENAME        USER        SRM         HOST        TYPE        TTYS
defaultname     don         yosemite    yosemite    4         console
```

2. When user "don" enters the command with the -a option from the remote development machine named "shasta", the command returns information about all cubes that don allocated from shasta:

```
cubeinfo -a
CUBENAME        USER        SRM         HOST        TYPE        TTYS
defaultname     don         klingon     shasta      4           ttyp1
alphabeta       don         klingon     shasta      8
```

3. Entered from the remote development machine "shasta", the -s option returns information about all cubes allocated by anyone from that system:

```
cubeinfo -s
CUBENAME        USER        SRM         HOST        TYPE        TTYS
defaultname     don         klingon     shasta      4           ttyp1
alphabeta       don         klingon     shasta      8           ttyp3
testcube        jan         argon       shasta      1           ttyp8
defaultname     root        salmon      shasta      2m8         ttyp0
```

CUBEINFO *(cont.)*

CUBEINFO *(cont.)*

Errors

`cubeinfo: There is no attached cube`

Use **-a** or **-s** for more information.

`cubeinfo: There are no cubes allocated`

No information was available because no cubes have been allocated.

`cubeinfo: Bootcube in progress`

A **bootcube** is in progress. Try again when the **bootcube** is completed.

`cubeinfo: Commser not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with a **bootcube** or **rebootcube** command.

`cubeinfo: Lifeline not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with a **bootcube** or **rebootcube** command.

See Also

getcube, relcube, attachcube

FSPLIT FSPLIT

Splits a multi-routine Fortran file into individual files.

Syntax

fsplit [*file*]

Description

The **fsplit** command attempts to split Fortran source into separate files, each containing a routine. If no *filename* is specified, standard input is used.The resulting files have names of the form *name*.**f**, where *name* is the name of the program unit (e.g. function, subroutine, block data or program). Unnamed block data subprograms have names of the form **blkdta***NNN*.**f** where *NNN* is three digits and a file of this name does not already exist. For unnamed main programs the name has the form **MAIN.f**. If there is an error in classifying a program unit, or if *name*.**f** already exists, the program unit will be put in a file of the form *zzzNNN*.**f** where *zzzNNN*.**f** does not already exist.

Errors

None

NOTE

The **fsplit** command assumes that the name of the subprogram is on the first non-comment line of the subprogram. A non-standard source format may confuse **fsplit**.

GETCUBE GETCUBE

Allocate a cube and make it the current cube.

Syntax

getcube [**-c** *cubename*] [**-t** *cubetype*] [**-h** *srmname*] [**-a** *account*]

Arguments

-c *cubename* Allows you to allocate multiple cubes by assigning a unique name to each cube as you get it. Valid names are any ASCII character string of 15 characters or less. Without this option, the cube is automatically assigned the name "defaultname."

-t *cubetype* Allows you to specify the size and type of your cube. Specify size first followed by *type* in a contiguous string (no spaces). Without this option, you get the largest available cube. The number of nodes allocated will always be a power of two. If the size you specify is not a power of two, it will be rounded up.

Size can be specified in one of two ways:

n	where *n* is the number of nodes (0 is allowed) or,
d*n*	where *d* indicates dimension and *n* is the size of the dimension (for example: d3).

Specifying no type gets a cube of any type. Other choices are:

rx	i860-based compute nodes
cx	386-based compute nodes
vx	nodes with vector companion board
sx	nodes with SX installed
f	nodes with 387 coprocessor installed
b	disable message buffering
m1	nodes with at least 1M byte of memory each
m4	nodes with at least 4M bytes of memory each

GETCUBE *(cont.)*

GETCUBE *(cont.)*

m8	nodes with at least 8M bytes of memory each
m16	nodes with at least 16M bytes of memory each
m32	nodes with at least 32M bytes of memory each
m64	nodes with at least 64M bytes of memory each
n*s*	gets a continuous set of nodes starting with slot *s*

-h *srmname* Allows you to specify a particular System Resource Manager (SRM) when there are multiple system resource managers on the network. Specify the network name of the SRM. This switch is only useful on a remote development machine.

-a *account* Allows you to specify a account name that is used by your own accounting packages. If this parameter is used, the specified string is passed as an account name to **account_hook()**. Otherwise "defaultaccount" is used.

Description

Use the **getcube** command to allocate a new cube, name it, and make it the current cube. You must allow any **bootcube** command to complete before running the **getcube** command. To run more than one application, allocate additional cubes, assigning each one a unique name using the **-c** parameter. If you specify a name that you (or anyone using your login name) are already using, an error is returned. You are automatically attached to the last cube allocated. To make another cube the current cube, use the **attachcube** command. You can get only one cube at a time. If multiple names are specified, an error is returned. If no cubes can be allocated, an error is returned.

Entered directly on an SRM, **getcube** can only allocate cubes from that SRM.

The **getcube** command also starts a file server to handle I/O operations performed by the nodes in the cube. To redirect the output of the file servers, redirect **getcube** using the UNIX file redirection convention. For example, to redirect the output to a file named "outfile," enter:

```
getcube > outfile
```

To redirect the output of the file server after you execute **getcube**, use the **waitcube** command to pass these changes to the file server. Use **waitcube** to redirect standard input to node programs. For more information on redirection, refer to the shell tutorial in the *UNIX System V User's Guide*.

GETCUBE *(cont.)*

GETCUBE *(cont.)*

The **getcube** command always allocates a number of nodes that is a power of two. If you specify a number of nodes that is not a power of two, **getcube** attempts to allocate additional nodes to make the number of nodes a power of two. If the cube with the next largest dimension is unavailable, no nodes are allocated. **numnodes()** reports the number of nodes actually allocated, not necessarily the number of nodes you requested.

When the memory size is specified in **getcube**, the only guarantee is that nodes containing enough memory to satisfy the request will be allocated. Even when nodes containing the exact amount of memory specified are available, they might not be allocated by **getcube**. For example, if 1M, 4M, 8M, and 16M boards are resident and available, in the cube,

```
getcube -td0m1
```

may allocate either the 1M, 4M, 8M, or 16M node, while

```
getcube -td0m4
```

may allocate either the 4M, 8M, or 16M node.

All of the parameters are optional; defaults are assigned for any parameters omitted. Entering the **getcube** command with no arguments gets a cube composed of the largest unallocated number of nodes (the number is always a power of two) and assigns it the name "defaultname."

Cube commands (such as **load**, **killcube**, etc.) use *cubename* as their target. If you need to find out the names of the cubes that you have allocated, use the **cubeinfo** command.

Use the **-t** option to specify the size and the type of cube you want. Cubes are always allocated such that the number of nodes is a power of two. Thus, in addition to the higher powers of two, you may allocate a cube composed of 0 or 1 node. You may also specify the minimum required memory amount, so that the nodes allocated will have at least the required memory, but may have more.

For example, **-t 2** specifies a 2-node cube in which the nodes can be of any type, **-t 16rx** specifies a 16-node cube in which all of the nodes are i860-based, and **-t d3m4** specifies an 8-node cube in which each node has at least 4M bytes of memory per node.

To get a cube composed of different types of nodes, specify the different sizes and types, separated by "/" in one string. For example, **4vx/4m8** gets four vector nodes and four nodes with at least 8M bytes of memory each.

The **getcube** command reports whether or not it is successful, and the size and type of the cube allocated.

GETCUBE *(cont.)*

GETCUBE *(cont.)*

The **getcube** command logs accounting information with a default name string of "defaultaccount" or with the *account* name string that you specify with the **-a** flag. The accounting information includes the event type (startup, shutdown, getcube, or relcube operations), and a structure with specific cube information (including the *account* string).

The **getcube** command calls the routine **account_hook**() before logging the accounting information. Exactly what **account_hook**() does with the accounting information is specific to your installation. The default routine simply prints a log of user activity in */usr/ipsc/commser.log*.

The source code for **account_hook**() is provided. Modifying **account_hook**() is superuser activity that should only be performed by the system administrator. The system administrator may modify the **account_hook**() routine to perform other actions that control and monitor node usage. For example, the **account_hook**() routine could be modified to limit node access during certain periods of the day, or deny allocation to users with insufficient privilege.

All the cubes that you own are automatically released when you log out from the SRM unless you specify otherwise by invoking **getcube** with the UNIX command **nohup**, as, for example:

```
nohup getcube &
```

Remember that if you log out of a remote host, cubes are not automatically released.

Examples

1. Allocate first available cube of any type and size; without the **-c** option, the cube will be assigned the name "defaultname":

   ```
   getcube
   ```

2. Allocate a 4-node, vector cube named "alpha":

   ```
   getcube -c alpha -t 4vx
   ```

3. Allocate first available cube of any type and size with the default name on a System Resource Manager named "hal":

   ```
   getcube -h hal
   ```

4. Allocate a 2-node cube with the account name "cs 130". The "cs 130" account name will be passed to the **account_hook**() routine when the **getcube** command executes:

   ```
   getcube -t2 -a "cs 130"
   ```

GETCUBE *(cont.)*

GETCUBE *(cont.)*

Errors

`getcube: Internal cube usage limit`

Limit of 10 cubes may be allocated. Try again later when cube is not as busy.

`getcube: Commser not responding`

`getcube: Lifeline not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with a **bootcube** or **rebootcube** command.

`getcube: Bootcube in progress`

A **bootcube** or **rebootcube** command is currently in progress. The system does not allow any cubes to be allocated until the **bootcube** or **rebootcube** command is complete.

`getcube: Cubename already exists`

Use a different name.

`getcube: Cubetype not available`

The cubetype is present, but currently in use.

`getcube: Cubetype not installed`

The cubetype is not present.

`getcube: Cubetype not valid`

The request is invalid.

`getcube: No SRM that matched your request was found`

Remote development machine error only. Could not find an SRM with a cube of the type requested.

GETCUBE *(cont.)*

GETCUBE *(cont.)*

`getcube: Commser out of buffers`

Communication server out of buffers.

`getcube: permission denied`

If the accounting routines reject a user, this message is given.

See Also

attachcube, cubeinfo, newserver, relcube

HEAD HEAD

Display the first few lines of specified files.

Syntax

head [*-n*] [*filename...*]

Arguments

-n	number of lines to display in specified file.
filename	name of specified file.

Description

The **head** command copies the first *n* lines of each *filename* to the standard output. If no *filename* is given, **head** copies lines from the standard input. The default value of *n* is 10 lines.

A common way to display a set of short files, identifying each one, is:

```
head -9999 filename1 filename2 ...
```

Examples

Entering ***head -4 /etc/{hosts,motd}*** produces:

```
127.0.0.1       localhost
128.0.0.1       dorothy
128.0.0.2       toto
128.0.0.3       tinman

################################################################
Finished printing /etc/hosts
################################################################
   ****************************************
   *                                      *
   *     Welcome to dorothy               *
   *                                      *
################################################################
Finished printing /etc/motd
################################################################
```

KILLCUBE KILLCUBE

Kill node process(es).

SRM Syntax

killcube [**-c** *cubename*] [**-p** *pid*] [*node...*]

NSH Syntax

killcube -p *pid* [*node...*]

Arguments

-c *cubename*	The name of the cube on which you wish to kill a process. Without this argument, the current attached cube is used. If you specify a *cubename* that does not exist, an error is returned.
-p *pid*	The *pid* of the process to be killed. If no *pid* is given, all processes are killed. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on the RX node).
node...	A node or list of nodes on which the process is to be killed. If you do not specify a node, all nodes are killed.

Description

The **killcube** command kills the specified processes and flushes messages related to those processes. It is not an error to use **killcube** to kill a nonexistent process. It is recommended that you use the **killcube** command to kill any remaining cube processes before you release the cube (with the **relcube** command).

Examples

1. Kill all processes on all nodes in the cube named "alpha":

```
killcube -c alpha
```

2. Kill process 1 on the first three nodes in the current cube:

```
killcube -p 1 0 1 2
```

KILLCUBE *(cont.)*

KILLCUBE *(cont.)*

Errors

`killcube: Invalid pid`

The *pid* must not be negative. For RX nodes, *pid* must be 0.

`killcube: Invalid node`

Node number must be in range of cube size.

`killcube: Bootcube in progress`

A **bootcube** is in progress. Try again when the **bootcube** is completed.

`killcube: There is no attached cube`

Use **getcube** to get a cube.

`killcube: Cubetype does not exist`

Use **cubeinfo** to find names of available cubes.

See Also

relcube, load

LESS LESS

Similar to **more**, but with forward and backward movement and additional features.

Syntax

less [**-dstwcCeEmMqQuU**] [**-h** *N*] [**-b**[**fp**] *N*] [**-x** *N*] [**-**[**z**] *N*] [**-P**[**mM**] *string*] [**-**[**lL**] *logfile*] [+ *cmd*] [*filename...*]

Description

The **less** command is a program similar to **more**, but which allows backwards movement in the file as well as forward movement. Also, **less** does not have to read the entire input file before starting, so with large input files it starts up faster than text editors like **vi**. The **less** command uses *termcap*, so it can run on a variety of terminals. There is even limited support for hardcopy terminals. (On a hardcopy terminal, lines which should be printed at the top of the screen are prefixed with an up-arrow.)

Commands are based on both **more** and **vi**. Commands may be preceded by a decimal number, called *N* in the descriptions above. The number is used by some commands, as indicated.

Commands

H	Help: display a summary of these commands. If you forget all the other commands, remember this one.
<SPACE>	Scroll forward *N* lines, default one window (see option **-z** below). If *N* is more than the screen size, only the final screenful is displayed.
f	Same as **<SPACE>**. You can also use **<Ctrl-F>** to scroll forward.
b	Scroll backward *N* lines, default one window (see option **-z** below). If *N* is more than the screen size, only the final screenful is displayed. You can also use **<Ctrl-B>** to scroll backward.
<RETURN>	Scroll forward *N* lines, default **1**. The entire *N*-lines are displayed, even if *N* is more than the screen size.
e or **j**	Same as **<RETURN>**. You can also use **<Ctrl-E>** or **<Ctrl-J>** to scroll forward.

LESS *(cont.)*

LESS *(cont.)*

y or **k**	Scroll backward *N* lines, default 1. The entire *N*-lines are displayed, even if *N* is more than the screen size. You can also use **<Ctrl-Y>** or **<Ctrl-K>** to scroll backward.
d	Scroll forward *N* lines, default 10 or the last *N* lines that were applied to a **d** or **u** command. You can also use **<Ctrl-D>** to scroll forward.
u	Scroll backward *N* lines, default 10 or the last *N* lines that were applied to a **d** or **u** command. You can also use **<Ctrl-U>** to scroll backward.
r	Repaint the screen. You can also use **<Ctrl-R>** to repaint the screen.
R	Repaint the screen, discarding any buffered input.
/pattern	Search forward in the file for the *N*-th line containing the pattern. *N* defaults to 1. The pattern is a regular expression, as recognized by **ed**. The search starts at the second line displayed (but see the **-t** option, which changes this).
?*pattern*	Search backward in the file for the *N*-th line containing the pattern. The search starts at the line immediately before the top line displayed.
n	Repeat the previous forward or backward search for the *N*-th line containing the pattern.
g	Go to line number *N*, default of line 1.
G	Same as **g**, but the default is the last line in the file.
p or **%**	Go to *N* percent into the file (100% is the end of file).
m*letter*	Mark the current cursor position with *letter*.
'*letter*	Go to the position in the file marked by *letter*.
''	Return to the previous position in the file.
E [*filename*]	Examine a new file. If you do not include *filename*, you will be prompted for the new file.
N	Examine the next file (from the list of files given in the command line). If a number *N* is specified (not to be confused with the command **N**), the *N*-th next file is examined.
P	Examine the previous file (from the list of files given in the command line). If a number *N* is specified, the *N*-th previous file is examined.

LESS *(cont.)* LESS *(cont.)*

=	Print the current file name.
V	Prints the version number of **less** being run.
q	Exits **less**.

The following two commands may or may not be valid, depending on your particular installation.

v	Invokes an editor to edit the current file being viewed. The editor is taken from the environment variable *EDITOR*, or defaults to **vi**.
!	Invokes a shell to run the shell-command given. A percent sign in the command is replaced by the name of the current file. "!!" repeats the last shell command.

Options

Command line options are described below. Most options may be changed while **less** is running, via the "-" command.

Options are also taken from the environment variable *LESS*. For example, if you like **more**-style prompting, to avoid typing **less -m...** each time **less** is invoked, you might tell **csh**:

```
setenv LESS m
```

or if you use **sh**:

```
LESS=m; export LESS
```

The environment variable is parsed before the command line, so command line options override the *LESS* environment variable. A dollar sign ($) may be used to signal the end of an option string. This is important only for options like **-P**, which take a following string.

-b [f | p] *n*	The **-b***n* option tells **less** to use a non-standard buffer size. There are two standard (default) buffer sizes, one is used when a file is being read and the other when a pipe (standard input) is being read. The current defaults are 5 buffers for files and 12 buffers for pipes. (Buffers are 1024 bytes.) The number *n* specifies a different number of buffers to use. The **-b** may be followed by **f**, in which case only the file default is changed, or by **p** in which case only the pipe default is changed. Otherwise, both are changed.
-c	Normally, **less** will repaint the screen by scrolling from the bottom of the screen. If the **-c** option is set, when **less** needs to change the entire display, it will paint from the top line down.

LESS *(cont.)*

LESS *(cont.)*

-C The **-C** option is like **-c,** but the screen is cleared before it is repainted.

-d Normally, **less** will complain if the terminal is dumb; that is, it lacks some important capability, such as the ability to clear the screen or scroll backwards. The **-d** option suppresses this complaint (but does not otherwise change the behavior of the program on a dumb terminal).

-e Normally the only way to exit **less** is via the **q** command The **-e** option tells **less** to automatically exit the second time it reaches end-of-file.

-E The **-E** flag causes less to exit the first time it reaches end-of-file.

-h Normally, **less** will scroll backwards when backwards movement is necessary. The **-h** option specifies a maximum number of lines to scroll backwards. If it is necessary to move backwards more than this many lines, the screen is repainted in a forward direction. (If the terminal does not have the ability to scroll backwards, **-h0** is implied.)

-l The **-l** option, followed immediately by a *filename,* will cause **less** to copy its input to the named file as it is being viewed. This applies only when the input file is a pipe, not an ordinary file. If the file already exists, **less** will ask for confirmation before overwriting it.

-L The **-L** option is like **-l,** but it will overwrite an existing file without asking for confirmation.

-m Normally, **less** prompts with a colon. The **-m** option causes **less** to prompt verbosely (like **more**), with the percent into the file.

-M The **-M** option causes **less** to prompt even more verbosely than **more.**

LESS *(cont.)*

LESS *(cont.)*

-P The **-P** option provides a way to tailor the three prompt styles to your own preference. You would normally put this option in your *LESS* environment variable, rather than type it in with each **less** command. Such an option must either be the last option in the *LESS* variable, or be terminated by a dollar sign. A **-P** followed by a *string* changes the default (short) prompt to that *string*. **-Pm** changes the medium (**-m**) prompt to the *string*, and **-PM** changes the long (**-M**) prompt. The *string* consists of a sequence of letters which are replaced with certain predefined strings, as follows:

`F`	file name
`f`	file name, only once
`o`	file *n* of *n*
`o`	file *n* of *n*, only once
`b`	byte offset
`p`	percent into file
`P`	percent if known, else byte offset. Angle brackets, "<" and ">," may be used to surround a literal string to be included in the prompt. The defaults are `fo` for the short prompt, `foP` for the medium prompt, and `Fobp` for the long prompt. Example: Setting your *LESS* variable to `PmFOP$PMFObp` would change the medium and long prompts to always include the file name and "file *n* of *n*" message. Another example: Setting your *LESS* variable to `mPmFoPe` would change the medium prompt to the string "--Less--" followed by the file name and percent into the file. It also selects the medium prompt as the default prompt (because of the first **m**).

-q Normally, if an attempt is made to scroll past the end of the file or before the beginning of the file, the terminal bell is rung to indicate this fact. This option tells **less** not to ring the bell at such times. If the terminal has a *visual bell*, it is used instead.

-Q Even if **-q** is given, **less** will ring the bell on certain other errors, such as typing an invalid character. The **-Q** option tells **less** to be quiet all the time; that is, never ring the terminal bell. If the terminal has a *visual bell*, it is used instead.

LESS *(cont.)*

LESS *(cont.)*

-s The **-s** option causes consecutive blank lines to be squeezed into a single blank line. This is useful when viewing **nroff** output.

-t Normally, forward searches start just after the top displayed line (that is, at the second displayed line). Thus forward searches include the currently displayed screen. The **-t** option causes forward searches to start just after the bottom line displayed, thus skipping the currently displayed screen.

-u If the **-u** option is given, backspaces are treated as printable characters; that is, they are sent to the terminal when they appear in the input.

-U If the **-U** option is given, backspaces are printed as the two character sequence "^H." If neither **-u** nor **-U** is given, backspaces which appear adjacent to an underscore character are treated specially: the underlined text is displayed using the terminal's hardware underlining capability. Also, backspaces which appear between two identical characters are treated specially; the overstruck text is printed using the terminal's hardware boldface capability. Other backspaces are deleted, along with the preceding character.

-w Normally, **less** uses a tilde character to represent lines past the end of the file. The **-w** option causes blank lines to be used instead.

-x The **-x***n* option sets tab stops every *n* positions. The default for *n* is **8**.

-[z]*n* When given a backwards or forwards window command, **less** will by default scroll backwards or forwards one screenful of lines. The **-z***n* option changes the default scrolling window size to *n* lines. If *n* is greater than the screen size, the scrolling window size will be set to one screenful. Note that the **z** is optional for compatibility with **more**.

+ If a command line option begins with +, the remainder of that option is taken to be an initial command to **less**. For example, **+G** tells **less** to start at the end of the file rather than the beginning, and **+/xyz** tells it to start at the first occurrence of `xyz` in the file. As a special case, **+#** acts like **+#g**; that is, it starts the display at the specified line number (however, see the caveat under the **g** command above). If the option starts with ++, the initial command applies to every file being viewed, not just the first one. The + command described previously may also be used to set (or change) an initial command for every file.

See Also

UNIX **more** command

LOAD LOAD

Loads a user process into the cube.

SRM Syntax

load [**-c** *cubename*] [**-p** *pid*] [**-H**] [*node...*] *filename* [*arguments...*]

NSH Syntax

load -p *pid* [*node...*] *filename* [*arguments...*]

Arguments

-c *cubename*	The name of the cube that you wish to load. Without this argument, the current attached cube is used. If the cube does not exist, an error is returned.
-p *pid*	The process ID (*pid*) assigned to this process. If you do not specify a *pid*, the process ID is zero by default. To load multiple processes on a single node, you must assign each a unique *pid*. Valid *pids* for CX nodes include any non-negative integer value. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on the RX node).
-H	Do not start the process immediately. The process will not start until you execute the **startcube** command.
node...	List of the nodes into which the process is to be loaded. Separate multiple node numbers with a space. If no nodes are specified, all nodes are loaded.
filename	The file name (include the pathname if the file is not in the current directory) of the process that you wish to load. The first character of *filename* must not be a numeric digit.
arguments...	Any arguments required by the file to be loaded.

LOAD *(cont.)*

LOAD *(cont.)*

Description

The **load** command loads the executable process designated by *filename* onto the node(s). To load more than one process, invoke the **load** command for each process separately (command arguments do not need to be the same for these separate loads). Assign each a unique process ID (*pid*). For RX nodes, do not use the **-p** option unless you set *pid* to 0.

In a RX node, when a node program executes a **fork**(), a new process is created on any available node. An available node is a node that does not have a process running on it. If a process has been loaded with **load** and has terminated, then that node is unavailable until it has been waited on with the **waitcube** command, **waitall**(), or **waitone**().

For example, assume that you have allocated a two-node cube. You load a program on both nodes and they execute correctly and terminate. Then you load another program on node 0 which tries to **fork**() another process . However, node 1 is not available until it has been waited on. Hence, the **fork**() fails with either of two error messages:

```
No more processes

Out of process slots
```

Examples

1. Load a file named *proc3* into all the nodes in the current cube, assigning it process ID 3

   ```
   load -p 3 /usr/tom/proc3
   ```

2. Load a file named *node_pgm* (which requires two arguments, here shown as "input_file" and "1000") into nodes 3 and 4 of a cube named "alpha":

   ```
   load -c alpha 3 4 node_pgm input_file 1000
   ```

Errors

`load: Invpid`

The *pid* must not be negative. For RX nodes, *pid* must be 0.

`load: Invalid node`

Node number must be in range of cube size.

LOAD *(cont.)* LOAD *(cont.)*

`load: Out of process slots`

Use fewer processes. Up to 20 processes per node are allowed.

`load: Pid already in use`

Use a different *pid.*

`load: Not enough memory`

Reduce the size of the process or get a different cube.

`load: Bootcube in progress`

A **bootcube** is in progress. Try again when the **bootcube** is completed.

`load: Invalid loader record`

Internal error, try again. You may have to use a **bootcube** or **rebootcube** command.

`load: Invalid loader message`

Internal error, try again. You may have to use a **bootcube** or **rebootcube** command.

`load: No such file or directory`

Correct the *filename.*

`load: Invalid object file`

Specify a loadable file.

`load: There is no attached cube`

Use **getcube** to allocate a cube and, if necessary, the **attachcube** command to change cubes within your program.

See Also

attachcube, killcube, getcube, relcube, startcube

LSIZE LSIZE

Increase the size of a file.

Syntax

lsize [**-a**] *size*[**k**|**m**|**g**] *file* ...

Arguments

-a	append to the file. Otherwise *size* represents the final size of the file.
size	the size of the file. Without a suffix, *size* is measured in bytes.
k|**m**|**g**	**k** means that *size* is measured in kilobytes; **m** means *size* is measured in megabytes, and **g** means *size* is measured in gigabytes.

Description

NOTE

This command operates only under the node shell.

The **lsize** command runs under the node shell and increases the size of an existing file. You can specify the final size of the file. With the **-a** option you can specify the amount you want appended to the file. The extra bytes added to the file contain whatever happened to be on the disk at the time.

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

Examples

The following example creates a CFS file called *myfile*. Then it executes **lsize** to increase the size of the file to 100 bytes.

```
nsh- cat >myfile
hello everyone
                                                    Enter a <Ctrl-D>
nsh- ls -l myfile
-rw-r--r--   1 you      other         15  Mar  5 11:40 myfile
nsh- lsize 100 myfile
nsh- ls -l myfile
-rw-r--r--   1 you      other         100  Mar  5 11:40 myfile
```

The next example increases the size of *myfile* by 1K bytes.

```
nsh- lsize -a 1k myfile
nsh- ls -l myfile
-rw-r--r--   1 you      other         1124  Mar  5 11:40 myfile
nsh-
```

Errors

`lsize: No space left on device`

Not enough space on the device to which you are writing. Create more space in the file system, or if you have restricted the disks to which you are writing with **restrictvol()**, re-execute **restrictvol()** to include more drives.

`lsize: Not enough core`

Not enough memory on node for operation.

MAKEWHATIS MAKEWHATIS

Create *whatis* files for **man** directories. This command describes the **man** implementation on the SRM only

Syntax

makewhatis *directory ...*

Arguments

directory	The name of the directory containing the *man?* directories. This directory is the one that will contain the *whatis* file

Description

makewhatis collects summary information for all the online manual pages in a manual page directory set. The **apropos** and **whatis** commands use this summary information.

A set of manual page directories consists of a parent directory, typically called *man*, and a set of subdirectories of that directory called *man1*, *man2*, through *man8*. **makewhatis** places all the summary information in a file named *whatis* in the parent directory for each set of manual page directories. For example, if the manual pages are *man/man1* through *man/man8*, then *whatis* resides in *man*.

You must execute **makewhatis** from the parent of the directory that contains the *man?* directories. For example, if the manual pages are *man/man1* through *man/man8*, then you must execute **makewhatis** from *man*'s parent. **makewhatis** resides in *$IPSC_XDEV/i860/lib/man*.

Examples

For example, assume that you have installed the ProSolver™ SES manual pages. They now reside in *$IPSC_XDEV/i860/man/man3*. To make a new *whatis* file, make *$IPSC_XDEV* your current directory and execute **makewhatis**, specifying *man*.

```
% cd $IPSC_XDEV/i860
% makewhatis man
```

See Also

man, manpath, apropos, whatis

MAN MAN

Format and display the online manual pages. This command describes the **man** implementation on the SRM only.

Syntax

man [**-a**] [**-M** *path*] [**-P** *pager*] [**-S** *system*] [*section*] *name* . . .[[*section*] *name* ...] ...
man -f [**-M** *path*] [**-P** *pager*] *name* . . .
man -k [**-M** *path*] [**-P** *pager*] *keyword* . . .
man -h

Arguments

-a	Forces **man** to display all the manual pages that match *name*, not just the first it finds. **man** searches for commands and then for calls. It searches for C calls and then Fortran calls. By default, **man** exits after displaying the first manual page it finds.
-f	Equivalent to **whatis**.
-h	Prints a one line help message and exits.
-k	Equivalent to **apropos**.
-M *path*	Specifies an alternate directory to search for man page files. If there is no *MANPATH* environment variable, **man** uses the *manpath.config* file. This file resides in *$IPSC_XDEV/i860/lib/man*. If the environment variable *MANPATH* exists man uses its value to determine the directories to search. This option overrides the *manpath.config* file and the *MANPATH* environment variable.
-P *pager*	Specifies which pager to use. By default, man uses **less**, which has the ability to back up on files and pipes, so you can go backwards while reading the manual. This option overrides the *PAGER* environment variable.
-S *system*	Adds *system* to the searched directories under the directory *man*. That is, in addition to searching *man/man1* through *man/man8*, the **man** command also searches *man/system/man1* through *man/system/man8*.
section	Specifies the section number that the manual page appears in.

MAN *(cont.)* MAN *(cont.)*

name Specifies the command or system call whose manual page you want.

keyword Specifies a word that appears in the command or call description in the NAME section.

Description

man formats and displays the online manual pages. If *section* is specified, **man** only looks in that section of the manual. This version knows about the *MANPATH* and *PAGER* environment variables.

MANPATH If *MANPATH* is set, **man** uses its value as the path to search for manual pages. If *MANPATH* is not set, **man** uses the file *manpath.config* and the execution search path.

PAGER If *PAGER* is set, **man** uses its value as the name of the program to use to display the **man** page. By default, **less** is used.

When the environment variable *MANPATH* is not set, **man** uses the file *manpath.config*. This file resides in *$IPSC_XDEV/i860/lib/man* and contains two types of **man** path information. The first is called **MANDATORY_MANPATH**, and the second is called **MANPATH_MAP**.

Each **MANDATORY_MANPATH** line takes one directory name. These directories are searched for directories containing man page files (for example, *man1*). For example, if *manpath.config* contains the lines

```
MANDATORY_MANPATH       /usr/man
MANDATORY_MANPATH       /usr/local/man
```

then **man** searches both */usr/man* and */usr/local/man*.

MAN *(cont.)* MAN *(cont.)*

Each **MANPATH_MAP** line takes two directories: the *path_element* and the *manpath_element*. If the *path_element* exists in your execution search path, then the *manpath_element* will be in your **man** search list. For example, if *manpath.config* contains

```
MANPATH_MAP     /usr/local/bin/X11      /usr/local/X11/man
```

/usr/local/bin/X11 is the *path_element*, and */usr/local/X11/man* is the *manpath_element*. If */usr/local/bin/X11* is in your execution search path, then */usr/local/X11/man* is in **man**'s search list.

Finally, **man** checks your execution search path and includes subdirectories *man* or *MAN* that are underneath execution search path directories. For example, if your execution search path contains */usr/ipsc/XDEV/i860*, then **man** searches */usr/ipsc/XDEV/i860/man*. If both *man* and *MAN* subdirectories exist in the same search directory, only the *man* path will be added.

Examples

The following example displays the online manual page for the C version of the **crecv()** call.

```
% man crecv
CRECV(3)      iPSC(Reg.)/860 System      CRECV(3)

NAME
     CRECV - Receive a message and wait for the
     receive to complete before proceeding.

SYNOPSIS
     crecv(typesel, buf, len)

PARAMETER DECLARATIONS
     long typesel;
     char *buf;
     long len;

RETURN VALUE
     None
          .
          .
          .
```

MAN *(cont.)* MAN *(cont.)*

The next example displays the Fortran version of the system call **crecv()**. If you do not specify the *section*, you get the C version. **man -a crecv** gives you both. The section for the Fortran versions is **3f**.

```
% man 3f crecv
CRECV(3f)      iPSC(Reg.)/860 System      CRECV(3f)

NAME
     CRECV - Receive a message and wait for the receive to
     complete before proceeding.

SYNOPSIS
     SUBROUTINE CRECV(typesel, buf, len)

PARAMETER DECLARATIONS
     INTEGER typesel
     INTEGER buf(*)
     INTEGER len

RETURN VALUE
     None
          .
          .
          .
```

See Also

apropos, whatis, manpath

MANPATH MANPATH

Determine user's search path for manual pages. This command describes the **man** implementation on the SRM only.

Syntax

manpath [-q]
manpath -h

Arguments

-q	Operate quietly. Only echo the final **man** path.
-h	Displays a help line.

Description

manpath tries to determine the **man** search path from a set of system defaults (as defined in *manpath.config*), the execution search path and the user's environment, echoing the result to the standard output. Warnings and errors are written to the standard error. The result of **manpath** is used by **man** to determine its search path.

If the *MANPATH* environment variable is set, **manpath** returns the directories listed in that variable. **manpath** echoes its value on the standard output and issues a warning on the standard error.

When the environment variable *MANPATH* is not set, **man** uses the file *manpath.config*. This file resides in *$IPSC_XDEV/i860/lib/man* and contains two types of **man** path information. The first is called **MANDATORY_MANPATH**, and the second is called **MANPATH_MAP**.

Each **MANDATORY_MANPATH** line takes one directory name. These directories are searched for directories containing man page files (for example, *man1*). For example, if *manpath.config* contains the lines

```
MANDATORY_MANPATH       /usr/man
MANDATORY_MANPATH       /usr/local/man
```

then **man** searches both */usr/man* and */usr/local/man*.

Each **MANPATH_MAP** line takes two directories: the *path_element* and the *manpath_element*. If the *path_element* exists in your execution search path, then the *manpath_element* will be in your **man** search list. For example, if *manpath.config* contains

```
MANPATH_MAP     /usr/local/bin/X11      /usr/local/X11/man
```

MANPATH *(cont.)*

MANPATH *(cont.)*

/usr/local/bin/X11 is the *path_element*, and */usr/local/X11/man* is the *manpath_element*. If */usr/local/bin/X11* is in your execution search path, then */usr/local/X11/man* is in **man**'s search list.

Finally, **man** checks your execution search path and includes subdirectories *man* or *MAN* that are underneath execution search path directories. For example, if your execution search path contains */usr/ipsc/XDEV/i860*, then **man** searches */usr/ipsc/XDEV/i860/man*. If both *man* and *MAN* subdirectories exist in the same search directory, only the *man* path will be added.

Example

Echo a typical **man** path when *MANPATH* is set.

```
% setenv MANPATH /usr/local/man:$IPSC_XDEV/i860/man
% manpath
(Warning: MANPATH environment variable set)
/usr/local/man:/usr/ipsc/XDEV/i860/man
%
```

See Also

apropos, whatis, man

MKCFS MKCFS

Construct a file system. Intended for use by system administrator only.

CAUTION

This command is to be used by your system administrator only. If your file system has existing files on it, they will be destroyed in creating a new concurrent file system (CFS).

Syntax

mkcfs

Arguments

None

Description

The **mkcfs** command constructs a new concurrent file system, or allows you to add new drives to the current system. Before running **mkcfs**, execute the commands:

```
# bootcube -D22
```

The **bootcube** command with the **-D22** option is required to initialize the disk software without initializing the file system. The size of the cube you get with **getcube** is not important because **mkcfs** requires only the cube software, not the nodes themselves.

The **mkcfs** command has no options; instead, it is interactive, giving you several opportunities to discontinue the process so you do not accidently replace an existing CFS with a new one. After running **mkcfs**, perform a **bootcube** command. The command creates a volume label for each drive in the system. You define the file system name when you execute **mkcfs**. The volume label identifier and version are predefined.

MKCFS *(cont.)*

MKCFS *(cont.)*

If there is no existing file system, it prompts you for a name for the file system, and then builds the CFS. To see a list of available drives, execute the **showvol** command. If there is an existing file system, entering the command causes this response:

n drives ready on *m* nodes File system *name* already exists. Continue?

You can respond either with a ***y*** or an ***n***. If your response is negative (***n***), the CFS remains unchanged. If your response is ***y***, it will ask you if you want to change the file system name. If you do, it will prompt you for a new name and create a new CFS, assigning volume numbers to all drives. File system names may have spaces within the name, and a maximum of 30 characters.

If you do not want to change the name, but only want to add new drives to the file system, it will detect the new drives, and ask you if you need labels for the new drives. If you respond with ***y*** (yes), it will supply all new drives with labels, either all at once, or asking you for each new drive individually.

Examples

Add 8 new drives to an existing CFS with 32 existing drives.

```
# mkcfs
8 drives ready on 4 nodes
File system "old cfs" already exists. Continue? y
Do you wish to change the file system name? y
Enter new file system name: new cfs
Are you sure you want to destroy all files? y
#
```

Errors

`mkcfs: Disk process not initialized`

Reboot the cube and initialize the disk processes.

`mkcfs: Invalid node` *nnn* `for volume` *nnn*

Reboot the cube and reinitialize the disk processes.

`mkcfs: Bootcube in progress`

A **bootcube** is in progress. Try again when the **bootcube** is completed.

MKCFS *(cont.)*

MKCFS *(cont.)*

See Also

bootcube, getcube, showvol

MKDEV MKDEV

Allocates a file that represents a tape managed by CFS. This command requires root access and is intended for use by the system administrator only.

NSH Syntax

mkdev *volume devicename*

Arguments

volume	the volume number for the device. It is the same volume number returned by the **showvol** command.
devicename	the name for the file representing the device. It must be a complete path and filename.

Description

NOTE

This command only operates within the **nsh** command.

The **mkdev** command allocates a file which represents a tape drive managed by CFS. Referencing this file will access the device. This file resides only in CFS. Use the **showvol** command to find the volume number of your tape devices.

MKDEV *(cont.)*

MKDEV *(cont.)*

Examples

The following example creates a CFS file which represents the device whose volume number is 3. This file would have to be created in order to access a tape device.

```
mkdev 3 /cfs/mt0
```

Errors

`Usage: mkdev volume device`

Either the *volume* or *devicename* arguments are missing.

`mkdev: volume number: Invalid argument`

The *volume* argument is not a valid volume number.

`mkdev: Pathname must be within CFS`

The *devicename* argument must be within the CFS.

See Also

nsh, tapemode

MT

MT

A magnetic tape manipulating program.

NSH Syntax

mt [**-f** *tapename*] *command* [*count*]

Arguments

tapename the name of a CFS tape device. This must not be the SRM tape device.

command one of the following:

bsf	Space backward *count* files.
bsr	Space backward *count* records.
erase	Erase the whole tape.
fsf	Space forward *count* files.
fsr	Space forward *count* records.
reten	Retension the tape.
rew	Rewind the tape (ignore *count*).
stat	Print status information about the tape unit.
weof	Write *count* end-of-file marks at the current position on the tape.

count the number of times to perform the operation.

MT *(cont.)* MT *(cont.)*

Description

NOTE

This command only operates within the **nsh** command.

The **mt** command issues commands to a magnetic tape drive. If a tape name is not specified, **mt** uses the raw magnetic tape interface device at */cfs/tape*. By default, **mt** performs the requested operation once. Specify *count* to determine how many times you wish to perform the operation.

Examples

The following example erases the whole tape mounted on */cfs/backup*:

```
mt -f /cfs/backup  erase
```

Errors

`mt: I/O error`

Be sure there is a tape mounted on the drive.

See Also

UNIX System V Reference Manual: **mtio(), dd(), ioctl()**

NEWSERVER NEWSERVER

Start a new file server for the specified cube.

Syntax

newserver [**-c** *cubename*]

Arguments

-c *cubename*	Allows you to name the cube to which the new file server is to be attached. Without this argument, the current attached cube is used. If the specified cube does not exist, an error is returned. You can specify only one cube at a time; specifying more than one name returns an error.

Description

When you allocate a cube with **getcube**, a file server to handle I/O for the nodes is automatically created. If, after executing the **getcube** command you wish to redirect the cube output, invoke **newserver**. The **newserver** command actually kills the original file server and starts a new one. When you invoke **newserver**, changes that you make to your environment are passed to the new file server.

If you want to redirect the standard output or standard error of the file servers, redirect **newserver**. For example, to redirect the standard output to a file named *outfile*, enter:

```
newserver > outfile
```

For information on redirection, refer to the shell tutorial in the *UNIX System V User's Guide*.

One file server is started for each 32 nodes allocated, that is, if you have a **D6** cube, you will have two file servers.

NEWSERVER *(cont.)* NEWSERVER *(cont.)*

Examples

1. Start a new file server for a cube named "alpha" and redirect output to *outfile*:

```
newserver -c alpha > outfile
```

2. To redirect input to a node program, use the **waitcube** command:

```
waitcube < node_data
```

Errors

`newserver: Cubename does not exist`

Use **cubeinfo** to find out cube names.

`newserver: Commser not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with a **bootcube** or **rebootcube** command.

`newserver: Lifeline not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with bootcube or rebootcube.

`newserver: Bootcube in progress`

A **bootcube** is in progress. Try again when the **bootcube** is completed.

See Also

getcube, syslog

NSH NSH

Load the C shell onto node 0.

Syntax

nsh [**-s**] [*arguments*]

Arguments

-s	**nsh** loads on the user's currently attached cube. When using RX nodes, this attached cube must have at least four nodes. Without **-s**, **nsh** loads the node shell on a service node. A service node is an I/O node without any disks.
arguments	Arguments valid for the C shell are valid for the shell loaded on the node.

Description

The **nsh** command loads a C shell onto a service node of your cube. Normal C shell command-line arguments are valid. The shell process on node 0 then manages the distribution of processes created under the C shell to the other nodes of the cube. The node's C shell has access to both the host file system and the CFS, so it is possible to copy a file from one to the other from within the shell. To return to the UNIX shell from the node's C shell, enter ***exit***.

nsh uses three special files:

.login.ipsc
.cshrc.ipsc
.logout.ipsc

You should create these files in your home directory. These files are used in the same way that UNIX uses the *.login*, *.cshrc*, and *.logout* files. In the *.cshrc.ipsc* file, set the shell and path variables as follows:

```
set shell = /usr/ipsc/bin/csh
set path = /usr/ipsc/bin
```

You may also add other elements to the path, but the added directories must contain only node executables, not UNIX executables.

If you invoke **nsh** with no arguments, it acts as a login shell and reads *.login.ipsc*, normally invoked with the **-l** command line argument (described in C-shell documentation). If you use other arguments, you must also include the **-l** switch for the shell to read *.login.ipsc*.

NSH *(cont.)*

NSH *(cont.)*

Appendix A shows the **csh** built-in commands, the UNIX-compatible utilities, and the iPSC commands supported by the node's C shell. The built-in commands and the UNIX-compatible utilities operate the same as those running on the host. The node's C shell can also run any node program intended to be executed by a single node.

For CX nodes, the **killcube**, **startcube**, and **waitcube** commands running on the node must have the **-p** switch and do not accept the **-c** switch because they cannot communicate with other allocated cubes. Specifying a *pid* with the **-p** switch prevents you from affecting the node's C-shell itself. The node's C-shell *pid* is 1999999999.

For RX nodes, the **killcube**, **startcube**, and **waitcube** commands do not use the **-c** options because they cannot communicate with other allocated cubes. Also, the minumum subcube size is 4 nodes.

Examples

Load the node's C shell ontoa service node. The display assumes that you have set the prompt in *.login.ipsc* to "node%." Then, use the UNIX **ls** command to obtain a directory listing of */cfs*:

```
%nsh
nsh% ls -l /cfs
```

Errors

`No Service nodes available`

You have executed **nsh** without the **-s** option, and there are no service nodes available.

`cubeinfo: Bootcube in progress`

A **bootcube** is in progress. Try again when the **bootcube** is completed.

See Also

mkdev, **tapemode**
iPSC®/2 and iPSC®/860 User's Guide

PLOGON, PLOGOFF PLOGON, PLOGOFF

Log process related information such as cube allocation, cube deallocation, process initiation and process completion. This command is for use by the system administrator only.

Syntax

plogon [**-a**] [**-c**] [**-i**] [*pathname*]
plogoff

Arguments

-a	Append logging information to *pathname* (if it exists).
-c	Only cube allocations and deallocations are logged.
-i	Query for current output log file name. No change of logging state.
pathname	The output log file. If no file is specified, the default is */usr/ipsc/log/process.log*.

Description

The **plogon** command opens a log file and logs process related information. You can specify a log file on the command line or use the default */usr/ipsc/log/process.log*. The **plogon** command logs seven types of actions: execution of **bootcube**, allocating a cube, loading a process onto nodes, completing a process, executing **plogon** or **plogoff**, and releasing a cube. With the **-c** option, **plogon** logs only the allocation and release of cubes. Upon completion **plogon** always prints the name of the output log file.

PLOGON, PLOGOFF *(cont.)*

PLOGON, PLOGOFF *(cont.)*

Starting Process Logging

When executing **plogon**, one of three possible entries is created in the log file. For example:

```
   Cube 2          Wed Apr 27 13:59:41 1989  Plogon: Start logging.
```

is the entry created when a log file is first opened by **plogon.**

```
Cube2          Wed Apr 27 13:58:51 1989
  Plogon: Already logging to this file (/usr/ipsc/log/process.log)
```

is the entry created when **plogon** is run twice specifying the same output log file with no intervening **plogoff** execution. Running **plogon** twice in a row may be desirable in order to change the state of the (**-c**) cube-only switch. Logging will continue to the same output file.

```
Cube 2        Wed Apr 27 13:49:41 1989
  Plogon:  Switching to log file /tmp/process.log.
```

is the entry created when **plogon** is run twice with no intervening **plogoff** with the second **plogon** specifying a new output log file. The new logfile will have a **plogon** start logging entry.

Booting the Cube

When the first steps of **bootcube** are execute if process logging is still running from a previous invocation of **plogon** an entry will be made in the log file of the form:

```
Cube 2:     Wed Apr 27 13:49:41 1989
    Commserver terminated.
```

Allocating a Cube

When a cube is allocated, the following information is logged: cube ID, time stamp, the action (a **getcube**), the cube owner, the cube name, the host network name, the process ID of the process that ran the **getcube**, and the cube type. For example:

```
Cube 2
Wed Apr 27 13:59:41 1989
    Getcube: owner joelc name cube1 host acda pid 4470 type 1m8n4
```

PLOGON, PLOGOFF *(cont.)*

PLOGON, PLOGOFF *(cont.)*

This cube has a cube ID of 2, its index in the cubeinfo table. It was allocated on Wednesday, April 27, 1989 at 13:56:41. The user executing the command was *joelc*. The name of the cube was *cube1*, and the network name of its host was *acda*. The process that ran the **getcube** was 4470. If a program issued **getcube**, this *pid* is the UNIX *pid* of that program. If a user issued **getcube** at the command line, this *pid* is the UNIX *pid* of the process that was created to run the **getcube**. This *pid* is also displayed in the corresponding **relcube** log entry. The allocated cube has one node with 8M bytes of memory in slot 4.

Loading a Process Onto Nodes

When a process is loaded onto a cube, the following information is logged: the cube ID, timestamp, the action (a new process), the new process *pid*, the set of physical slot IDs, and the filename of the loaded process. The set of physical slot IDs is represented as a binary number displayed in hex. A set bit indicates that the node in that slot was loaded. For example, 0x00000000F identifies slots 0, 1, 2, and 3; 0x000000001 identifies slot 0. Here is a typical entry:

```
Cube 4 Wed Apr 27 13:59:41 1989
New process: pid 0 nodeset 0x000000010 /usr/joelc/tests/hello
```

This cube has a cube ID of 4. It was loaded on Wednesday, April 27, 1989 at 13:63:41. The node process loaded had an NX/2 *pid* of 0. It was loaded onto the node in slot 4. The filename of the process was */usr/joelc/tests/hello*.

Completing a Process

When a node process completes, the following information is logged: the cube ID, time stamp, the action (process done), the NX/2 *pid* of the completed process, the node slot on which the process ran, and the process exit code. For example:

```
Cube 4 Wed Apr 27 13:61:41 1989
Process done: pid 0 nodeset 0x10 e-code 0
```

This cube has a cube ID of 4. A process running on this cube completed on Wednesday, April 27, 1989 at 13:57:41. The loaded process had an NX/2 *pid* of 0. It was loaded onto the node in slot 4. Its exit code was 0.

PLOGON, PLOGOFF *(cont.)*

PLOGON, PLOGOFF *(cont.)*

Releasing a Cube

When a cube is deallocated, the following information is logged: cube ID, time stamp, the action (a **relcube**), the cube owner, the cube name, the host network name, the process ID of the process that ran the corresponding **getcube**, and the cube type. For example:

```
Cube 2 Wed Apr 27 13:63:41 1989
    Relcube: owner joelc name cube1 host acda pid 4470 type 1m8n4
```

This cube has a cube ID of 2, its index in the cubeinfo table. It was released on Wednesday, April 27, 1989 at 13:57:41. The user executing the command was *joelc*. The name of the cube was *cube1*, and the network name of its host was *acda*. The *pid* of the process that ran the corresponding **getcube** was 4470. The released cube consisted of one node with 8M bytes of memory in slot 4.

Example

To turn on logging and store the log in */usr/joelc/mylog*, enter the following:

```
# plogon /usr/joelc/mylog
```

To turn off logging, enter the following:

```
# plogoff
```

Error

`Permission denied`

Only the superuser can run a **plogon** or **plogoff**.

`No active log exists`

The **plogoff** command cannot find a log function to terminate.

PROF860 PROF860

Display profile data.

Syntax

prof860 [**-tcan**] [**-ox**] [**-g**] [**-z**] [**-h**] [**-s**] [**-m** *mdata*] [*prog*]

Arguments

The mutually exclusive options **-t**, **-c**, **-a**, and **-n** determine the type of sorting of the output lines.

-t	Sort by decreasing percentage of total time (default).
-c	Sort by decreasing number of calls.
-a	Sort by increasing symbol address.
-n	Sort lexically by symbol name.

The mutually exclusive options **-o** and **-x** specify the printing of the address of each symbol monitored.

-o	Print each symbol address (in octal) along with the symbol name.
-x	Print each symbol address (in hexadecimal) along with the symbol name.

The following options may be used in any combination.

-g	Include non-global symbols (static functions).
-z	Include all symbols in the profile range, even if associated with zero number of calls and zero time.
-h	Suppress the heading normally printed on the report. (This is useful if the report is to be processed further.)
-s	Print a summary of several of the monitoring parameters and statistics on the standard error output.
-m *mdata*	Use file *mdata* instead of *000.mon.out* as the input profile file.
prog	The symbol table in the executable *prog* (**a.out** by default) is read and correlated with a profile file (*000.mon.out* by default).

PROF860*(cont.)* PROF860 *(cont.)*

Description

The **prof860** command interprets a profile file produced by the **monitor(3C)** function. The symbol table in the executable *prog* is read and correlated with a profile file *mdata*. For each external text symbol the percentage of time spent executing between the address of that symbol and the address of the next is printed, together with the number of times that function was called and the average number of milliseconds per call.

A program creates a profile file if it has been linked with the **-p** option of **icc** or **if77**. This option arranges for calls to **monitor(3C)** at the beginning and end of execution. It is the call to **monitor(3C)** at the end of execution that causes a profile file to be written. The number of calls to a function is tallied if the **-p** option was used when the file containing the function was compiled.

The name of the file created by a profiled program is controlled by the environment variable PROFDIR. If PROFDIR does not exist, *nnn.mon.out*, where *nnn* is the node ID of the node in the allocated cube, is produced in the directory that is current when the program terminates. If PROFDIR = *string*, *string*/*nnn.progname* is produced where *progname* consists of argv[0] with any path prefix removed, and *nnn* is the program's node. If PROFDIR is the null string, no profiling output is produced.

The times reported in successive identical runs may show variances of 5% or more, because of varying cache-hit ratios due to sharing of the cache with other processes. Even if a program seems to be the only one using the machine, hidden background or asynchronous processes may blur the data. In rare cases, the clock ticks initiating recording of the program counter may beat with loops in a program, grossly distorting measurements.

Call counts are always recorded precisely.

Only programs that call **exit(2)** or return from **main** will cause a profile file to be produced, unless a final call to **monitor(3C)** is explicitly coded.

The use of the **-p** option to invoke profiling imposes a limit of 1000 functions that may have call counters established during program execution. For more counters, you must call **monitor(3C)** directly. The number of call counters used will be reported automatically by the **prof860** command whenever the number exceeds 5/6 of the maximum.

prof860 is not for use with programs loaded with remote host software.

See Also

UNIX Programmer's Reference Manual: **monitor(3C)**

REBOOTCUBE REBOOTCUBE

Reload the NX/2 operating system onto the nodes of the cube.

Syntax

rebootcube

Arguments

None

Description

Allows a user (who is not superuser) to reboot the system in case the superuser is not available to reboot the system (with **bootcube**) in case of some problem with the host daemons or to reload and reset the node operating system. This command sets its environment to a known safe state, and calls **bootcube** with predefined flags (the full version of **bootcube** is only available to the superuser). This command can be executed from either the SRM or a remote host.

The **rebootcube** command should be used with caution because it resets all cubes in the system, even if there are multiple users.

Errors

See **bootcube**

See Also

bootcube

RELCUBE RELCUBE

Release cube(s) previously allocated with **getcube**.

Syntax

relcube [**-c** *cubename* | **-a** | **-f** *cubeid* | **-F**]

Arguments

-c *cubename*	Allows you to release a specific named cube. Names are assigned with **getcube**. For a list of cube names, use the **cubeinfo** command. If the specified cube does not exist, an error is returned. Without this argument, the command releases the currently attached cube. If you have not allocated a cube, an error is returned.
-a	Allows you to release all of the cubes that you own on the system from which you invoked **relcube**.
-f *cubeid*	(Available to superuser only.) Allows superuser to release the cube specified by *cubeid*. For a list of cube IDs, superuser must execute **cubeinfo**.
-F	(Available to superuser only.) Releases all cubes in a system.

Description

The **relcube** command allows you to release one or all cubes previously allocated with the **getcube** command or call. With no arguments, **relcube** releases the currently attached cube; with the **-c** argument, it releases the named cube, and with the **-a** argument it releases all of the cubes that you own on the system. The **-f** and **-F** arguments can be used by the superuser to release cubes that users have forgotten to release. Only one of the above arguments can be used with **relcube** at a time.

If you log out from an SRM host before entering the **relcube** command, all the cubes that you own are automatically released unless you invoke **getcube** with the UNIX **nohup** command as described under **getcube**. On a remote host, cubes are not automatically released when you log out.

After **relcube** has completed releasing the cube(s), it displays the number of cubes released.

RELCUBE *(cont.)*

RELCUBE *(cont.)*

Examples

1. Release the current cube:

```
relcube
```

2. Release the cube named "alpha":

```
relcube -c alpha
```

3. Release all cubes owned by you:

```
relcube -a
```

Errors

`relcube: Cubename does not exist`

Use **cubeinfo** to find out cube names.

`relcube: Commser not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with the **bootcube** or **rebootcube** command.

`relcube: Too many options specified`

Use one option only.

`relcube: Bootcube in progress`

A **bootcube** is in progress. Try again when the **bootcube** is completed.

See Also

getcube, cubeinfo

REMOTE HOST REMOTE HOST

Remote iPSC development utilities: the C compiler, Fortran compiler, linker, assembler, and librarian. These utilities are run from a remote development machine.

Syntax

rcc [**-cpp**] [**-h** *srmname*] [*cc_options*] *filename...*
rf77 [**-cpp**] [**-h** *srmname*] [*f77_options*] *filename...*
rld [**-h** *srmname*] [*ld_options*] *filename...*
ras [**-h** *srmname*] [*as_options*] *filename...*
rar [**-h** *srmname*] [*ar_options*] *filename...*

Arguments

-cpp	Run the remote host C preprocessor before compiling the files on the System Resource Manager (SRM). This allows use of include files that exist only on the remote host.
-h *srmname*	Run the remote utility on the specified SRM. If this switch is not used, the remote environment software uses the first System Resource Manager available in the *srms* file.
**_options*	You can use any of the options that are available to the utility when it is running on the SRM (not the options the utility may have when running on the remote development machine). Refer to the various compiler and UNIX manuals for valid options and descriptions.
filename...	The necessary input file(s) for the specified remote utility.

Description

CX node programs must be compiled, assembled, and linked on an iPSC SRM. Therefore, CX node programs developed on a remote development machine may be processed with these utility programs before being run on a cube. For RX nodes you must use the **icc** or **if77** compilers.

The **rcc**, **rf77**, **rld**, **ras**, and **rar** programs are designed for used on Sun 3 and Sun 4 workstations. These utiities allow you to generate executables for the iPSC SRM or cube remotely by sending the files from your remote development machine to an available SRM. Unless you specify an SRM by using the **-h** switch, the remote environment software finds an SRM that can be used for execution, as listed in the *srms* file.

REMOTE HOST *(cont.)*

REMOTE HOST *(cont.)*

Once found, the remote utility copies all of the files and the command line to that SRM. The status of the command is returned to the remote development machine along with all resulting files. Since no files remain on the SRM, the remote utilities can run on any SRM.

The resulting files have these extensions:

.o	iPSC object files. These files are produced by **ras, rcc -c**, and **rf77 -c**.
.s	iPSC Assembly Language source files, produced by **rcc -S** and **rf77 -S**.
.i	Preprocessed C source file, produced by running the C preprocessor on the SRM using **rcc -P**.
no extension	SRM executable program. As with executable programs on the SRM, some are intended to be run on the nodes and some on the SRM, depending on how they were linked. If they are1rccrf77rldrasrar.doc node executable, they must be "loaded." In both cases, trying to run them on the remote development machine will produce unpredictable results.
.a	Archives made by **rar**. You can only link to these files on the SRM.

	CX	RX
SRM	**cc, f77**	**icc, if77**
Sun 3/4	**rcc, rf77** or **rlogin** to SRM	**icc, if77**
Other workstations	**rlogin** to SRM	**rlogin** to SRM

Examples

1. This command preprocesses a file named *ctest.c* on the remote development machine and compiles "ctest" on the first SRM available, producing the file *a.out* as output:

```
rcc -cpp -c ctest.c -node
```

2. This command, entered from a remote development system, compiles and links the Fortran source file *ftest.f* to produce the file *ftest*:

```
rf77 -o ftest ftest.f -node
```

REMOTE HOST *(cont.)*

REMOTE HOST *(cont.)*

3. This command, entered from a remote development system, performs a remote link of the object files *ldtest1.o* and *ldtest2.o* to produce the file *ldtest.o*:

```
rld -r ldtest1.o ldtest2.o -o ldtest.o
```

4. This command, entered from a remote development system, performs a remote assemble of a file named *astest.s* to produce the file *astest.o*:

```
ras astest.s
```

5. This command, entered from a remote development system, performs a remote archive of two object files to produce the file *local_archive.a*

```
rar cr local_archive.a local_obj1.o local_obj2.
```

Remote Development Machine Errors

`No srm that matched your request was found`

Could not find an SRM with a cube of the type requested.

`Cannot read the length of` *`filename`* `from ripscd`

Attempt to read the file length from the remote iPSC daemon on the SRM failed. Check the connection between the remote host and the SRM.

`Cannot open` *`filename`* `to write`

Attempt to open the file to write failed. Check connections.

`Cannot read` *`filename`* `from ripscd`

Attempt to read the file from the remote iPSC daemon on the SRM failed. Check connections.

`Cannot open` *`filename`* `to read`

Attempt to open the file to read failed. Check the connection between the remote host and the SRM.

REMOTE HOST *(cont.)*

REMOTE HOST *(cont.)*

`Cannot write the length of filename to ripscd`

Attempt to read the file length to the remote iPSC daemon on the SRM failed. Check the connection between the remote host and the SRM.

`Cannot write filename to ripscd`

Attempt to write the file failed. Check the connection between the remote host and the SRM.

`Cannot write command to ripscd`

Attempt to write command to the remote iPSC daemon on the SRM failed. Check the connection between the remote host and the SRM.

`SRM not reachable`

Attempt to read the file length from the remote iPSC daemon on the SRM failed. Check the connection between the remote host and the SRM.

See Also

UNIX System V Programmer's Reference: **cc, ld, as, ar**
iPSC®/860 Fortran Language Reference Manual: Compile Time Options (for the **f77** compile command); *iPSC®/2 and iPSC®/860 C Language Reference Manual*

SHOWVOL SHOWVOL

Display information about available drives.

SRM and NSH Syntax

showvol

Arguments

None

Description

The **showvol** command displays information about all available iPSC Concurrent I/O drives in the system. For each drive present, it displays the volume number, the designation of the I/O node to which it is connected, the size of the drive in bytes, space remaining on the drive in bytes, and the drive type, model, and revision.

Each disk drive belonging to the CFS has a unique volume number which is used to designate drives in the **cbackup** and **crestore** commands. The node number is the number of the I/O node to which the drive is connected and is in the range from 641 to 767. It is equal to 640 plus the physical slot number of the compute node whose channel 7 is connected to the I/O node. The drive number is local to the I/O node.

Example

Show current drive information:

```
% showvol

Vol. Node Drive Size        Free        Vendor Model    Rev.

  0  642  0     649,359,360 648,548,352 MAXTOR XT-8760S B2
  1  646  2     649,359,360 648,552,448 MAXTOR XT-8760S B2
  2  642  1     649,359,360 648,552,448 MAXTOR XT-8760S B2
  3  646  3     649,359,360 648,552,448 MAXTOR XT-8760S B2
```

SHOWVOL *(cont.)*

SHOWVOL *(cont.)*

Errors

`showvol: Disk initialization not complete, no anchor node`

Try rebooting with the **bootcube** or **rebootcube** command to initialize volumes.

`showvol: Bootcube in progress`

A **bootcube** is in progress. Try again when the **bootcube** is completed.

See Also

cbackup, crestore

STAR STAR

Saves and restores files on magnetic media.

Syntax

star [-]**c**[**wfb**] *starfile blocksize filename...*
star [-]**r**[**vwb**] *starfile blocksize* [*filename...*]
star [-]**t**[**vf**] *starfile*
star [-]**u**[**vwb**] *starfile blocksize* [*filename...*]
star [-]**x**[**vlmopwf**] *starfile* [*filename...*]

Description

The **star** command saves and restores files on magnetic media. Its actions are controlled by the *key* argument. The *key* is a string of characters containing one function letter (**c, r, t, u, x**) and possibly followed by one or more function modifiers (**w, f, b**). Other arguments to the command are files (or directory names) specifying which files are to be archived or restored. In all cases, appearance of a directory name refers to the files and (recursively) subdirectories of that directory.

Function

c Create a new *starfile* and write the named files onto it.

r Write the named files on the end of the *starfile*. This option does not work with quarter-inch archive tapes.

t List the table of contents of the *starfile*.

u Add the named files to the *starfile* if they are not there or if they have been modified since they were last archived. This option does not work with quarter-inch archive tapes.

x Extract the named files from the *starfile*. If a named file matches a directory with contents written onto the tape, this directory is (recursively) extracted. The owner, modification time, and mode are restored (if possible). If no filename arguments are given, all files in the archive are extracted. If multiple entries specifying the same file are on the tape, the last one overwrites all earlier versions.

STAR *(cont.)* STAR *(cont.)*

Function Modifiers

w Wait for user confirmation before taking the specified action. If you use **w**, **star** displays the action to be taken followed by the file name, and then waits for a *y* response to proceed. No action is taken on the named file if you type anything other than a line beginning with *y*.

f Use the next argument as the name of the *starfile*. If **f** is omitted, use the device indicated by the *TAPE* environment variable, if set. Otherwise, use */dev/rmto*. If *starfile* is given as **-**, **star** writes to the standard output or reads from the standard input, whichever is appropriate. Thus, **star** can be used as the head or tail of a filter chain. The **star** command can also be used to copy hierarchies. For example:

```
cd fromdir; star cf - . | (cd todir; star xfp -)
```

b Use the next argument as the blocking factor for tape records. The default blocking factor is 20 blocks. The block size is determined automatically when reading tapes (functions **x** and **t**). The maximum blocking factor is determined only by the amount of memory available to **star** when it is run. Larger blocking factors result in better throughput, longer blocks on nine-track tapes, and better media utilization. The blocking factor **b** must be an even number greater than 0 and less than 20.

l Display error messages if all links to archived files cannot be resolved. If **l** is not used, no error messages are printed.

m Do not extract modification times of extracted files. The modification time will be the time of extraction.

o Suppress information specifying owner and modes of directories which **star** normally places in the archive. Such information makes former versions of **star** generate an error message like:

```
<filename>/: cannot create
```

p Restore the named files to their original modes, ignoring the present **umask(2)**. *SetUID* and sticky information are also extracted if you are the superuser. This option is only useful with the **x** function.

v Verbose. Normally, star does its work silently; this option displays the name of each file star treats, preceded by the function letter. When used with the **t** function, **v** displays the *tarfile* entries in a form similar to ls -l. Do not use **c** and **v** together.

STAR *(cont.)*

STAR *(cont.)*

Examples

To save some files and directories called *myfiles*, *ccode*, and *src* on the streamer:

```
star c myfiles ccode src
```

To examine a list of what is archived on a specific streamer:

```
star tv
```

To extract only those files from the above tape which were archived from the directory *src*:

```
star xv src
```

To extract all of the files archived on the tape:

```
star xv
```

To quickly backup all files in */bin* to streamer.

```
star cb 20 /bin
```

Errors

None

See Also

UNIX **tar** command

STARTCUBE STARTCUBE

Start a process executing on the cube.

Syntax

startcube [**-c** *cubename*] [**-p** *pid*] [*node...*]

Arguments

-c *cubename*	The name of the cube that you wish to start a process on. If no name is given, the attached cube is used.
-p *pid*	For CX nodes, the *pid* of the process that you wish to start. If no *pid* is given, all processes are started. For RX nodes, *pid* is 0.
node...	The node(s) on which processes are to be started. If no node is specified, processes on all nodes are started.

Description

This command is used to start processes that were loaded with the **load** command using the **-H** switch. It has no effect on processes that are already running. It is not an error to use **startcube** for a nonexistent process.

Examples

1. Start all processes (on the current attached cube) loaded with the **-H** argument:

```
startcube
```

2. Start process 3 on nodes 5 and 6 in cube named "alpha":

```
startcube -c alpha -p 3 5 6
```

STARTCUBE *(cont.)*

STARTCUBE *(cont.)*

Errors

`startcube: Invalid pid`

Pid must not be negative. For RX nodes, *pid* must be 0.

`startcube: Invalid node`

Node number must be in range of cube size.

`startcube: There is no attached cube`

Use **getcube** to get a cube.

`startcube: Cubetype does not exist`

Use **cubeinfo** to find names of available cubes.

`startcube: Bootcube in progress`

A **bootcube** is in progress. Try again when the **bootcube** is completed.

See Also

load

STREAM STREAM

Transfers large amounts of data to and from tape.

Syntax

stream [*Kbytes_of_buffer*] [*source*] [*destination*]

Arguments

Kbytes_of_buffer	The number of kilobytes worth of buffer RAM desired. By default, **stream** uses as much as it can get from the system.
[*source*] or [*destination*]	Any file or partition. Use */dev/tape* to refer to the tape device as a source or destination. Use **-** to refer to *stdin* or *stdout* as a source or destination.

Description

The **stream** command is a bidirectional, high speed utility for transferring large amounts of data to and from tape using large RAM buffers. It will use as much RAM as is available in the system, up to any process size limit imposed by your configuration of UNIX, or up to the optional number of bytes specified for the buffer.

The **stream** command may be used for making "disk image" copies of files or disk partitions, or as a filter to enable standard programs like **cpio** or user-written programs to access the tape device.

Example

To archive the entire contents of a directory and lower subdirectories to tape using the UNIX **cpio** utility:

```
find . -print | cpio -ov | stream - /dev/tape
```

To do the above task with no more than 200K of RAM buffers:

```
find . -print | cpio -ov | stream 200 - /dev/tape
```

See Also

cptape, star

SYSLOG SYSLOG

Send the output of a host process to the file server handling I/O from the nodes.

Syntax

syslog [-e]

Argument

-e	Send data to standard error of file server.

Description

The default (no switch specified) is to send data to the standard output of the file server. The **-e** option sends it to standard error.

A file server, to handle I/O for the nodes, is automatically created on the system resource manager when you allocate a cube with **getcube**. The **syslog** command can be used as a filter to send the standard output of a host process to the standard output or standard error of the file server. Thus, you can ensure that I/O statements from both the node and host processes go to the same */dev/file*.

The **syslog** command reads from its standard input, and when a newline character is encountered, it sends all the characters read, up to and including the newline character, as a "write-to-standard-{output/error}" message to the file server.

Examples

1. Send all the characters that are the output of the executable **host** to the standard output of the file server.

```
host | syslog
```

2. Send all the characters written to the standard output of the executable **host** to the standard error of the file server.

```
host | syslog -e
```

SYSLOG *(cont.)*

SYSLOG *(cont.)*

Errors

`setpid: Bootcube in progress`

A **bootcube** is in progress. Try again when the **bootcube** is completed.

See Also

newserver, getcube

TAPEMODE TAPEMODE

Changes operating mode for tape drives.

NSH Syntax

tapemode [*switches*] *devicename*

Arguments

switches Operates on specific attributes of the tape device:

-d *density*	where *density* is 800, 1600, 3200, or 6250 bpi.
-s *speed*	where *speed* is 0 (low) or 1 (high).
-b *blocksize*	where *blocksize* is in number of bytes; a value of zero means variable blocksize.
-r	perform a rewind after the device is closed.
-n	do not perform a rewind after the device is closed.
-B	tape device placed in buffered mode of operation.
-N	tape device placed in non-buffered mode of operation.
-t *retries*	where *retries* is the total retry limit.
-p *retries*	where *retries* is the limit of the number of retries without an erase for writes (retries in place).
-q	displays new mode, default if no other switches used.

devicename the name(s) for the file(s) representing the device. The name(s) must be complete pathname(s).

NOTE

This command only operates within the **nsh** command.

TAPEMODE *(cont.)*

TAPEMODE *(cont.)*

Description

The **tapemode** command changes the mode in which a specified tape drive operates. With switches, it allows for varying attributes of the tape device such as tape density, speed, and buffered on non-buffered mode of operation. It then sets the blocksize, rewind parameters, selects the number of retries, and retries in place.

If there are no switches on the command line, the current mode of the tape drive is displayed :

```
device pathname : [Vendor] [Model] [Revision]
          [blksize] [speed] [density] [buffered]
          [rewind] [retry limit] [retry in place]
```

where :

Vendor	Name of the vendor of the device.
Model	Model of the device.
Revision	Revision level of the device.

The rest correspond to switches described above.

Examples

The following example:

```
tapemode /cfs/mt0
```

will display the following:

```
/cfs/mt0:    STK     2925              CS05
                blksize 4096   speed 1   density 6250 buffered
                rewind  retry  limit 0   in place 0
```

The following example sets the blocksize to 1024 bytes.

```
tapemode -b 1024 /cfs/mt0
```

TAPEMODE *(cont.)*

TAPEMODE *(cont.)*

Errors

`Invalid <blksize> must be less than 262144.`

The *blocksize* specified with the **-b** switch was not less than 262144.

`Invalid <blksize> must be a multiple or submultiple of <BLOCKSIZE>`

The *blocksize* specified with the **-b** switch must be a multiple or submultiple of 4096.

`Invalid <retries in place> greater than <total retries>`

The number of retries inplace specified with the **-p** switch must not be greater than the total number of retries.

`device``: not in CFS`

The *devicename* specified on the command line is not with the CFS file system.

`device``: not found`

The *devicename* specified on the command line does not exist.

`device``: not a device`

The *devicename* specified on the command line does not refer to a *device* special file.

`device``: not a tape drive`

The *devicename* specified on the command line does not represent a tape device. The *devicename* must refer to a tape device.

`Requested density not available`

The *density* specified with the **-d** switch is not supported on the specified device.

`Fixed block length not valid`

The *blocksize* specified with the **-b** switch is not supported on the specified device.

TAPEMODE *(cont.)*

TAPEMODE *(cont.)*

`Not at BOT`

The tape drive was not at the Beginning Of Tape when a density change was requested with the **-d** switch.

`Drive does not accept these values`

This is the default message when the device will not accept the tapemode commands and the error is not of the previously listed error.

See Also

nsh, mkdev

WAITCUBE WAITCUBE

Wait for process(es) on node(s) to finish before proceeding.

SRM Syntax

waitcube [**-c** *cubename*] [**-p** *pid*] [**-f**] [**-i**] [**-s**] [*node...*]

NSH Syntax

waitcube -p *pid* [**-f**] [**-i**] [*node...*]

Arguments

-c *cubename*	The name of the cube that contains the process to be waited for. If no name is given, the attached cube is used. If the cube does not exist, an error is returned.
-p *pid*	For CX, the process ID *(pid)* of the process to be waited for. If no *pid* is given, **waitcube** waits for all processes to finish. For RX nodes, *pid* is 0.
-f	Causes **waitcube** to return when the first process that meets the requirements specified by the other switches is completed. The default is to wait for all specified processes.
-i	Allows an interrupt signal (caused by pressing the **<Delete>** key) to kill the processes you are waiting for. As soon as the processes are killed, **waitcube** exits.
-s	Allows an interrupt signal (caused by pressing the **<Delete>** key) to be passed to node processes. The **waitcube** command does not exit until specific processes terminate.
node...	The node(s) on which the process you are waiting for runs. If no node is specified, **waitcube** waits for all nodes.

WAITCUBE *(cont.)*

WAITCUBE *(cont.)*

Description

This command is useful when you are running multiple applications from a shell script. You can start an application, do a **waitcube**, and when it returns (because the application is finished), the shell script can start the next application. It is not an error to use **waitcube** to wait for a nonexistent process. If you are running a job with node-only programs (no host programs), **waitcube** provides you with the capability of reading the keyboard from the nodes.

You can redirect standard input. For example, if you had a host file consisting of data for the cube, you could issue **waitcube** and redirect its input as follows:

```
%load node
%waitcube < node_data
%relcube -a
relcube released 1 cube
```

To reliably receive input from the terminal, you can also issue a **waitcube**. Subsequent data then entered from the terminal is sent directly to the node program.

```
%cat myfile | waitcube
```

or

```
%waitcube < myfile
```

or

```
%echo "Hello" | waitcube
```

The **-i** and **-s** switches allow you to use the **<Delete>** key to produce an interrupt signal to stop node programs. Without either of these switches, the interrupt signal kills the **waitcube** command without affecting node processes. If you use **-i**, pressing the **<Delete>** key will kill the node processes that you were waiting for, and **waitcube** will then exit.

The **-s** switch is for use with the shell on a node. After loading the shell, you must use **waitcube** to allow terminal input to pass freely to the nodes through the file server. If you use the **-s** switch with the command, pressing the **<Delete>** key while in the shell sends an interrupt signal to all processes in your cube so you can stop processes as if they were running on the host.

WAITCUBE *(cont.)* WAITCUBE *(cont.)*

Examples

1. Wait for all processes on all nodes of attached cube to be completed:

```
waitcube
```

2. Wait for process 5 on cube named "alpha" to be completed:

```
waitcube -c alpha -p 5
```

3. Wait for the first process on node 6 to be completed:

```
waitcube -f 6
```

Errors

`waitcube: Invalid pid`

The *pid* must not be negative. For RX nodes, the *pid* must be 0.

`waitcube: Invalid node`

Node number must be in range of cube size.

`waitcube: There is no attached cube`

Use **getcube** to get a cube.

`waitcube: Cubetype does not exist`

Use **cubeinfo** to find names of available cubes.

`waitcube: Bootcube in progress`

A **bootcube** is in progress. Try again when the **bootcube** is completed.

See Also

getcube, nsh

WHATIS WHATIS

Search the *whatis* database for strings. This command describes the **man** implementation on the SRM only.

Syntax

whatis *name* ...

Arguments

name	The name of a command or call.

Description

whatis searches a set of database files containing short descriptions of system commands for *name* and displays the result on the standard output. Only exact word matches are displayed.

Examples

```
% whatis flushmsg
FLUSHMSG (3)          - Flush specified messages from the system.
FLUSHMSG (3f)         - Flush specified messages from the system.
```

See Also

man, manpath, apropos, whatis

iPSC®/2 and iPSC®/860 C LIBRARY CALLS 2

INTRODUCTION

This chapter documents each of the iPSC routines in the two C system interface libraries: host and node. These routines allow you to perform such tasks as accessing and releasing cubes, loading, starting, and killing processes, and passing messages between processes.

Routines for host processes are found in the */usr/lib/libhost.a* library and routines for node processes are found in the */usr/lib/libnode.a* library.

There are two versions for many of the C system routines. The standard version, described in this chapter, terminates a process when an error occurs and sends a message to standard output describing the error. The error return version does not terminate a process when an error occurs. Refer to Appendix C for more information on the error return version and a list of possible error messages with descriptions.

The routines are listed alphabetically. Appendix A gives a summary of each routine with a synopsis, brief description, and whether the routine is available for host, node, or both programs.

attachcube()	**estat()**	**gland()**	**infonode()**	**msgcancel()**
cprobe()	**esub()**	**glor()**	**infopid()**	**msgdone()**
cread()	**etos()**	**glxor()**	**infotype()**	**msgwait()**
createstruc()	**flick()**	**gopf()**	**iodone()**	**myhost()**
crecv()	**flushmsg()**	**gray()**	**iomode()**	**mynode()**
csend()	**fstat()**	**gsendx()**	**iowait()**	**mypid()**
csendrecv()	**gcol()**	**gshigh()**	**iprobe()**	**newserver()**
ctohc()	**gcolx()**	**gslow()**	**iread()**	**nodedim()**
ctohd()	**gdhigh()**	**gsprod()**	**irecv()**	**numnodes()**
ctohf()	**gdlow()**	**gssum()**	**isend()**	**plogoff()**
ctohl()	**gdprod()**	**gsync()**	**isendrecv()**	**plogon()**
ctohs()	**gdsum()**	**handler()**	**iseof()**	**relcube()**
cubeinfo()	**getcube()**	**hrecv()**	**iwrite()**	**relstruc()**
cwrite()	**getiphosts()**	**hsend()**	**killcube()**	**restrictvol()**
dclock()	**gland()**	**hsendrecv()**	**killproc()**	**setiomode()**
eadd()	**glhigh()**	**htocc()**	**killsyslog()**	**setiphost()**
ecmp()	**gllow()**	**htocd()**	**led()**	**setpid()**
ediv()	**glnv()**	**htocf()**	**load()**	**setsyslog()**
emod()	**glor()**	**htocl()**	**lsize()**	**stoe()**
emul()	**glprod()**	**htocs()**	**masktrap()**	**waitall()**
eseek()	**glsum()**	**hwclock()**	**mclock()**	**waitone()**
esize()	**glxor()**	**infocount()**		

The C library routines above are documented as follows:

- name of the routine and brief description
- synopsis
- parameter declarations
- return values where applicable
- example
- environment
- languages
- detailed description including input parameters
- error messages returned by this routine
- list of related routines

ATTACHCUBE() ATTACHCUBE()

Make a different cube the current cube.

Synopsis

attachcube(*cubename*)

Parameter Declarations

char **cubename*;

Return Value

None

Example

This example first allocates two cubes with **getcube()**. The last cube allocated becomes the current cube. When you need to use the other cube, use **attachcube()**.

```
static char cube1[] = "alpha";
static char cube2[] = "beta";

getcube(cube1, "32", NULL, 0);
getcube(cube2,"8vx","trainon",1);          /* Current cube is "beta" */
     .
     .
attachcube(cube1);                         /* Current cube is "alpha" */
     .
     .
attachcube(cube2);                         /* Current cube is "beta" */
     .
     .
```

ATTACHCUBE() *(cont.)*

ATTACHCUBE() *(cont.)*

Environment

Host

Languages

C, Fortran

Description of Parameters

cubename refers to a character string that specifies the name of the cube that you want to attach.

Discussion

Use **attachcube()** to change a process's current cube to another cube you previously allocated. When you call **attachcube()**, you are making the specified *cubename* your current cube and all subsequent message passing and loader calls made by the process that called **attachcube()** will apply to *cubename*.

Use **attachcube()** to run applications in several cube partitions. You can only be attached to one cube at a time. Use **getcube()** to allocate a cube and **attachcube()** to change cubes within your program.

Names of cubes belonging to a single user must be unique. Names are assigned with **getcube()**. If the parameter is null or refers to a null string, the default *cubename* becomes "defaultname." If you specify a cube that has not been allocated, an error message results and the process will be terminated. Use **cubeinfo()** to find out the names of allocated cubes.

ATTACHCUBE() *(cont.)*

ATTACHCUBE() *(cont.)*

Errors

`attachcube: Cubename does not exist`

Use **cubeinfo()** to determine cube names.

`attachcube: Commser not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with **bootcube** or **rebootcube.**

`attachcube: Lifeline not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with **bootcube** or **rebootcube.**

See Also

cubeinfo(), getcube(), relcube()

CPROBE() CPROBE()

Wait for a message to arrive. Blocks the calling process until the message is available for receipt.

Synopsis

cprobe(*typesel*)

Parameter Declarations

long typesel;

Return Value

None

Example

This example uses **cprobe()** to block any further action by the calling process until the message of the given type is available to be received. Then, after using **infocount()** to ensure that the message is of the proper length, **crecv()** is used to receive the message.

```
#define INIT_TYPE 10

long msg_id;
char msgbuf[80];
    .
    .
cprobe(INIT_TYPE);                          /* Wait for message of type 10 */
if(infocount() <= sizeof(msgbuf)){          /* If message is right length */
crecv(INIT_TYPE, msgbuf, sizeof(msgbuf));   /* receive message */
    .
    .
```

CPROBE() *(cont.)*

CPROBE() *(cont.)*

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

typesel is an integer value that specifies the type(s) of message(s) you are waiting for. 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 the probe operation will be recognized. After the first message has been received, you can use `-1` again to probe for the next message.
- If *typesel* is any negative number other than `-1`, a set of message types will be recognized. Refer to Appendix B for information on building a *typesel* mask.

Discussion

Use **cprobe()** as a synchronous call that causes a process to wait until a message of the selected type (or types) is available to be received. The type must match the type specified in the **csend()** or **isend()** sending the message. When **cprobe()** returns, you can use **crecv()** or **irecv()** to initiate the receipt of the desired message. Use the **info...()** calls to get more information about the message.

Use **iprobe()** rather than **cprobe()** when you want to determine whether a message of a selected type is pending but do not want the calling process blocked if the message is not available.

CPROBE() *(cont.)*

CPROBE() *(cont.)*

Errors

`cprobe: No pid defined`

Use **setpid()** to define a host process id.

`cprobe: Too many requests`

Use **msgwait()** for outstanding **isend()** or **irecv()** requests.

`cprobe: Invalid type`

Cannot probe for force-type messages.

See Also

infocount(), infonode(), infopid(), infotype(), iprobe()

CREAD() CREAD()

High speed synchronous read from a CFS file. Blocks the calling process until the read is complete.

Synopsis

cread(*fildes, buf, len*)

Parameter Declarations

long *fildes;*
char **buf;*
long *len;*

Return Value

None

Example

The following example uses **cread()** to read the contents of a file. The example first opens the file with the **open()** call and stores the file descriptor (required to identify the file) in the variable *fd*. Then it uses **cread()** to read the information into a buffer you created:

```
     .
     .                                        /* open a file on CFS named "mydata" */
fd = open("/cfs/mydata", O_CREAT|O_RDWR, 0644);
     .
     .
                                        /* read data in open file into buffer named "buffer" */
cread(fd, buffer, sizeof(buffer));
```

Environment

Node

Languages

C, Fortran

CREAD() *(cont.)*

CREAD() *(cont.)*

Description of Parameters

fildes the file descriptor.

buf refers to the buffer where the read data will be stored. The buffer can be any valid data type.

len an integer value that specifies the size of the buffer in bytes. Transfer sizes are limited only by the available memory for the buffer.

Discussion

Use **cread()** to read data from the file specified by the *fildes* parameter (the value returned by the **open(), creat(),** or **dup()** system calls). The **cread()** call is a synchronous call that prevents the process from executing further instructions until the read is complete and the read data is in the buffer you designate. If you do not want the calling process blocked during the read, use the **iread()** call.

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

Errors

`cread: Invalid argument`

Check arguments.

`cread: Mixed file operations`

In I/O mode 2 or 3, nodes are attempting different operations (reads and writes) to a shared file. Ensure that nodes are performing the same operation.

`cread: I/O error`

Make sure file is open and of the proper format.

See Also

cwrite(), iread(), iseof(), iwrite()
UNIX System V/386 Programmer's Reference: **open(), creat(), dup(), read()**

CREATESTRUC(), RELSTRUC() CREATESTRUC(),RELSTRUC()

Define a character array to accommodate a message structure with **createstruc()**. Close the structure with **relstruc()**.

Synopsis

createstruc(*ptr*)
unsigned long **relstruc()**

Parameter Declarations

char **ptr;*

Return Value

The size (in bytes) of the padded message.

Example

Consider an example that sends a structure from the host to a CX node and back again. The structure is defined as follows:

```
struct msg_struct_tag {
    char a:
    short s;
    char b;
    long l;
    char c;
    float f;
    char d;
    double ff;
};
```

The host code appears as follows:

```
    .
    .
struct msg_struct_tag *in_msg_struct, *out_msg_struct;
struct msg_struct_tag in_msg, out_msg;
char in_msg_array[100], out_msg_array[100];
long msg_size;
```

CREATESTRUC() ... *(cont.)*

CREATESTRUC() ... *(cont.)*

```
    .
    .
in_msg_struct = &in_msg;
out_msg_struct = &out_msg;
    .
    .
setpid(100);
createstruc(out_msg_array);
HTOCC(out_msg_struct->a,1);          /* The structure on the host to be sent */
HTOCS(out_msg_struct->s,1);             /* is out_msg_struct. Its contents */
HTOCC(out_msg_struct->b,1);             /* are copied to out_msg_array, */
HTOCL(out_msg_struct->l,1);                /* which is sent to the node. */
HTOCC(out_msg_struct->c,1);
HTOCF(out_msg_struct-f,1);
HTOCC(out_msg_struct->d,1);
HTOCD(out_msg_struct->ff,1);
array_size = relstruc();

csend(10, out_msg_array, array_size, 0, 0);

crecv(20, in_msg_array, array_size);

createstruc(in_msg_array);
CTOHC(in_msg_struct->a,1);           /* The host receives in_msg_array.*/
CTOHS(in_msg_struct->s,1);           /* The host then copies the bytes in */
CTOHC(in_msg_struct->b,1);           /* in_msg_array into the structure */
CTOHL(in_msg_struct->l,1);              /* in_msg_struct. The bytes */
CTOHC(in_msg_struct->c,1);           /* in in_msg_struct obey the byte */
CTOHF(in_msg_struct->f,1);           /* ordering convention on the host. */
CTOHC(in_msg_struct->d,1);
CTOHD(in_msg_struct->ff,1);
relstruc();
    .
    .
```

The node code appears as follows:

```
    .
    .
struct msg_struct_tag in_msg, out_msg;
crecv(10, &in_msg, sizeof(in_msg));
memcpy((char *)&out_msg, (char *)&in_msg, sizeof(in_msg));
csend(20, &out_msg, sizeof(out_msg), myhost(), 100);
    .
    .
```

CREATESTRUC() ... *(cont.)*

CREATESTRUC() ... *(cont.)*

Environment

Host

Languages

C

Description of Parameters

ptr is a pointer to a character array that will hold the node version of the message. This array is used for output of the **HTOC***x*() calls and input of the **CTOH***x*() calls, but is not a parameter in these calls.

Discussion

When you send messages between a remote host and the cube, you may need to use the byte swapping calls, **HTOC***x*() and **CTOH***x*(). The byte swapping calls are necessary if you're working on a remote workstation that does not observe Intel's byte ordering convention. If the remote workstation observes Intel's byte ordering convention, the byte swapping calls have no effect.

In a C program, a message may be a structure consisting of different types. To convert a structure, ensuring that all its elements are aligned properly, you must use the **createstruc**() and **relstruc**() calls in addition to the byte swapping calls.

On the host, define a character array whose size is at least large enough to accommodate the message structure, allowing for any padding that may be introduced to start each element on a four-byte boundary. Then, call **createstruc**() with that array as an argument and issue a byte swapping call for each element of the message structure in the order in which it appears in the structure. Finally, close the structure with a **relstruc**(). This call returns the size of the padded message in bytes.

CREATESTRUC() ... *(cont.)*

CREATESTRUC() ... *(cont.)*

When you send a message, initialize the structure with **createstruc()**, issue the **HTOC***x*() calls and close the structure with **relstruc()**. What happens is that the data from your structure are copied into the array. The bytes in this array are aligned and stored according to Intel's byte ordering convention. When you send the message, send this new array, not your original structure. The program on the receiving node has a receiving structure defined as the same type as your original structure. This receiving structure is the receive buffer. There is no byte swapping that needs to be done on the node.

When you receive a message, initialize the structure with **createstruc()**, issue the **CTOH***x*() calls and close the structure with **relstruc()**. Indicate the array with the receive call. The data from the array are copied into the structure.

Errors

None

See Also

CTOH*x*()

CRECV() CRECV()

Receive a message and wait for the receive to complete before proceeding.

Synopsis

crecv(*typesel, buf, len*)

Parameter Declarations

long *typesel;*
char **buf;*
long *len;*

Return Value

None

Example

The following example contains a **crecv()** statement that will initiate the receive as soon as a message of any type is available. It will receive the message into the buffer *buf1*, of length 256.

```
#define BUFLEN1 256.

char buf1[BUFLEN1];
     .
     .
                              /* Receive the first message to arrive and store it in 'buf1'. */
crecv(-1, buf1, BUFLEN1);
```

Environment

Node, Host

Languages

C, Fortran

CRECV() *(cont.)*

CRECV() *(cont.)*

Description of Parameters

typesel is an integer value that specifies the type(s) of message you are waiting for. 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 non-force-type message to arrive for the process that initiated the receive operation will be recognized. After the first message has been received, you can use `-1` again to receive the next message.
- If *typesel* is any negative number other than `-1`, a set of message types will be recognized. Refer to Appendix B for information on building a *typesel* mask.

buf refers to the buffer where the received message will be stored. The buffer can be any valid data type. It is recommended that data types match in send and receive operations.

len is an integer value that specifies the size of the message buffer in bytes. The maximum number of bytes that you can send in a single message to or from a host process is 256K. There is no limit on message length between nodes.

Discussion

Use **crecv()** to initiate the receipt of a message, and then cause the calling process to wait for a message whose type matches the type(s) specified in the *typesel* parameter. When the message is received, it is stored in *buf* and the calling process resumes execution. You can use the **info...()** calls to get more information about the message after it is received.

This is a synchronous receive, causing the calling process to be blocked until the desired message is received. Use **irecv()** when you do not want the calling process to be blocked.

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

Errors

`crecv: No pid defined`

Use **setpid()** to define a host process id.

`crecv: Invalid length`

Use a non-negative number or a length less than or equal to maximum message length.

`crecv: Too many requests`

Use **msgwait()** for **isend()** or **irecv()** requests that are outstanding.

`crecv: Buffer length exceeds allocation`

Make sure parameter length does not exceed buffer size.

`crecv: Invalid buffer pointer`

Specify a pointer which contains the address of a valid data buffer (C only).

`crecv: Received message too long for buffer`

Make sure the buffer is large enough to hold the message.

See Also

csend(), csendrecv(), infocount(), infonode(), infopid(), infotype(), irecv(), isend()

CSEND() CSEND()

Send a message. Blocks until execution is complete.

Synopsis

csend(*type, buf, len, node, pid*)

Parameter Declarations

long *type;*
char **buf;*
long *len, node, pid;*

Return Value

None

Example

This example defines a simple message ("Hello node") and uses a **csend()** statement to send it to process 0 on node 0. Using the same message buffer, a new message is defined ("Hello host") and **csend()** is used again to send it to the host.

```
char *msg;
long len;
    .
    .
msg = "Hello node 0";
len = strlen(msg) + 1;          /* Length of string (including the null terminator) */

csend(5, msg, len, 0, 0);       /* Send type 5 message to process 0 on node 0 */
    .                                                  /* for RX nodes, pid must be 0*/
    .
msg = "hello host";                                    /* Re-use the message buffer */
    .
csend(2, msg, strlen(msg)+1, myhost(), 10);
                                /* Send new type 2 message to process 10 on host */
```

CSEND() *(cont.)*

CSEND() *(cont.)*

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

type an integer that identifies the type of the message that you are sending.

buf refers to the buffer that contains the message to be sent. The buffer may be any legal data type. It is recommended that data types match in send and receive operations.

len a positive integer that specifies the size (in bytes) of the message that you wish to send. Messages to or from a host process are limited to a maximum of 256K bytes. There is no limit on message length between nodes.

node an integer that defines where the message is to be sent.

pid an integer that specifies the id of the process that is to receive the message. For the RX node (which supports the single process model), *pid* must be set to `0`. Values other than `0` return an error. The CX node supports the multiple process mode, and *pid* can be any integer.

Discussion

Use **csend()** to send a message to a node or host process and cause the calling process to wait until the message is sent. Completion of **csend()** does not imply that the message was received by the destination process, only that the message was sent and that the buffer (*buf*) is available for reuse.

CSEND() *(cont.)*

CSEND() *(cont.)*

It is recommended that you use a *type* with a value from `0` to `999,999,999`. Types `1,073,741,824` to `1,999,999,999` are special force types that bypass the normal flow control mechanisms, do not match the `-1` wildcard, and are discarded if no receive is posted. Force types are intended to be used with **csendrecv()**, **isendrecv()**, and **hsendrecv()**, but can be used by other message passing calls.

NOTE

It is not recommended that you use types in other ranges because they are used by the system and could produce unpredictable results.

If *node* is a positive integer, the message is sent to the node with that number. Nodes within a partition are numbered consecutively from 0, and a node number higher than the highest node in the cube causes an error to be returned. To send a message to the host, use the host's id, which is returned by **myhost()**.

If *node* is a negative number up to a certain value, it causes a "global" send. If *node* is `-1`, the message is sent to all nodes. Other negative numbers cause the message to be sent to a subcube composed of a set of the nodes surrounding the sender. The dimension of this cube is calculated as (*d* - **nodedim()** `-1`), where *d* is the dimension of the subcube, and is less than or equal to **nodedim()**.

For an RX node, if a global send specifies its own process (**mypid()**) it does not receive its own message. For CX nodes, however, if a process other than its own is specified, the sending node also receives the message.

For the CX node, Valid *pids* include any integer value. Negative numbers are reserved for system programs. A node *pid* is assigned when you load a process and a host *pid* is assigned with **setpid()**. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on the RX node).

To initiate a send request when you want the calling process to continue during the send operation, use **isend()**. To define a *pid*, use the **setpid()** call.

CSEND() *(cont.)*

CSEND() *(cont.)*

Errors

`csend: No pid defined`

Use **setpid()** to define a host process id.

`csend: Invalid type`

Use a non-negative number.

`csend: Invalid length`

Use a non-negative number or a length less than or equal to maximum message length.

`csend: Invalid node`

Use **numnodes()** to determine cube size or **myhost()** to determine host number.

`csend: Invalid buffer pointer`

Specify a pointer which contains the address of a valid data buffer.

`csend: Not enough memory`

Make sure message buffer starts on a boundary which is a multiple of four or make more memory available to this process.

See Also

cprobe(), crecv(), csendrecv(), iprobe(), irecv(), isend()

CSENDRECV() CSENDRECV()

Send a message and receive a reply. The calling process is blocked until the reply is received.

Synopsis

long **csendrecv**(*type, sbuf, slen, tonode, topid, typesel, rbuf, rlen)*

Parameter Declarations

long *type, typesel;*
char **sbuf, *rbuf;*
long *slen, tonode, topid, rlen;*

Return Value

Length of the received message.

Example

The following example illustrates how to use **csendrecv()**. Part 1 runs on node 0 and uses **csendrecv()** to make a remote procedure call to node 1 and post a receive for the message containing the result. Part 2 runs on node 1, and receives the message; it performs the receive operation and returns the result to the first procedure.

Part 1 (runs on node 0):

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

#define SIN_REQUEST_TYPE 100
#define SIN_REPLY_TYPE 200
#define SERVER_NODE 1
#define SERVER_PID  0

double rsin();

long iam;
main()
{
double pi, x, angle;
```

CSENDRECV() *(cont.)*

CSENDRECV() *(cont.)*

```
   iam = mynode();
   pi = acos(-1.0);
   angle = pi/2.0;
   x = rsin(angle);
   printf("\nNode %d: x = %lf  for angle = %lf
radians\n",iam,x,angle);
}

double rsin(x)
double x;
{
double  result;
long    len;

   len = csendrecv(SIN_REQUEST_TYPE, &x, sizeof(x), SERVER_NODE,
                   SERVER_PID, SIN_REPLY_TYPE, &result,
                   sizeof(result));
   return result;
}
```

Part 2 (runs on node 1):

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

#define SIN_REQUEST_TYPE 100
#define SIN_REPLY_TYPE 200

main()
{
int req_node, req_pid;
double x, result;

   for(;;) {
      crecv(SIN_REQUEST_TYPE, &x, sizeof(x));
      req_node = infonode();
      req_pid = infopid();
      result = sin(x);

      csend(SIN_REPLY_TYPE, &result, sizeof(result),
            req_node, req_pid);
   }
```

CSENDRECV() *(cont.)*

CSENDRECV() *(cont.)*

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

type a non-negative integer that identifies the type of the message that you are sending.

sbuf refers to the buffer that contains the message to be sent.

slen a positive integer that specifies the size (in bytes) of the message that you wish to send. Messages to or from a host process are limited to a maximum of 256K bytes. There is no limit on message length between nodes.

tonode an integer that determines the node(s) to which the message is to be sent.

topid an integer that specifies the id of the process that is to receive the message. Valid *pids* for RX nodes include any integer value. Negative numbers are reserved for system programs. A node *pid* is assigned when you load a process and a host *pid* is assigned with **setpid()**. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on the RX node).

typesel is an integer value that specifies the type(s) of the reply message .

rbuf refers to the buffer where the reply message will be stored. The buffer can be any valid data type. It is recommended that data types match in send and receive operations.

rlen an integer value that specifies the size of the reply message buffer in bytes.

CSENDRECV() *(cont.)*

CSENDRECV() *(cont.)*

Discussion

Use **csendrecv()** as a synchronous call that simultaneously sends a message to a node or host process and posts a receive for the reply. The process is blocked from continuing until the return message is received. When a message whose type matches the type(s) specified in the *typesel* parameter arrives, the calling process receives the message, stores it in *rbuf* and the calling process resumes execution. You cannot use the **info...()** calls with this call; it does not affect the information.

This call is intended to be used for remote procedure calls. If you do not want to block the procedure while the send-receive operation is occurring, use **isendrecv()**.

The *typesel* parameter is a user-defined variable that identifies the kind of information contained in the message being sent. Types `0` to `999,999,999` are normal types that can be used by any send or receive call, including this one. Types `1,073,741,824` to `1,999,999,999` are special force types available to the user intended specifically for the "sendrecv" calls. Force types have special properties:

- A message with a force type bypasses the normal flow control mechanisms and is not delayed by clogged message buffers on the node.
- 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.

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.

The buffer for the send and receive may be any legal data type. It is recommended that data types match in send and receive operations.

If *tonode* is a positive integer, the message is sent to the node with that number. Nodes within a partition are numbered consecutively from 0, and a node number higher than the highest node in the cube causes an error to be returned. To send a message to the host, use the host's id, which is returned by **myhost()**.

CSENDRECV() *(cont.)*

CSENDRECV() *(cont.)*

If *tonode* is a negative number up to a certain value, it causes a "global" send. If node is `-1`, the message is sent to all nodes. Other negative numbers cause the message to be sent to a subcube composed of a set of the nodes surrounding the sender. The dimension of this cube is calculated as (*d* - **nodedim()** `-1`), where *d* is the dimension of the subcube, and is less than or equal to **nodedim()**.

The *typesel* parameter describes the type(s) of the message(s) expected to be received. 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. You can use a force type, as described above.
- If *typesel* is `-1`, the first message to arrive for the process that initiated the receive operation will be recognized. After the first message has been received, you can use `-1` again to receive the next message.
- If *typesel* is any negative number other than `-1`, a set of message types will be recognized. Refer to Appendix B for information on building a *typesel* mask.

Errors

`csendrecv: No pid defined`

Use **setpid()** to define a host process id.

`csendrecv: Invalid type`

Use a non-negative number.

`csendrecv: Invalid length`

Use a non-negative number or a length less than or equal to maximum message length.

`csendrecv: Invalid node`

Use **numnodes()** to determine cube size or **myhost()** to determine host number.

CSENDRECV() *(cont.)*

CSENDRECV() *(cont.)*

`csendrecv: Buffer length exceeds allocation`

Make sure parameter length does not exceed buffer size.

`csendrecv: Invalid buffer pointer`

Specify a pointer which contains the address of a valid data buffer.

`csendrecv: Not enough memory`

Make sure message buffer starts on a boundary that is a multiple of four or make more memory available to this process.

`csendrecv: Received message too long for buffer`

Make sure buffer is large enough to hold the message.

See Also

cprobe(), crecv(), hsendrecv(), iprobe(), irecv(), isend(), isendrecv()

CTOH*x*(), HTOC*x*() CTOH*x*(),HTOC*x*()

Convert node byte order to remote workstation byte order, or remote workstation byte order to node byte order.

Synopsis

CTOHC(*sv*,*n*) byte swap cube to host **char** within a structure
HTOCC(*sv*,*n*) byte swap host to cube **char** within a structure

CTOHD(*sv*,*n*) byte swap cube to host **double**
HTOCD(*sv*,*n*) byte swap host to cube **double**

CTOHF(*sv*,*n*) byte swap cube to host **float**
HTOCF(*sv*,*n*) byte swap host to cube **float**

CTOHL(*sv*,*n*) byte swap cube to host **long**
HTOCL(*sv*,*n*) byte swap host to cube **long**

CTOHS(*sv*,*n*) byte swap cube to host **short**
HTOCS(*sv*,*n*) byte swap host to cube **short**

Parameter Declarations

CTOHC() and **HTOCC()**:	**unsigned char** **sv*; **long** *n*;
CTOHD() and **HTOCD()**:	**double** **sv*; **long** *n*;
CTOHF() and **HTOCF()**:	**float** **sv*; **long** *n*;
CTOHL() and **HTOCL()**:	**long** **sv*; **long** *n*;
CTOHS() and **HTOCS()**:	**short** **sv*; **long** *n*;

Return Value

None

CTOH*x*(), HTOC*x*() *(cont.)*

CTOH*x*(), HTOC*x*() *(cont.)*

NOTE

To swap bytes of all the fields in a structure in a C program, use these calls and the **createstruc()** and **relstruc()** calls. Swapping bytes for values of type **char** is required only within a structure.

Example

The following example is a fragment of a host program that sends a message (of type *NODE_TYPE* and consisting of the array *datain*) from the host to node 0, and then receives a message back (the same type, but consisting of the array *dataout*). Sometimes the byte order in which data is stored on the remote host is different than the byte order of data on the nodes. In these cases, the **HTOCL()** call changes the byte order of *datain* before it goes to the nodes, and **CTOHL()** changes the byte order back to the remote host's byte order after the message from the nodes is received.

```
    .
    .
long datain[100];
long dataout[100];
    .
    .
HTOCL(datain,100);                    /* convert datain to node byte order */

csend(NODE_TYPE, datain, sizeof(datain), 0, NODE_PID);
    .
    .
crecv(NODE_TYPE, dataout, sizeof(dataout));
CTOHL(dataout,100);                   /* convert dataout to remote host byte order */
    .
    .
```

Environment

Host

Languages

C, Fortran

CTOH*x*(), HTOC*x*() *(cont.)*

CTOH*x*(), HTOC*x*() *(cont.)*

Description of Parameters

sv starting address of variable(s) to swap . Individual variables of any type, elements of structures, or arrays can be used as long as the address is specified. When you want to swap an entire array, use *n* for the number of elements and *sv* as the array starting address.

n number of *sv*'s to swap. If you want to swap multiple elements that are the same type and are also contiguous in memory, use *n* to specify the number of elements to swap. Otherwise, specify `1`.

Discussion

Remote development machines that do not use a 286™ or 386™ microprocessor as its main processor may have a different internal byte order representation and byte alignment of integer and floating point values than the nodes. If so, you need to use these utilities in host programs whenever you send or receive data between node and host programs. These utilities are provided to convert byte order between the node and the host. These calls must be used in the host program, because the node expects to receive that data in the proper byte order.

Therefore, before sending data from remote host to node with one of the send operations (**csend()** or **isend()**), use one of the **HTOC*x*** calls depending on the data type, and after receiving data from the node program via **crecv()** or **irecv()**, change the byte order back to the remote host byte order with one of the **CTOH*x*** calls.

The byte-swapping utilities are provided for source code compatibility, and the utilities have no effect on programs running on the SRM and remote development machines that do not require byte-order change.

Errors

None

See Also

csend(), **crecv()**, **irecv()**, **isend()**
For C programs only, **createstruc()**

CUBEINFO() CUBEINFO()

Obtain information about allocated cubes. This call is for use by host programs; it is not available for node programs.

Synopsis

long **cubeinfo**(*ct, numslots, global*)

Parameter Declarations

struct cubetable **ct;*
long *numslots;*
long *global;*

Return Value

An integer containing the number of cubes for which information is available. If the number of cubes allocated was greater than the value you specify for *numslots*, this value will be larger than *numslots*. However, the number of cubes for which information is returned to the buffer table never exceeds *numslots*.

Example

The following fragment is an example of how you might use the **cubeinfo()** call in a program. In this example, the number of slots is the maximum number of cubes allowed in a system (10). The **cubeinfo()** call is called with the *global* parameter set to 2, to get information about all allocated cubes on the current System Resource Manager (SRM). This returns the number of cubes found (storing it in the variable *numcubes*) and also fills in the table *ctbl* with the cube information for each of the cubes. It then does a string compare to make sure that the *cubename* of one of the allocated cubes is the one desired (in this case, the cube's name is "alpha").

```
main()
{
    struct cubetable ctbl[10];
    struct cubetable *ct = ctbl;
    int numslots = 10;
    int numcubes, i;
        .
        .
```

CUBEINFO() *(cont.)* CUBEINFO() *(cont.)*

```
numcubes = cubeinfo(ctbl, numslots, 2);
for (i=0;i<numcubes; i++)
   if (!strcmp(ctbl[i].cubename, "alpha"))
        .
        .
```

Notes

When you execute **cubeinfo()**, the information is stored in a structure named *cubetable*, which is defined in */usr/include/cube.h*. The *ct* parameter is a pointer to this structure.

```
#define NAMELEN 16
#define NUMTTYS 4

struct cubetable {
    char cubename[NAMELEN];                          /* Name of cube*/
    char username[NAMELEN];                          /* Name of user*/
    char srmname[NAMELEN];          /* Name of SRM to which cube is attached*/
    char hostname[NAMELEN];         /* Name of host where getcube was invoked,*/
                                    /* -either the remote system name or srmname*/
    char tty[NAMELEN];                        /* tty where getcube was invoked*/
    char cubetype [NAMELEN];                         /* Type of this cube*/
    char cindex [NAMELEN];                  /* Char of index into cubetable*/
    char ttys[NUMTTYS][NAMELEN];                 /* Used for attached ttys.*/
}
```

A user logged into the same host on different *ttys* may be attached to the same cube from each *tty*. Up to four *ttys* may be attached to the same cube. Strings for attached *ttys* are of format `0` for no *tty* and a string containing the full pathname on the system for valid *ttys*.

Environment

Host

Languages

C, Fortran

CUBEINFO() *(cont.)* CUBEINFO() *(cont.)*

Description of Parameters

ct specifies a buffer table where cube information is to be returned.

numslots is the number of elements in the table to which cube information is returned. Each element contains the cube information for a single cube.

global is an integer value from `0` through `4` that determines the cubes for which information is returned. The possible settings are:

`0` returns information about the current attached cube.

`1` returns information about all cubes that you own that were allocated from the current host

`2` returns information about all the cubes on the system from which the command was executed. If executed on an SRM, information is returned for all cubes allocated from that SRM; if executed on a remote development machine, supplies information on all cubes that have been allocated from that workstation is returned.

`3` returns information about how cubes are allocated on all SRMs. This is equivalent to setting *global* to `2` on each SRM on the network. This value is not allowed on the SRM.

`4` requires that the call have the additional parameter *srmname*. This call should only be used in a program to be run on a remote host, and allows you to specify the name of a particular SRM in a network. The call becomes:

cubeinfo(*ct, numslots, global, srmname*)

where the parameter *srmname* in C is a pointer to a character string, and in Fortran is a character string, naming the SRM of interest.

CUBEINFO() *(cont.)* CUBEINFO() *(cont.)*

Discussion

Use **cubeinfo()** to obtain information about allocated cubes. The return value is the number of cubes matching the *global* parameter. In addition, **cubeinfo()** fills a slot in a buffer with an array of information about each of these cubes. The parameter *numslots* limits the number of cubes for which information will be placed in a slot. If there are fewer selected cubes than *numslots*, the remaining slots will be cleared to contain all zeros. If there are no cubes found, **cubeinfo()** returns a `0` and the buffer is cleared.

Errors

`cubeinfo: Commser not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with the **bootcube** or **rebootcube** command.

`cubeinfo: Lifeline not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with the **bootcube** or **rebootcube** command.

`cubeinfo: Global value invalid`

Use a value of `0`, `1`, `2`, `3`, or `4` for global.

See Also

attachcube(), **getcube()**, **relcube()**

CWRITE() CWRITE()

High speed synchronous write to a CFS file. Blocks the calling process until the write is complete.

Synopsis

cwrite(*fildes, buf, len*)

Parameter Declarations

long *fildes;*
char **buf;*
long *len;*

Return Value

None

Example

The following example uses **cwrite**() to write the contents of a predefined buffer to a CFS file named *mydata*. The example first creates the file with the **open**() call and stores the file descriptor in the variable *fd*. Then, after the buffer is set up, **cwrite**() writes the data in the buffer into the file:

```
        .
        .                                   /* Create a file on CFS named mydata with open() */

fd = open("/cfs/mydata", O_CREAT|O_WRONLY,0644);
                                            /* Set up a buffer named "buffer" with data */
        .
        .
cwrite(fd, buffer, sizeof(buffer));         /* Write data in buffer to file */
        .
        .
```

CWRITE() *(cont.)*

CWRITE() *(cont.)*

Environment

Node

Languages

C, Fortran

Description of Parameters

fildes	the file descriptor.
buf	refers to the buffer where the data to be written is stored. The buffer can be any valid data type.
len	an integer value that specifies the buffer size in bytes. Transfer size is limited only by memory available for the buffer.

Discussion

The **cwrite()** call writes the contents of the named buffer to the file specified by the *fildes* parameter (the value returned by the **open(), creat(),** or **dup()** UNIX-compatible system calls). The **cwrite()** call is a synchronous call that prevents the process from executing further instructions until the write is complete. If you do not want the calling process blocked during the write, use the **iwrite()** call.

CWRITE() *(cont.)*

CWRITE() *(cont.)*

Errors

`cwrite: Invalid argument`

Check arguments.

`cwrite: Mixed file operations`

In I/O mode 2 or 3, nodes are attempting different operations (reads and writes) to a shared file. Ensure that nodes are performing the same operation.

`cwrite: I/O error`

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

`cwrite: Not enough core`

Not enough memory on node for operation.

`cwrite: No space left on device`

Not enough space on device to which you are writing. Create more space in file system, or if you have restricted the disks to which you are writing with **restrictvol()**, re-execute **restrictvol()** to include more drives.

See Also

cread(), **iread()**, **iwrite()**
UNIX System V/386 Programmer's Reference: **open()**, **creat()**, **dup()**, **write()**

DCLOCK DCLOCK()

Returns double precision time in seconds since booting the cube.

Synopsis

double **dclock**( )

Parameter Declarations

None

Return Value

Double precision time (in seconds)

Example

The following example illustrates the proper use of **dclock**().

```
double start_time, end_time, elapse_time;
    .
    .
start_time = dclock();                          /* find starting time */
                                                /* run calculations */
    .
    .
end_time = dclock();                            /* find ending time */
elapse_time = end_time - start_time;            /* find elapsed time */
```

Environment

Node

Languages

C, Fortran

DCLOCK() *(cont.)*

DCLOCK() *(cont.)*

Discussion

The **dclock()** subroutine returns double precision time in seconds since the cube was booted. In this form, there is enough precision to maintain 100ns accuracy (which is the count rate for the RX node board counter) for a period of approximately 14 years. On the CX node, the maximum precision will be 1ms.

Errors

None

See Also

hwclock(), mclock()

EADD(), ECMP(), EDIV(), EMOD(), EMUL(), ESUB()

Performs mathematical operations on extended file sizes.

Synopsis

esize_t **eadd**(*e1*, *e2*)
esize_t *e1;*
esize_t *e2;*

long **ecmp**(*e1*, *e2*)
esize_t *e1*
esize_t *e2*

long **ediv**(*e*, *n*)
esize_t *e;*
long *n;*

long **emod**(*e*, *n*)
esize_t *e;*
long *n;*

esize_t **emul**(*e*, *n*)
esize_t *e;*
long *n;*

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

EADD(), ECMP(), EDIV(), EMOD(), EMUL(), ESUB() *(cont.)*

Return Values

ecmp() returns:

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

All others return the computed value of type *esize_t* as defined in *cube.h*.

Example

The following example shows a few typical extended math operations.

```
#include <estat.h>
      .
      .
      .
int r, q;
esize_t e1, e2;

e1.slow = 0;
e1.shigh = 2;                          /* e1 is 8G bytes */

e2.slow = 1;
e2.shigh = 0;
e1 = eadd(e1, e2);                     /* increment e1 by 1 */
e1 = esub(e1, e2);                     /* decrement e1 by 1 */
e1 = emul(e1, 2);                      /* multiply e1 by 2 */
e1 = emul(e1, 2);                      /* divide e1 by 2 */
r = emod(e1, 100);                     /* remainder */
```

EADD(), ECMP(), EDIV(), EMOD(), EMUL(), ESUB() *(cont.)*

This example compares two extended numbers *e1* and *e2*.

```
#include <estat.h>
      .
      .
      .
esize_t e1, e2;
e1.slow = 3;
e1.shigh = 4;
e2.slow = 0;
e2.shigh = 2;
if (ecmp(e1,e2) > 0)
    printf ("e1 is greater than e2 \n");
```

Environment

Node

Languages

C, Fortran

Discussion

The following calls perform mathematical functions on 64-bit integers. These calls support the extended file sizes in CFS.

eadd()	Adds two extended numbers
ecmp()	Compares two extended numbers
ediv()	Divides extended number by integer
emod()	Gets remainder of **ediv()**
emul()	Multiplies extended number with integer
esub()	Subtracts two extended numbers

EADD(), ECMP(), EDIV(), EMOD(), EMUL(), ESUB() *(cont.)*

Description of Parameters

e1, e2	extended number values
n	integer value

Errors

`ediv: Invalid size`

Quotient should fit in a long integer.

`emod: Invalid size`

Quotient should fit in a long integer.

See Also

etos(), stoe()

ESEEK() ESEEK()

Move file pointer in extended files.

Synopsis

esize_t **eseek**(*fildes, offset, whence*)

Parameter Declarations

long *fildes;*
esize_t *offset;*
long *whence;*

Return Values

The resulting file pointer location is returned, as measured in bytes from the beginning of the file.

Example

This example uses **eseek()** to move the file pointer to the beginning of a file.

```
#include <estat.h>
          .
          .
esize_t offset, newpos;
long fd;
          .
          .
offset.slow = 0;
offset.shigh = 0;
          .
          .
newpos = eseek(fd, offset, 0);
```

ESEEK() *(cont.)*

ESEEK() *(cont.)*

Environment

Node

Languages

C, Fortran

Description of Parameters

fildes the file descriptor returned by the **open()**, **creat()**, or **dup()** UNIX system calls.

offset a 64-bit value that affects the current size of the file depending upon the value of *whence*.

whence an integer value of `0`, `1`, or `2` that determines how the offset value is used to set the size of the file. These values have the following effects:

`0`	sets the file pointer to offset.
`1`	sets the file pointer to the current location of the file pointer plus offset.
`2`	sets the file pointer to the current size plus offset.

Discussion

Use **eseek()** to move the read/write file pointer in the file designated by the *fildes* parameter, which should be the value returned by **open()**, **creat()**, or **dup()** system call. The *offset* parameter is a value in bytes that depends upon the value of the *whence* parameter. If *whence* is `0`, the pointer is set to *offset* bytes from the beginning of the file. If *whence* is `1`, the pointer is set to the *offset* added to the current location of the file pointer. If *whence* is `2`, the pointer is set to the size of the file plus *offset* (the units are bytes). If successful, the location of the new file pointer is returned in units of bytes.

This call supports the extended file sizes and only operates on CFS files.

ESEEK() *(cont.)*

ESEEK() *(cont.)*

Errors

`eseek: Bad file number`

Use the unit number assigned by **open()**, **creat()**, or **dup()**.

`eseek: Mixed file operations`

In I/O mode 2 or 3, nodes are attempting different operations (reads and writes) to a shared file. Ensure that nodes are performing the same operation.

`eseek: Not implemented on the SRM`

Available only for CFS files.

See Also

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

ESIZE() ESIZE()

Increase the size of a large file.

Synopsis

esize_t **esize**(*fildes, offset, whence*)

Parameter Declarations

long *fildes;*
esize_t *offset;*
long *whence;*

Return Values

The file size in bytes is returned if the call is successful; otherwise, it returns a `-1`.

Example

The following example increases the size of the file designated by the file descriptor variable *fd* (the value returned by the UNIX **open()**, **creat()**, or **dup()** call) to 4G bytes.

```
#include <estat.h>
      .
      .
esize_t offset, newsize;
long fd;
offset.slow = 0;
offset.shigh = 1;
newsize = esize(fd, offset, 0);          /* increase file size to 4 gigabyte */
```

Environment

Node

Languages

C, Fortran

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

Description of Parameters

fildes the file descriptor returned by the **open()**, **creat()**, or **dup()** UNIX system call.

offset a 64-bit value that affects the current size of the file depending upon the value of *whence*.

whence an integer value of `0`, `1`, or `2` that determines how the offset value is used to set the size of the file. These values have the following effects:

`0`	sets the file size to the greater of the current size or *offset*.
`1`	sets the file size to the greater of the current size or the current location of the file pointer plus *offset*.
`2`	sets the file size to the greater of the current size or the current size plus *offset*.

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

Discussion

The **esize()** call changes the size of the file designated by the *fildes* parameter, which must be the value returned by the **open()**, **dup()**, or **creat()** call. The *offset* parameter is a value in bytes that, depending upon the value of the *whence* parameter, is used to determine the resulting size of the file.

A major use of **esize()** is to ensure that enough space is allocated before you do any sensitive calculations. If *offset* is positive, **esize()** ensures that the disk blocks are allocated to the file from the starting position to the ending position (as determined by *whence* and *offset*). If allocation has been restricted by **restrictvol()**, any new disk blocks allocated will come from the restricted set.

If the call is successful, the call returns the new size of the file in bytes. If the call is not successful, the call will fail and return a `-1`. Using **esize()** does not affect the position of the file pointer.

ESIZE() *(cont.)*

ESIZE() *(cont.)*

Errors

`esize: Bad file number`

Use the file descriptor returned by **open()**, **creat()**, or **dup()**.

`esize: No space left on device`

Not enough space on device to which you are writing. Create more space in file system, or if you have restricted the disks to which you are writing with **restrictvol()**, re-execute **restrictvol()** to include more drives.

`esize: Not implemented on the SRM`

Available only for CFS files.

See Also

lsize(), **restrictvol()**, **eseek()**
UNIX System V/386 Programmer's Reference: **open()**, **creat()**, **dup()**

ESTAT(), FESTAT() ESTAT(),FESTAT()

Get extended file status.

Synopsis

long **estat**(*path, buf*)
long **festat**(*fildes, buf*)

Parameter Declarations

char **path;*
struct estat **buf;*

long *fildes;*
struct estat **buf;*

Return Values

Upon successful completion, a value of `0` returns. Otherwise, a value of `-1` returns.

Example

This example uses **estat()** to retrieve the status of the file */cfs/workfile*. If successful, the program continues; otherwise, it exits.

```
#include <estat.h>
          .
          .
struct estat *stbuf;
          .
          .
if (estat ("/cfs/workfile", stbuf) == -1) {
      perror("estat");
      exit(1);
}
          .
          .
```

ESTAT(), FESTAT() *(cont.)*

ESTAT(), FESTAT() *(cont.)*

Environment

Node

Languages

C

Description of Parameters

fildes	the file descriptor returned by **open()**, **creat()**, or **dup()**.
path	points to a pathname identifying a file.
buf	points to an *estat* structure where the program places information concerning the file.

Discussion

The **estat()** call obtains information about the named file. **festat()** obtains information about an open file recognized by the file descriptor *fildes*, obtained from a successful **open()**, **creat()**, or **dup()** system call. The *estat* structure is the same as *stat* structure except the file size which is of the type *esize_t*.

Errors

`festat: Bad file number`

Use the file descriptor returned by the **open()**, **creat()**, or **dup()** system call.

`festat: Not implemented on the SRM`

Not supported on the System Resource Manager (SRM).

`festat: Invalid size`

Invalid size for the file.

ESTAT(), FESTAT() *(cont.)*

ESTAT(), FESTAT() *(cont.)*

See Also

eseek(), esize()
UNIX System V/386 Programmer's Reference: **stat(), fstat()**

ETOS(), STOE() ETOS(),STOE()

Converts between extended numbers and strings.

Synopsis

etos(*e, s*)
esize_t **stoe**(*s*)

Parameter Declarations

esize_t *e;*
char **s;*

Return Values

stoe() returns an extended number.

Example

The following example displays a 5G-byte value:

```
#include <estat.h>
          .
          .
esize_t e;
char s[20];
e.slow = 1073741824;
e.shigh = 1;
          .
          .
etos(e, s);
printf("5G bytes %s\n", s);      /* 5368709120 is displayed for the value of s */
```

ETOS(), STOE() *(cont.)*

ETOS(), STOE() *(cont.)*

The following example converts the string "5368709120" to a 64-bit number. The high portion of *res* receives `1`, which represents 4G bytes, while the low portion of *res* receives 1G byte; that is, *res.slow* equals `1073741824` and *res.shigh* equals `1`.

```
#include <estat.h>
          .
          .
esize_t res;
res = stoe("5368709120");
```

Environment

Node

Languages

C, Fortran

Description of Parameters

e	extended number.
s	character string.

Discussion

The **etos()** call converts an extended number to a string of characters. The **stoe()** call converts a string of characters to an extended number. These calls support the extended file sizes in CFS.

Errors

`etos: Invalid size`

Size should be less than 2^{60} or overflow occurs.

See Also

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

FLICK() FLICK()

Relinquish node processor to another process.

Synopsis

flick()

Parameter Declarations

None

Return Value

None

Example

The following example illustrates the proper use of **flick()**. Two asynchronous receives are posted, each of which return a message id (*id1* and *id2*). After some other processing, you enter a for-loop that tests to see if either message has been received, and if so, process the one received. If not, however, you use **flick()** to defer execution of this process, to allow another process on the same node to execute while waiting for these messages to be received.

```
integer node, pid, id1, id2;
integer buf1(2), buf2(10);
id1 = irecv(101, buf1, sizeof(buf1));
id2 = irecv(102, buf2, sizeof(buf2));

                                    /* Do some other processing here. */
    .
    .
```

FLICK() *(cont.)*

FLICK() *(cont.)*

```
for (;;) {                                  /* Now wait for either receive to complete. */
    if (msgdone(id1)) {
    .
    .                                                    /* Process message 1. */
    .
        break;
    } else if (msgdone(id2)) {
        .
        .                                                /* Process message 2. */
        .
        break;
    } else {
        flick();
    }
}
```

Environment

Host, Node

Languages

C, Fortran

Discussion

You should use the **flick()** routine when the current process has no useful work to do (e.g., when a process is waiting for any of several **irecv()**'s to complete). It is recommended that a process never "busy loop" without flicking. The **flick()** call is a delay mechanism, not a synchronization mechanism. The scheduler does not know the conditions for which a process is waiting.

CX Node Description

The **flick()** call temporarily relinquishes the node processor to another process on the same node. The scheduler runs processes on a node in round-robin fashion, running each process until it blocks in a **csend()**, **crecv()**, **cprobe()**, or **msgwait()**, or for approximately 50 milliseconds, or until the process gives up the processor with the **flick()** routine. The execution of the process is not deferred indefinitely. Instead, the process will resume execution during its next scheduled time frame. The **flick()** call simply allows other node processes to execute. The **csend()**, **crecv()**, **cprobe()**, and **msgwait()** operations also defer execution of a calling process until they complete their task.

FLICK() *(cont.)*

FLICK() *(cont.)*

RX Node Description

The **flick**() call relinquishes the node processor until an interrupt of any kind occurs within NX. It is then scheduled again for processing. The execution of the process can be deferred up to a maximum of 10 ms. Using **flick**() in this way can provide higher performance, because part of the interrupt overhead is eliminated.

Host Description

On a host machine, **flick**() performs no operation. It is included so that you can write programs that call **flick**() and still be able to run them on a host machine.

Errors

None

See Also

UNIX System V Programmer's Reference Manual: **sleep**()

FLUSHMSG() FLUSHMSG()

Flush specified messages from the system.

Synopsis

flushmsg(*typesel, node, pid*)

Parameter Declarations

long *typesel, node, pid;*

Return Value

None

Example

This example sends two synchronous messages followed by one asynchronous message to process 0 on node 1. It then flushes all messages of all types waiting to be received for that process and node. Issuing **msgcancel()** to cancel the asynchronous message (*msg3*) before the **flushmsg()** call ensures that all messages are canceled.

```
char *msg1, *msg2, *msg3;
long node, pid, msg_id;                          /* for RX nodes, pid must be 0 */
    .
    .
node = 1;
pid = 0;

csend(10, msg1, sizeof(msg1), node, pid);          /* Blocking send */
csend(11, msg2, sizeof(msg2), node, pid);          /* Blocking send */
msg_id = isend(12, msg3, sizeof(msg3), node, pid);  /* Async send */
    .
    .
msgcancel(msg_id);                                  /* Cancel msg3 */
flushmsg (-1, node, pid);                        /* Now msg3 will be flushed */
            /* If msg1 and msg2 have not been picked up, they*/
                                                    /* will be flushed also*/
```

FLUSHMSG() *(cont.)*

FLUSHMSG() *(cont.)*

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

typesel is an integer value that specifies the type(s) of message that you want to flush. The type depends on the type specified in the send. 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, messages of the specified type will be flushed. All other messages will be ignored.
- If *typesel* is `-1`, all messages to the node process will be flushed.
- If *typesel* is any negative number other than `-1`, a set of message types will be flushed. Refer to Appendix B for information on how to build a *typesel* mask.

node is a positive integer that identifies the receiving node or `-1` to indicate all nodes and the host. Nodes within a partition are numbered consecutively from 0. If you want to flush messages just from the host, use **myhost()** to determine the host's id.

pid for CX nodes, *pid* is an integer that specifies the receiving process id or `-1` to indicate all processes. Valid *pids* include any integer value, but negative numbers other than `-1` are reserved for system programs. A node *pid* is assigned when you load a process and a host *pid* is assigned with **setpid()**. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on the RX node).

FLUSHMSG() *(cont.)*

FLUSHMSG() *(cont.)*

Discussion

Use **flushmsg()** to flush all messages of the specified type from system buffers. All messages of the type(s) selected by *typesel* in the selected node(s) that have been sent to the selected *pid(s)* will be eliminated from the system.

Use of **flushmsg()** has no effect on processes. It also has no effect on messages in transit to processes which do not have the selected *pid*, even if they were sent by a process which does have a matching *pid*. That is, **flushmsg()** flushes messages by the destination *pid*, not the source *pid*. If a particular node is selected, only messages sent to that node which have arrived on that node but have not been received are affected. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on the RX node).

To ensure that all messages are flushed, use **flushmsg()** in conjunction with **msgcancel()** to ensure that all specified messages are flushed.

Errors

```
flushmsg: invalid node
```

Use **numnodes()** to determine cube size and **myhost()** to determine host number.

See Also

msgcancel(), **killcube()**

GCOL() GCOL()

Global concatenation operation.

Synopsis

gcol(*x, xlen, y, ylen, ncnt*)

Parameter Declaration

char *x[];*
long *xlen;*
char *y[];*
long *ylen;*
long **ncnt;*

Return Values

None

Example

This example assumes that a vector of length *n* is distributed among the nodes of the cube, where *x*(*m*) is a piece of the vector on a given node. Note that *m* need not have the same value on all nodes. After computing the dot product and summing the global contributions, the vector must be normalized and then the entire vector must be available to all nodes, using **gcol()** to collect the vector. The array *y* in each node then contains all of the vector *x*. The order of *x*'s in *y* is by increasing node number. This example is simplified, and the way in which the norm is computed in this example can lead to problems with numerical underflow or overflow. Safer and more complicated implementations are possible and ought to be used in practice.

GCOL() *(cont.)*

GCOL() *(cont.)*

```
int i, cnt;
double x[m], y[n], dot, norm, dummy;
int dpsize = 8;
                                    /* Assume that x, m, and n have been initialized */
                                                    /* Compute the local dot product */
dot = 0;
for (i = 0; i<m; i++)
    dot = dot + x[i]*x[i];
                                                              /* Sum global contributions */
gdsum (&dot, 1, &dummy);
                                                                   /* Compute global norm */

norm = sqrt(dot);
                                                              /* Do the local normalization */
for (i=0; i<m; i++)
    x[i] = x[i]/norm;
                                                                        /* Collect vector */
gcol(x, m*dpsize, y, n*dpsize, &cnt);
```

Environment

Node

Languages

C, Fortran

Description of Parameters

x	refers to the input buffer to be used in the operation. Note that *x* may be of any type.
xlen	the length of input buffer *x* (in bytes).
y	refers to the output buffer to be used in the operation (*y* contains the desired result). Note that *y* must be of the same data type as *x*.
ylen	the length of output buffer *y* (in bytes).
ncnt	refers to the number of bytes returned in *y*.

GCOL() *(cont.)*

GCOL() *(cont.)*

Discussion

Use **gcol()** to collect and concatenate a contribution from each node in the current cube and to ensure that contributions are received in order. The *x* and *y* parameters can be of any data type as long as they are of the same data type. The result is returned in *y* to every node. All participating processes must have the same *pid*, and for RX, the process ID must 0. This is a global operation, meaning that all nodes in the cube must issue this command before the execution of the process can continue.

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

Errors

`gcol: Received message too long for buffer`

Make sure buffer is long enough.

`gcol: Invalid buffer pointer`

Specify a pointer which contains the address of a valid buffer.

`gcol: Invalid length`

Use a non-negative number for *xlen* or *ylen*.

`gcol: Buffer length exceeds allocation`

Make sure buffers are large enough and length parameters are not greater than buffer size.

GCOLX() GCOLX()

Global concatenation operation for contributions of known length.

Synopsis

gcolx(*x, xlens, y*)

Parameter Declaration

char *x[];*
long *xlens[];*
char *y[];*

Return Values

None

Example

This example assumes that a vector of length *n* is distributed among the 16 nodes of the cube, where *x(m)* is a piece of the vector on a given node. Note that *m* need not have the same value on all nodes (but for this example, we assume that it does). After computing the dot product and summing the global contributions, the vector must be normalized and then the entire vector must be available to all nodes, using **gcolx()** to collect the vector. The contents of the *xlens* array used by **gcolx()** is easy to calculate, since we assume *x* to be the same length on every node. The array *y* in each node then contains all of the vector *x*. The number of bytes returned in *y* is simply the sum of the elements in *xlens*. The order of *x*'s in *y* is by increasing node number. This example is simplified, and the way in which the norm is computed in this example can lead to problems with numerical underflow or overflow. Safer and more complicated implementations are possible and ought to be used in practice.

GCOLX() *(cont.)*

GCOLX() *(cont.)*

```
int i;
double x[m], y[n], dot, norm, dummy;
int dpsize = 8;
int num_nodes = 16;
long xlens[num_nodes],xlen;
                                    /* Assume that x, m, and n have been initialized */

                                    /* Compute the local DOT product */
dot = 0;
for (i = 0; i<m; i++)
    dot = dot + x[i]*x[i];

                                    /* Sum global contributions */
gdsum (&dot, 1, &dummy);
                                    /* Compute global NORM */

norm = sqrt(dot);
                                    /* Do the local normalization */
for (i=0; i<m; i++)
    x[i] = x[i]/norm;

xlen = m*dpsize;
for (i=0; i < num_nodes; i++)
    xlens[i] = xlen;                /* get the length of all contributions into xlens */

gcolx(x, xlens, y);                 /* Collect vector */
```

Environment

Node

Languages

C, Fortran

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

Description of Parameters

x	refers to the input buffer to be used in the operation. Note that *x* may be of any type.
xlens	an array containing the length (in bytes) of the input buffer *x* expected on each node.
y	refers to the output buffer to be used in the operation (*y* contains the desired result). Note that *y* must be of the same data type as *x*.

Discussion

Use **gcolx()** to globally collect and concatenate a contribution of specified length from each node in node number order. By providing the expected length of each contribution, the speed of this operation is greatly improved due to the reduced overhead of calculating where each contribution belongs in the output buffer. The position of each element in *xlens* corresponds to the node number from which the *x* parameter of the given length is expected. The *x* and *y* parameters can be of any data type as long as they are of the same data type. The result is returned in *y* to every node. All participating processes must have the same *pid*. This is a global operation, meaning that all nodes in the cube must issue this command before the execution of the process can continue.

If the lengths of the contributions from all the nodes is unknown, use the **gcol()** routine.

Errors

`gcolx: Received message too long for buffer`

Make sure buffer is long enough.

`gcolx: Invalid buffer pointer`

Specify a pointer which contains the address of a valid buffer.

`gcolx: Invalid length`

Use a non-negative number for *xlens*.

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

```
gcolx: Buffer length exceeds allocation
```

Make sure buffers are large enough and length parameters are not greater than buffer size.

See Also

gcol()

GETCUBE() GETCUBE()

Allocate a cube. This call is intended for host programs and is not available for node programs.

Synopsis

getcube(*cubename, cubetype, srmname, keep, account*)

Parameter Declarations

char **cubename;*
char **cubetype;*
char **srmname;*
long *keep;*
char **account;*

Return Value

None

Example

This example gets a cube named "alpha." This cube is to be made up of 16 nodes of type vector (vx). By assigning `NULL` to the *srmname* field, the name defaults to the name of the System Resource Manager. A *keep* value of 0 releases the cube and terminates the node processes when the calling process exits or is killed.

```
getcube ("alpha", "16vx", NULL, 0, 0);
```

Environment

Host

Languages

C, Fortran

GETCUBE() *(cont.)*

GETCUBE() *(cont.)*

Description of Parameters

cubename specifies the name of the cube that you want to allocate. Valid names include any ASCII string, 15 characters or less. Names longer than 15 characters will be truncated to 15 characters. If the name is a null string, "defaultname" is used. It is an error to specify a name that you (or anyone using your login name) are already using. Use **cubeinfo()** or the **cubeinfo** command to find out the names of allocated cubes.

cubetype specifies the size and type of the cube you want to allocate. Specify size first followed by type in a contiguous string (no spaces). Without this option, you get the largest available cube. The number of nodes allocated will always be a power of two. If the size you specify is not a power of two, it will be rounded up.

Size can be specified in one of two ways:

n	where *n* is the number of nodes (for example, 8) or
d*n*	where **d** indicates dimension and *n* is the size of the dimension (for example, d3)

Specifying no type gets a cube of any type. Other choices are:

rx	i860-based compute nodes
cx	386-based compute nodes
vx	nodes with vector companion board
sx	nodes with SX installed
f	nodes with 387 coprocessor installed
b	disable message buffering
m1	nodes with at least 1M byte of memory each
m4	nodes with at least 4M bytes of memory each
m8	nodes with at least 8M bytes of memory each
m16	nodes with at least 16M bytes of memory each
m32	nodes with at least 32M bytes of memory each

GETCUBE() *(cont.)*

GETCUBE() *(cont.)*

	m64	nodes with 64M bytes of memory each
	n*s*	gets a continuous set of nodes starting with slot *s*

srmname specifies the name of the host connected to the cube that you want to allocate. If you are on a System Resource Manager, the name defaults to your System Resource Manager's name. Other *srmnames* are ignored. If you are on a remote host, specify the *srmname* that you want the cube to be allocated from. If *srmname* is not specified on a remote host, **getcube()** will select an available System Resource Manager for you from the list in the */usr/ipsc/lib/srms* file. The *srmname* must be the network host name of a System Resource Manager with an attached cube to be valid.

keep is one of two settings that allow you to choose either to keep the cube only until the process is terminated or until you release it. The possible settings are:

`0`	cube is released and all node processes are terminated when the process that invoked **getcube** exits or is killed.
`1`	cube is not released until the owner releases the cube with **relcube()** or the **relcube** command.

account is a pointer to a character string or a null pointer.

Discussion

Use **getcube()** to allocate a new cube, assign a name, type, host, and lifetime to the cube. It also starts a file server that will handle I/O (read, write and print) operations for the allocated cube. You assign names to cubes when you invoke **getcube()** so that the system will know which cube a program will be connected to. All cubes allocated by a single user must have unique names.

The *cubetype* parameter specifies the size and, if desired, the type of cube you want allocated. Cubes are always allocated such that the number of nodes is a power of two. Thus, in addition to the higher powers of two, you may allocate a cube composed of zero or one node. In addition to the type of node, you may also specify the minimum required memory amount, so that the nodes allocated will have at least the required memory, but may have more.

So, `2` specifies a two-node cube in which the nodes can be of any type, `16vx` specifies a 16-node vector cube, and `d3m4` specifies an eight-node cube in which each node has at least 4M bytes of memory per node. To get a cube composed of different types of nodes, specify the different sizes and types, separated by a slash (/), in one string. For example, `4vx/4m8` gets four vector nodes and four nodes with at least 8M bytes of memory each.

GETCUBE() *(cont.)*

GETCUBE() *(cont.)*

You are automatically attached to the last cube allocated. Subsequent message passing and loader calls made by the process which called **getcube**() will apply to the newly allocated cube. Use **attachcube**() when you want to access a different allocated cube.

All the cubes that you own are automatically released when you log out of the SRM unless you invoke your program with **nohup**. To invoke your program with **nohup** enter:

```
nohup progname &
```

Logging out of a remote host does not automatically release cubes.

If any of the parameters are not specified, default values are used. With no arguments, **getcube**() will reserve a cube for you with the name, "defaultname." The number of nodes reserved for this cube will be the largest integral power of two that is possible given the existing set of unallocated nodes.

If the cube cannot be allocated, an error message results and the process will terminate. Use **cubeinfo**() to determine the size (as well as other information) of the cube that was allocated.

When **getcube**() is executed, it calls the routine called **account_hook**(). Exactly what **account_hook**() does is specific to your installation. The default version of **account_hook**() does no accounting. If you change **account_hook**() to write to standard output, the writes go to */usr/ipsc/commser.log*. The parameter *account* is passed to **account_hook**() as an account name. If *account* is a null pointer, the parameter passed is the string *defaultaccount*.

The source for **account_hook**() is provided. Modifying **account_hook**() is a superuser activity and should only be performed by the system administrator. The system administrator may modify **account_hook**() to perform other actions that control and monitor node usage. For example, one possible use of **account_hook**() is to verify the user requesting node allocation and deny allocation to users with insufficient privilege.

Errors

`getcube: Internal cube usage limit`

Limit of 10 cubes may be allocated. Try again later when system is not as busy.

`getcube: permission denied`

If the accounting routines reject a user, this message is given.

GETCUBE() *(cont.)*

GETCUBE() *(cont.)*

`getcube: Commser not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with the **bootcube** or **rebootcube** command.

`getcube: Lifeline not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with the **bootcube** or **rebootcube** command.

`getcube: Bootcube in progress`

A **bootcube** or **rebootcube** command is currently in progress. The system does not allow any cubes to be allocated until the **bootcube** or **rebootcube** command is complete.

`getcube: Cubename already exists`

Use a different name.

`getcube: No SRM that matched request was found`

Remote host error only. Could not find an SRM with a cube of the type requested.

`getcube: Commser out of buffers`

Communication server out of buffers.

`getcube: Cubetype not available`

The cubetype is present but currently in use.

`getcube: Cubetype not installed`

The cubetype is not present.

`getcube: Cubetype not valid`

The request is invalid.

GETCUBE() *(cont.)*

GETCUBE() *(cont.)*

See Also

attachcube(), cubeinfo(), newserver(), numnodes(), relcube()

GETIPHOSTS() GETIPHOSTS()

Get a list of socket nodes.

Synopsis

char **getiphosts()

Parameter Declarations

None

Return Value

A pointer to an array of character pointers.

Example

This example calls **getiphosts()** and puts the list of cube iphosts in *iphlist*. It then checks each value and makes a particular one the iphost. Notice the need to declare **getiphosts()** as external.

```
#include <CMC/sys/types.h>
#include <CMC/sys/socket.h>
#include <CMC/netinet/in.h>
#include <CMC/netdb.h>
#include <CMC/ntoh.h>
#include <cube.h>

#define NULL 0
extern char **getiphosts();

long iam;
main(argc,argv)
int argc;
char **argv;
```

GETIPHOSTS() *(cont.)*

GETIPHOSTS() *(cont.)*

```
{
char            **iphlist;
struct hostent  *hp;
int             i;
   iam = mynode();
   if(argc != 2) {
      if(!iam)
         printf("Usage: %s <iphostname>\n",argv[0]);
      exit(1);
   }
   iphlist = getiphosts();
   i       = 0;
   while(iphlist[i] != NULL) {
      if(!strcmp(iphlist[i],argv[1])) {
         setiphost(iphlist[i]);
         hp = gethostbyname(iphlist[i]);
         break;
      }
      i++;
   }
   if(iphlist[i] != NULL) {
      if(!iam) printf("%d: host is %s\n",iam,hp->h_name);
   }
   else {
      if(!iam) printf("Unable to find:%s\n",argv[1]);
   }
}
```

Environment

Node

Languages

C

Description of Parameters

None

GETIPHOSTS() *(cont.)*

GETIPHOSTS() *(cont.)*

Discussion

Use **getiphosts()** to obtain a list of possible cube iphosts. When a node program is loaded, the NX/2 operating system assigns a socket node as its iphost if there are iphosts available. It chooses the socket node that has been assigned the least number of times. If you want to override this designation, use **setiphost()**. **getiphosts()** returns a list that you can choose from.

The names on the list returned by **getiphosts()** are the official network names (without the domain) of the socket nodes installed as specified by the name server or */etc/hostfile* for the SRM.

Errors

```
getiphosts: Can't assign requested address
```

The iPSC/860 system has no running socket nodes.

See Also

setiphost()

GINV() GINV()

Returns the position of an element in the binary-reflected Gray code sequence.

Synopsis

long **ginv** (*j*)

Parameter Declarations

long *j*;

Return Value

An integer interpreted as the corresponding position of the gray code value used as input.

Example

This example returns the position of the gray code value 6 and stores it in the variable *x*:

```
long x;
x = ginv(6);
```

For the above example, the variable *x* will hold a 4.

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

j a positive integer value.

GINV() *(cont.)*

GINV() *(cont.)*

Discussion

The **ginv()** takes a Gray code element (positive integer) as input and returns the position of the element in the binary-reflected Gray code sequence. This is the inverse of **gray()**. For the inverse function, obtaining the corresponding gray code value with an integer value as input, use the **gray()** function. The following table shows the gray code translation for the integer values from 0 through 7.

000	001	011	010	110	111	101	100	Binary Gray code
0	1	3	2	6	7	5	4	Decimal Gray code
0	1	2	3	4	5	6	7	Position of each element

Errors

None

See Also

gray()

GOPF() GOPF()

Makes a global operation of a user-defined associative and commutative function.

Synopsis

gopf(*x, xlen, work, f*)

Parameter Declarations

char *x[];*
long *xlen;*
char *work[];*
long *(*f)();*

Return Value

None

Example

In this example a vector is distributed over the nodes of a partition. Node 1 has the maximum element.

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

```
#include <math.h>

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

GOPF() *(cont.)*

GOPF() *(cont.)*

```
    N = 10;
    xlen      = sizeof(x);
    mine.node = mynode();
    iam       = mine.node;
    max_loc   = 0;
    for(i=0;i<N;i++)
       x[i] = (double) (iam*N + i);
    if(iam ==1) x[4] = 1000.00;
    mine.max  = fabs(x[0]);
    if(iam==0) {
      printf("\n");
      printf(" %2d:  ",iam);
      for(i=0;i<N; i++)
         printf("  %3.1f ",x[i]);
      printf("\n");
      gsync();
    }
    else {
       gsync();
       printf(" %2d:  ",iam);
       for(i=0;i<N; i++)
          printf("  %3.1f",x[i]);
       printf("\n");
    }
      for(i=1; i<N; i++) {
        if(mine.max < fabs(x[i])) {
           mine.max = fabs(x[i]);
           max_loc  = i;
        }
     }
     gopf((char *)&mine, sizeof(mine), (char *)&work, max_node);
     if(iam==0) {
        printf("mine.max  = %f\n",mine.max);
        printf("mine.node = %d\n",mine.node);
     }
}

long max_node(mine,work)
char *mine, *work;
```

GOPF() *(cont.)*

GOPF() *(cont.)*

```
{
struct PIVOT_NODE *smine, *swork;
int iam;
   iam = mynode();
   smine = (struct PIVOT_NODE *)mine;
   swork = (struct PIVOT_NODE *)work;
    if( smine->max <= swork->max) {
      smine->max = swork->max;
      smine->node = swork->node;
   }
}
```

Environment

Node

Languages

C, Fortran

Description of Parameters

x	refers to the input vector to be used in the operation. When the operation completes, it contains the final result. Note that *x* can be any type.
xlen	the length of the vector *x* in bytes.
work	refers to an array which is used to receive contributions from other nodes. The length of work must be at least *xlen* .
f	the function to be called. It is called as *f(x, work)*, which sets *x* to whatever function of *x* and *work* is desired.

GOPF() *(cont.)*

GOPF() *(cont.)*

Discussion

Use **gopf()** to give some function *f()* the same global properties that the other global communications routines, such as **gdsum(),** have. These properties are:

- All nodes must issue **gopf(),** which in turn calls the function.
- Each node calculates the result, which is stored in *x*. The work array is used to receive messages from other nodes.
- The result is returned in *x* to all nodes.

The function *f()* must be a commuttive and associative function of two vectors, returning its result in hte first vector. It must have two parameters:

x	the input vector to be used in the operation. When the operation is complete, it contains the final result.
work	an array which is used to receive the other contributions. The length of *work* must be at least *xlen.*

All participating processes must have the same *pid.* Thisis not a problem for RX nodes (using the single process model) which always have *pid* set to 0.

Errors

None

GRAY() GRAY()

Returns the binary-reflected gray code for an integer.

Synopsis

long **gray** (*j*)

Parameter Declarations

long *j*;

Return Value

An integer that is the binary-reflected Gray code for the input.

Example

To return the gray code value corresponding to a value of `4` and store it in the variable *x*, use the statements:

```
long x;
x = gray(4);
```

In this example, the returned value is `6`.

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

j is a positive long integer value.

GRAY() *(cont.)*

GRAY() *(cont.)*

Discussion

The gray code is a useful tool in determining where a particular node is in a given application topology (ring, mesh, tree, or linear array) and in optimally assigning tasks to nodes. **gray** () takes a positive integer (*j*) as input and returns the gray code element corresponding to *j* in the binary-reflected gray code sequence. For the inverse function, obtaining the position from a gray code value, use the **ginv**() call. The following table shows the gray code translation for values from 0 through 7.

000	001	011	010	110	111	101	100	Binary Gray code
0	1	3	2	6	7	5	4	Decimal Gray code
0	1	2	3	4	5	6	7	Position of each element

Errors

None

See Also

ginv()

GSENDX() GSENDX()

Send a vector to a list of nodes.

Synopsis

gsendx(*type, x, xlen, nodenums, nlen*)

Parameter Declaration

long *type;*
char *x[];*
long *xlen;*
long *nodenums[];*
long *nlen;*

Return Values

None

Environment

Node

Languages

C, Fortran

Description of Parameters

type	message type. Must be the same for all participating processes. There must be no other messages of this type in the system.
x	refers to the input vector to be sent. Note that *x* can be of any type.
xlen	the number of bytes of data in vector *x*.
nodenums	refers to the list of node numbers to which data is to be sent.
nlen	the number of node numbers in the list.

GSENDX() *(cont.)*

GSENDX() *(cont.)*

Discussion

Use **gsendx()** to send a vector to a set of nodes specified by *nodenums*. All the nodes that are to receive the vector must call **crecv(), irecv(),** or **hrecv()** to receive the message, using the type specified for **gsendx()** above. All participating processes must have the same *pid*. This is not a problem for RX nodes (using the single process model) with *pid* always set to `0`.

Errors

`gsendx: Invalid node`

Use **numnodes()** to determine cube size and **myhost()** to determine host node number.

`gsendx: Invalid buffer pointer`

Specify a pointer which contains the address of a valid vector.

`gsendx: Invalid length`

Use a non-negative number for *len* or *nlen*.

`gsendx: Buffer length exceeds allocation`

Make sure vector is large enough.

See Also

csend(), isend()

GSYNC() GSYNC()

Global synchronization operation.

Synopsis

gsync()

Parameter Declarations

None

Environment

Node

Languages

C, Fortran

Discussion

Use **gsync()** to synchronize node processes. This is not a precise synchronization; when a node process executes **gsync()**, it waits until all other nodes in the cube have called **gsync()** before continuing to execute the process. All participating processes must have the same *pid*. This is not a problem for RX nodes (using the single process model) with *pid* always set to `0`.

Errors

None

See Also

waitall(), waitone()

GxAND() GxAND()

Global AND operation for integer or logical vectors or scalars.

Synopses

Integer (bitwise operation):

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

Logical (logical operation):

gland(*x, n, work*)
long *x*[];
long *n;*
long *work*[];

Return Values

None

Environment

Node

Languages

C, Fortran

GxAND() *(cont.)*

GxAND() *(cont.)*

Description of Parameters

x	the input vector or scalar to be used in the operation. When the operation is complete, it contains the final result.
n	the number of elements in *x*.
work	an array which is used to receive the contributions from other nodes. The length of work must be at least *n*.

Discussion

Use **giand()** or **gland()** to calculate the bitwise or logical AND of each integer or logical component (depending on which routine you use) of *x* across all nodes. The result is returned in *x* to every node. Each node receives the results from all the other nodes in the partition in the work array. All participating processes must have the same *pid*. This is not a problem for RX nodes (using the single process model) with *pid* always set to 0.

This is a global operation, meaning that all nodes in a partition must execute this operation before the process can continue on each node. When *x* is a vector, each element of the resulting vector represents the accumulation of the corresponding index of the input vector from each node.

Errors

`gxand: Received message too long for buffer`

Make sure *n* is the same for all nodes.

`gxand: Received message too short for buffer`

Make sure *n* is the same for all nodes.

`gxand: Invalid buffer pointer`

Specify a pointer which contains the address of a valid vector.

GxAND() *(cont.)*

GxAND() *(cont.)*

```
gxand: Invalid length
```

Use a non-negative number for *n*.

```
gxand: Buffer length exceeds allocation
```

Make sure vector is large enough.

GxHIGH() GxHIGH()

Global MAX operation for double precision, integer, or real vectors or scalars.

Synopses

Double precision:

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

Integer:

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

Real:

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

Return Values

None

Example

This example uses **gihigh()** to find the largest among a list of numbers. The list is distributed among the processors in straightforward domain decomposition; specifically, list[*i*] on processor *p* represents the $(i+B*p)^{th}$ element of the global list, where *B* is the number of elements on each processor. It is assumed that the size of the global list is $n*B = 2^k$ where *n* is the number of processors and *k* is some positive integer.

GxHIGH() *(cont.)*

GxHIGH() *(cont.)*

```
                                    /* Returns the largest value among all values of the global list */
findMax(list,numElements)
{
     int list[];                                    /* the list to be searched */
     int numElements;                    /* the number of elements locally stored */
     int maxElement, index;
     int temp, k;
     index = 0;                                /* find largest value for this process */
     for (k=1; k< numElements; k++)
     {
         if (list[k] > list[index])
             index = k;
     }                                                                  /* endfor */
     maxElement = list(index)
     gihigh(&maxElement, 1, &temp);     /* Now check across all processors */
     return (maxElement);
     }
```

Environment

Node

Languages

C, Fortran

Description of Parameters

x	the input vector or scalar to be used in the operation. When the operation is complete, it contains the final result.
n	the number of elements in *x*.
work	an array which is used to receive the contributions from other nodes. The length of work must be at least *n*.

GxHIGH() *(cont.)*

GxHIGH() *(cont.)*

Discussion

Use **gdhigh(), gihigh(), or gshigh()** to calculate the maximum value of the double precision, integer, or real component (depending on which routine you use) of *x* across all nodes. The result is returned in *x* to every node. The work array is used to receive messages from other nodes. All participating processes must have the same *pid*. This is not a problem for RX nodes (using the single process model) which only run *pid* 0.

This is a global operation meaning that all nodes in a cube must execute this operation before the process can continue on each node. When *x* is a vector, each element of the resulting vector represents the accumulation of the corresponding index of the input vectors from each node.

Errors

`gxhigh: Received message too long for buffer`

Make sure *n* is the same for all nodes.

`gxhigh: Received message too short for buffer`

Make sure *n* is the same for all nodes.

`gxhigh: Invalid buffer pointer`

Specify a pointer which contains the address of a valid vector.

`gxhigh: Invalid length`

Use a non-negative number for *n*.

`gxhigh: Buffer length exceeds allocation`

Make sure value of *n* does not exceed control vector size.

GxLOW() GxLOW()

Global vector MIN operation for double precision, integer, or real vectors or scalars.

Synopses

Double precision:

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

Integer:

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

Real:

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

Return Values

None

Example

This example uses **gilow()** to find the smallest among a list of numbers. The list is distributed among the processors in straightforward domain decomposition; specifically, list[*i*] on processor *p* represents the $(i+B*p)^{th}$ element of the global list, where *B* is the number of elements on each processor. It is assumed that the size of the global list is $n*B = 2^k$ where *n* is the number of processors and *k* is some positive integer.

GxLOW() *(cont.)*

GxLOW() *(cont.)*

```
                                        /* Returns the smallest value among all values of the global list */
findMin(list,numElements)
int list[];                                                    /* the list to be searched */
int numElements;                                  /* the number of elements locally stored */
{
     int minElement, index;
     int temp, k;
     int index = 0;                                  /* find smallest value for this process */
     for (k=1; k< numElements; k++)
     {
         if (list[k] < list[index])
             index = k;
     }                                                                          /* endfor */
     minElement = list(index)
     gilow(&minElement, 1, &temp);          /* Now check across all processors */
     return (minElement);
}
```

Environment

Node

Languages

C, Fortran

Description of Parameters

x	the input vector or scalar to be used in the operation. When the operation is complete, it contains the final result.
n	the number of elements in *x*.
work	an array which is used to receive the contributions from other nodes. The length of work must be at least *n*.

GxLOW() *(cont.)*

GxLOW() *(cont.)*

Discussion

Use **gdlow(), gilow(), or gslow()** to calculate the minimum value of the double precision, integer, or real component (depending on which routine you use) of *x* across all nodes. The result is returned in *x* to every node. The work array is used to receive messages from other nodes. All participating processes must have the same *pid*. This is is not a problem for RX nodes (using the single process model) which only run *pid* 0.

This is a global operation meaning that all nodes in a cube must execute this operation before the process can continue on each node. When *x* is a vector, each element of the resulting vector represents the accumulation of the corresponding index of the input vectors from each node.

Errors

`gxlow: Received message too long for buffer`

Make sure *n* is the same for all nodes.

`gxlow: Received message too short for buffer`

Make sure *n* is the same for all nodes.

`gxlow: Invalid buffer pointer`

Specify a pointer which contains the address of a valid vector.

`gxlow: Invalid length`

Use a non-negative number for *n*.

`gxlow: Buffer length exceeds allocation`

Make sure value of *n* does not exceed control vector size.

GxOR() GxOR()

Global OR operation for integer or logical vectors or scalars.

Synopses

Integer (bitwise operation):

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

Logical (logical operation):

glor(*x, n, work*)
long *x*[];
long *n;*
long *work*[];

Return Values

None

Environment

Node

Languages

C, Fortran

G*x*OR() *(cont.)*

G*x*OR() *(cont.)*

Description of Parameters

x	the input vector or scalar to be used in the operation. When the operation is complete, it contains the final result.
n	the number of elements in *x*.
work	an array which is used to receive the contributions from other nodes. The length of work must be at least *n*.

Discussion

Use **gior()** or **glor()** to calculate the bitwise or logical OR of each integer or logical component (depending on which routine you use) of *x* across all nodes. The result is returned in *x* to every node. The work array is used to receive messages from other nodes. All participating processes must have the same *pid*. This is not a problem for RX nodes (using the single process model) which only use *pid* 0.

This is a global operation, meaning that all nodes in a cube must execute this operation before the process can continue on each node. When *x* is a vector, each element of the resulting vector represents the accumulation of the corresponding index of the input vectors from all nodes.

Errors

`gxor: Received message too long for buffer`

Make sure *n* is the same for all nodes.

`gxor: Received message too short for buffer`

Make sure *n* is the same for all nodes.

`gxor: Invalid buffer pointer`

Specify a pointer which contains the address of a valid vector.

GxOR() *(cont.)*

GxOR() *(cont.)*

```
gxor: Invalid length
```

Use a non-negative number for *n*.

```
gxor: Buffer length exceeds allocation
```

Make sure value of *n* does not exceed control vector size.

GxPROD() GxPROD()

Global multiplication operation for double precision, integer, or real vectors or scalars.

Synopses

Double precision:

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

Integer:

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

Real:

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

Return Values

None

Environment

Node

Languages

C, Fortran

GxPROD() *(cont.)*

GxPROD() *(cont.)*

Description of Parameters

x	the input vector or scalar to be used in the operation. When the operation is complete, it contains the final result.
n	the number of elements in *x*.
work	an array which is used to receive the contributions from other nodes. The length of work must be at least *n*.

Discussion

Use **gdprod()**, **giprod()**, or **gsprod()** to calculate the product of the double precision, integer, or real component (depending on which routine you use) of *x* across all nodes. The result is returned in *x* to every node. The work array is used to receive messages from other nodes. All participating processes must have the same *pid*. This is not a problem for RX nodes (using the single process model) which only use *pid* 0.

This is a global operation, meaning that all nodes in a cube must execute this operation before the process can continue on each node. When *x* is a vector, each element of the resulting vector represents the accumulation of the corresponding elements of the input vectors from other nodes.

Errors

```
gxprod: Received message too long for buffer
```

Make sure *n* is the same for all nodes.

```
gxprod: Received message too short for buffer
```

Make sure *n* is the same for all nodes.

```
gxprod: Invalid buffer pointer
```

Specify a pointer which contains the address of a valid vector.

GxPROD() *(cont.)*

GxPROD() *(cont.)*

```
gxprod: Invalid length
```

Use a non-negative number for *n*.

```
gxprod: Buffer length exceeds allocation
```

Make sure value of *n* does not exceed control vector size.

G*x*SUM() G*x*SUM()

Global sum operation for double precision, integer, or real vectors or scalars.

Synopses

Double precision:

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

Integer:

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

Real:

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

Return Values

None

Example

This example uses **gdsum**() to obtain the sum of the dot products of a vector of length *n* that is distributed among the nodes of the cube, where *x*(*m*) is a piece of the vector on a given node. Note that *m* need not have the same value on all nodes.

GxSUM() *(cont.)*

GxSUM() *(cont.)*

```
int i;
double x[m], dot, dummy;
                                        /* Assume that x and m have been initialized */

                                        /* Compute the local dot product*/
dot = 0;
for (i = 0; i<m; i++)
    dot = dot + x[i]*x[i];
                                        /* Sum global contributions */
gdsum (&dot, 1, &dummy);
```

Environment

Node

Languages

C, Fortran

Description of Parameters

x	the input vector or scalar to be used in the operation. When the operation is complete, it contains the final result.
n	the number of elements in *x*.
work	an array which is used to receive the contributions from other nodes. The length of work must be at least *n*.

Discussion

Use **gdsum()**, **gisum()**, or **gssum()** to calculate the sum of the double precision, integer, or real component (depending on which routine you use) of *x* across all nodes. The result is returned in *x* to every node. The work array is used to receive messages from other nodes. All participating processes must have the same *pid*. This is not a problem for RX nodes (using the single process model) which only use *pid* 0.

This is a global operation, meaning that all nodes in a partition must execute this operation before the process can continue on each node. When *x* is a vector, each element of the resulting vector represents the accumulation of the corresponding element of the input vectors from each node.

G*x*SUM() *(cont.)*

G*x*SUM() *(cont.)*

Errors

`gxsum: Received message too long for buffer`

Make sure *n* is the same for all nodes.

`gxsum: Received message too short for buffer`

Make sure *n* is the same for all nodes.

`gxsum: Invalid buffer pointer`

Specify a pointer which contains the address of a valid vector.

`gxsum: Invalid length`

Use a non-negative number for *n*.

`gxsum: Buffer length exceeds allocation`

Make sure value of *n* does not exceed control vector size.

GxXOR() GxXOR()

Global exclusive OR operation for integer or logical vectors or scalars.

Synopses

Integer (bitwise operation):

gixor(*x, n, work*)
long *x*[];
long *n;*
long *work*[];

Logical (logical operation):

glxor(*x, n, work*)
long *x*[];
long *n;*
long *work*[];

Return Values

None

Environment

Node

Languages

C, Fortran

GxXOR() *(cont.)*

GxXOR() *(cont.)*

Description of Parameters

x	the input vector or scalar to be used in the operation. When the operation is complete, it contains the final result.
n	the number of elements in *x*.
work	an array which is used to receive the contributions from other nodes. The length of work must be at least *n*.

Discussion

Use **gixor()** or **glxor()** to calculate the bitwise or logical exclusive OR of each integer or logical component (depending on which routine you use) of *x* across all nodes. The result is returned in *x* to every node. The work array is used to receive messages from other nodes. All participating processes must have the same *pid*. This is not a problem for RX nodes (using the single process model) which only use *pid* 0.

This is a global operation, meaning that all nodes in a cube must execute this operation before the process can continue on each node. When *x* is a vector, each element of the resulting vector represents the accumulation of the corresponding index of the input vectors from all nodes.

Errors

`gxxor: Received message too long for buffer`

Make sure *n* is the same for all nodes.

`gxxor: Received message too short for buffer`

Make sure *n* is the same for all nodes.

`gxxor: Invalid buffer pointer`

Specify a pointer which contains the address of a valid vector.

GxXOR() *(cont.)*

GxXOR() *(cont.)*

`gxxor: Invalid length`

Use a non-negative number for *n*.

`gxxor: Buffer length exceeds allocation`

Make sure value of *n* does not exceed control vector size.

HANDLER() HANDLER()

Provide a user-written handler for program failures.

Synopsis

handler(*etype, proc*)

Parameter Declarations

long *etype;*
void (**proc*)();

Return Value

None

Example

This example attaches a handler to exception 13. The example is intended for a CX node.

```
void expc13();
main()
{
char c;
int *p;
   handler(13,expc13);
   p  = (int *)-1;
   *p =1;
}

void expc13(ecode)
long ecode;
{
   printf("Exception 13 occurred.\n");
   exit(1);
}
```

On an RX node the same program would cause an exception 14.

HANDLER() *(cont.)*

HANDLER() *(cont.)*

Environment

Node

Languages

C, Fortran

Description of Parameters

etype the type of exception for which a user-written handler is provided. The *etype* is an integer between 0 and 31. The "Discussion" section of this manual page describes the meanings of these integers.

proc the procedure executed when the specified exception occurs. This procedure must be user-written and must be a C procedure with one argument. The argument is the hardware error code generated by the hardware depending upon the exception type selected.

For types other than 11-14, there is no code, and the argument passed to the procedure will be 0. For type 31, the argument passed is the current **mclock()** value. For exception handler types 11-14, the argument will be the hardware error code explained in the *Intel 386™ Programmer's Reference Manual.* For RX node operation, the only implemented exception types are 3, 6, 13, and 14. For these exception types, the argument passed to the procedure is 0.

Discussion

Use **handler()** to allow a node process to execute a user-written exception handler when a specific exception occurs. The handler must be written in C and have one argument, as discussed above. Only the process calling **handler()** can use the exception handler to which it refers.

For CX node operation, use the *proc* parameter as described above in the Description of Parameters. For the RX node, the only implemented exception types are 3, 6, 13, and 14. The preferred method of handling signals is to use the UNIX **signal()** call. Refer to the *UNIX System V Programmer's Reference Manual* for a discussion.

If an exception occurs for which no handler is specified, the default action is to print a message and kill the process. Exception types 0-31 indicate program failure. Exception types 2, 8, and 17-31 are reserved.

HANDLER() *(cont.)*

HANDLER() *(cont.)*

Available hardware exception types are as follows:

0	**Divide Error.** Occurs if an integer quotient is too large or an attempt is made to divide by zero.
1	**Single Step.** Allows programs to execute one instruction at a time.
3	**Break Point.** Occurs when the one-byte breakpoint instruction (INT 3) is executed.
4	**Overflow.** Occurs when the **INTO** instruction is executed when the overflow bit of the flags register is set.
5	**Bounds Check.** Occurs when the **BOUND** instruction is executed when the specified array index is found to be invalid with respect to the given array bounds.
6	**Invalid Opcode.** Occurs if execution of an invalid opcode is attempted. May also occur when the effective address given by certain instructions is a register rather than a memory location.
7	**Processor Extension Not Available.** Occurs when an 80387 floating point instruction is attempted on a node which does not have an 80387.
9	**Processor Extension Segment Overrun.** Occurs when a processor extension memory operand is not completely contained in a single segment.
11	**Segment or Call Gate Not Present.** Occurs when an attempt is made to load a nonpresent segment or to use a control descriptor that is marked not-present.
13	**General Protection Fault.** Occurs when a violation occurs which is not covered by another interrupt.
14	**Page Fault.** Occurs when the application makes a memory reference to a location which has not been allocated to the process.
16	**Processor Extension Error.** Occurs after the numeric instruction that caused the error. On the iPSC system, an exception 16 is always a floating point error.

HANDLER() *(cont.)*

HANDLER() *(cont.)*

Errors

`handler: Invalid handler type`

Specify one of the exception types listed.

See Also

386™ Programmer's Reference Manual
860™ Programmer's Reference Manual
UNIX System V Programmer's Reference Manual: **signal()**

HRECV() HRECV()

Provide user-written receive handler for receive traps.

Synopsis

hrecv(*typesel, buf, len, proc*)

Parameter Declarations

long *typesel;*
char **buf;*
long *len;*
void (**proc*) ();

Return Value

None

Example

This example posts an **hrecv**() to receive message type 101 and on receipt, execute an exception handler named "handleproc." However, following the posting of **hrecv**() is a section of code that must not be interrupted, so **masktrap(1)** is called to prevent the receive handler from being executed until the end of the critical processing, and then is called again to permit normal **hrecv**() operation. If the message type becomes available during the critical processing, it will wait until **masktrap(0)** is called, and then execute "handleproc" immediately.

```
                                        /* Initiate receive and set up handler */
hrecv(101,buf,sizeof(buf),handleproc);
masktrap(1);                            /* Temporarily disable receive trap */
                                            /* Do critical processing here */
    .
    .
masktrap(0);                            /* Re-enable receive trap */
```

HRECV() *(cont.)*

HRECV() *(cont.)*

NOTE

The handler must be written in C and have four arguments. If you are calling the handler from a C program, the first line of the handler and the declaration of the arguments would have the following form:

```
proc (type, count, node, pid)
long type, count, node, pid;
```

The **hrecv()** call passes the four values for the parameters: the message type (*type*), the length of the message (*count*), the node (*node*), and the process id (*pid*) to the handler for you so the **info...()** calls are not needed within the handler.

Environment

Node

Languages

C, Fortran

Description of Parameters

typesel is an integer value that specifies the type (or types) of message that you are waiting for. 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 the receive operation will be recognized. After the first message has been received, you can use `-1` again to receive the next message.

HRECV() *(cont.)*

HRECV() *(cont.)*

If *typesel* is any negative number other than `-1`, a set of message types will be recognized. Refer to Appendix B for instructions on how to build a *typesel* mask.

buf specifies the buffer where the received message will be stored. The buffer can be any legal data type in the language in which you are writing. Data types should match in send and receive operations. *Buf* is not available for use until *proc* is called indicating that the receive has completed.

len is an integer that specifies the size of the message buffer (in bytes) pointed to by *buf*. The maximum number of bytes that you can send in a single message to or from a host process is 256K. There is no limit on message length between nodes.

proc is the procedure executed when the receive is complete. This procedure must be written by the user and must be a C procedure with four arguments.

Discussion

Use **hrecv()** to post a request to receive a message into the buffer, and when receipt of the message is complete, immediately execute a receive trap handler procedure. The name of the handler is the *proc* parameter.

The receive trap handler must be written by the user in C, and must have four arguments: *type*, *count*, *node*, and *pid*. The **hrecv()** call contains **info...()** calls that return information about the message that was received and passes the returned values to the arguments of the handler for you.

It is possible to post several **hrecv()** requests simultaneously, with separate handlers or with a common handler. Because the handler is executed as soon as the receive is complete, you can ensure that a critical section of code is not interrupted by the execution of the handler by issuing the **masktrap()** call to protect that section of code. The example illustrates this situation.

If a message too long to fit in the buffer comes in, the receive is completed and the message is lost with no error returned. To detect this situation, check the count argument in the handler.

HRECV() *(cont.)*

HRECV() *(cont.)*

Errors

`hrecv: Buffer length exceeds allocation`

Make sure parameter length does not exceed buffer size.

`hrecv: Invalid length`

Use a non-negative number of a length less than or equal to maximum message length.

`hrecv: Not enough memory`

Reduce the size of the process.

`hrecv: Too many requests`

Use **msgwait()** or **msgdone()** for **isend()** or **irecv()** requests that are outstanding, or use a force type.

`hrecv: Invalid buffer pointer`

(C only) Specify a pointer which contains the address of a valid data buffer or correct the *proc* parameter.

See Also

crecv(), irecv(), handler(), hsend(), hsendrecv()

HSEND() HSEND()

Send a message and provide user-written exception handler on completion of the send.

Synopsis

hsend(*type, buf, len, node, pid, proc*)

Parameter Declarations

long *type;*
char **buf;*
long *len, node, pid;*
void (**proc*) ();

Return Value

None

Example

This example uses **hsend()** to send message type 101 and execute an exception handler named "handleproc" as soon as the send is complete.

```
                                        /* Initiate send and set up handler */
hsend(101, buf, sizeof(buf), node_id, 0, handleproc);
```

NOTE

The handler must be written in C and have four arguments. If you are calling the handler from a C program, the first line of the handler and the declaration of the arguments would have the following form:

```
proc (type, count, node, pid)
long type, count, node, pid;
```

The **hsend()** call passes the four values for the parameters: the message type (*type*), the length of the message (*count*), the destination node (*node*), and the destination process id (*pid*) to the handler for you.

HSEND() *(cont.)*

HSEND() *(cont.)*

Environment

Node

Languages

C, Fortran

Description of Parameters

type — a user-defined integer that identifies the type of the message that you are sending.

buf — refers to the buffer that contains the message to be sent. The buffer may be of any legal data type.

len — a positive integer that specifies the size (in bytes) of the message that you wish to send. Messages to or from a host process are limited to a maximum of 256K bytes. There is no limit on message length between nodes.

node — an integer that defines the node(s) to which the message is to be sent

pid — for CX nodes, an integer that specifies the id of the process that is to receive the message. Valid *pids* include any integer value. Negative numbers are reserved for system programs. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on RX nodes). A node *pid* is assigned when you load a process and a host *pid* is assigned with **setpid()**.

proc — the procedure executed when the send is complete. This procedure must be written by the user and must be a C procedure with four arguments.

HSEND() *(cont.)*

HSEND() *(cont.)*

Discussion

Use **hsend**() as an asynchronous send that sends a message to the specified node and process id and executes a handler procedure as soon as the send is complete. The name of the handler is the *proc* parameter for the call.

The handler must be written by the user in C, and must have four arguments: *type, count, node*, and *pid*. The **hsend**() call passes values to the arguments of the handler for you.

It is possible to post several **hsend**() requests simultaneously, with separate handlers or with a common handler. Because the handler is executed as soon as the send is complete, you can ensure that a critical section of code is not interrupted by the execution of the handler by issuing the **masktrap**() call to protect that section of code.

It is recommended that you use a *type* with a value from `0` to `1,000,000`. Types `1,073,741,824` to `1,999,999,999` are special force types that bypass the normal flow control mechanisms, do not match the `-1` wildcard, and are discarded if no receive is posted. Force types are intended to be used with **csendrecv**(), **isendrecv**(), and **hsendrecv**(), but can be used by other message passing calls.

NOTE

It is not recommended that you use *types* in other ranges because they are used by the system and could produce unpredictable results.

If *node* is a positive integer, the message is sent to the node with that number. Nodes within a partition are numbered consecutively from 0, and a node number higher than the highest node in the cube causes an error to be returned. To send a message to the host, use the host's id, which is returned by **myhost**().

If *node* is a negative number up to a certain value, it causes a "global" send. If node is `-1`, the message is sent to all nodes. Other negative numbers cause the message to be sent to a subcube composed of a set of the nodes surrounding the sender. The dimension of this cube is calculated as (*d* - **nodedim**() `-1`) , where *d* is the dimension of the subcube, and is less than or equal to **nodedim**().

For RX nodes, if a global send specifies its own process (**mypid**()) it does not receive its own message. For CX nodes, however, if a process other than its own is specified, the sending node also receives the message.

HSEND() *(cont.)*

HSEND() *(cont.)*

Errors

`hsend: No pid defined`

Use **setpid()** to define a host process id.

`hsend: Invalid type`

Use a non-negative number.

`hsend: Invalid length`

Use a non-negative number or a length less than or equal to maximum message length.

`hsend: Invalid node`

Use **numnodes()** to determine cube size or **myhost()** to determine host number.

`hsend: Invalid buffer pointer`

Specify a pointer which contains the address of a valid data buffer.

`hsend: Not enough memory`

Make sure message buffer starts on a boundary that is a multiple of 4 or make more memory available to this process.

See Also

hrecv(), **masktrap()**, **csend()**, **isend()**

HSENDRECV() HSENDRECV()

Send a message and receive a reply; on receipt of the reply, executes a user-defined trap handler.

Synopsis

hsendrecv(*type, sbuf, slen, tonode, topid, typesel, rbuf, rlen, proc*)

Parameter Declarations

long *type, typesel;*
char **sbuf, *rbuf,;*
long *slen, tonode, topid, rlen;*
void (**proc*) ();

Return Value

None

Example

This example uses **hsendrecv()** both to send a message and to post a receive for the result. On receipt, it will execute an exception handler named **handleproc**. However, following the posting of **hsendrecv()** is a section of code that must not be interrupted, so **masktrap(1)** is called to prevent the receive handler from being executed until the end of the critical processing, and then is called again to permit normal **hsendrecv()** operation. If the message type becomes available during the critical processing, it will wait until **masktrap(0)** is called, and then execute **handleproc** immediately.

```
    .
    .
hsendrecv(SIN_REQUEST_TYPE, &x, sizeof(x), SERVER_NODE,
                SERVER_PID, SIN_REPLY_TYPE, &result,
                sizeof(result),hproc);
masktrap(1);                          /* Temporarily disable receive trap */
    .                                     /* Do critical processing here */
    .
masktrap(0);                                  /* Re-enable receive trap */
}
```

HSENDRECV() *(cont.)*

HSENDRECV() *(cont.)*

```
void hproc(type,count,node,pid)
long type,count,node,pid;
{
    .
    .
}
```

The procedure **handleproc** is as follows:

```
handleproc(type, count, node, pid)
long type, count, node, pid;
{

}
```

NOTE

The handler must be written in C and have four arguments. If you are calling the handler from a C program, the first line of the handler and the declaration of the arguments would have the following form:

```
proc (type, count, node, pid)
long type, count, node, pid
```

The **hsendrecv()** call passes parameter values to the handler for you so the **info...()** calls are not needed within the handler. These parameters apply to the reply message, and are the reply type (*type*), the length of the reply (*count*), the source node (*node*), and the source process id (*pid*).

Environment

Node

Languages

C, Fortran

HSENDRECV() *(cont.)*

HSENDRECV() *(cont.)*

Description of Parameters

type identifies the type of the message that you are sending. Types are described in the Discussion.

sbuf refers to the buffer that contains the message to be sent.

slen is a positive integer that specifies the size (in bytes) of the message that you wish to send. Messages to or from a host process are limited to a maximum of 256K bytes. There is no limit on message length between nodes.

tonode an integer that determines the node(s) to which the message is to be sent.

topid for CX nodes, *pid* is an integer that specifies the id of the process that is to receive the message. Valid *pids* include any integer value. Negative numbers are reserved for system programs. A node *pid* is assigned when you load a process and a host *pid* is assigned with **setpid()**. For RX nodes, *pid* 0 is the only valid process ID (since only a single process can run on RX nodes).

typesel is an integer value that specifies the type(s) of message you are waiting to receive. This parameter is described further in the Discussion.

rbuf refers to the buffer where the reply message will be stored.

rlen is an integer value that specifies the size of the receive buffer in bytes.

proc is the procedure executed when the receive is complete. This procedure must be written by the user and must be a C procedure with four arguments.

Discussion

Use **hsendrecv()** to send a message to a node or host process and simultaneously post a receive for the reply, and when receipt of the message is complete, immediately execute a receive trap handler procedure. The name of the handler is the *proc* parameter. The receive trap handler (the *proc* parameter) must be written by the user in C, and must have four arguments: *type, count, node,* and *pid*. **hsendrecv()** passes these values to the arguments of the handler for you.

It is possible to post several **hsendrecv()** requests simultaneously, with separate handlers or with a common handler. Because the handler is executed as soon as the receive is complete, you can ensure that a critical section of code is not interrupted by the execution of the handler by issuing the **masktrap()** call to protect that section of code.

HSENDRECV() *(cont.)*

HSENDRECV() *(cont.)*

If a message that is too long to fit in the buffer comes in, the receive is completed and the message is lost with no error returned. To detect this situation, check the count argument in the handler.

When a message whose type matches the type(s) specified in the *typesel* parameter, the calling process receives the message, and stores it in *rbuf*. The calling process continues executing instructions while the send-receive operation is occurring. You can use the **info...()** calls to get more information about the reply message.

The *type* parameter is a user-defined variable typically used to identify the kind of information contained in the message being sent. Types `0` to `1,000,000,000` are normal types that can be used by any send or receive call, including this one. Types `1,073,741,824` to `1,999,999,999` are special force types available to the user intended specifically for the "sendrecv" 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. If no receive is posted, as when the receiving process has been killed, a message with a force type is discarded. In general, bypassing the normal flow control mechanisms causes no problem because **hsendrecv()** guarantees that a receive is posted for the message.

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. The buffer for the send and receive may be any legal data type. It is recommended that data types match in send and receive operations.

HSENDRECV() *(cont.)*

HSENDRECV() *(cont.)*

The *typesel* parameter describes the type(s) of the message(s) expected to be received. *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. You can use a force type, as described previously.
- If *typesel* is `-1`, the first message to arrive for the process that initiated the receive operation will be recognized. After the first message has been received, you can use `-1` again to receive the next message.
- If *typesel* is any negative number other than `-1`, a set of message types will be recognized. Refer to Appendix B for information on building a *typesel* mask.

RX nodes can only send messages with *pid*s of 0. Since receive processes ignore *pid*, it is possible to receive a message with a *pid* of any value. However, RX nodes interpret *pid* values other than 0 as an error. To avoid this situation, you should set *pid* to 0 for all processes.

Errors

`hsendrecv: No pid defined`

Use **setpid()** to define a host process id.

`hsendrecv: Invalid type`

Use a non-negative number.

`hsendrecv: Invalid length`

Use a non-negative number or a length less than or equal to maximum message length.

`hsendrecv: Buffer length exceeds allocation`

Make sure parameter length does not exceed buffer size.

`hsendrecv: Invalid node`

Use **numnodes()** to determine cube size or **myhost()** to determine host number.

HSENDRECV() *(cont.)*

HSENDRECV() *(cont.)*

```
hsendrecv: Invalid buffer pointer
```

Specify a pointer which contains the address of a valid data buffer.

```
hsendrecv: Not enough memory
```

Make sure message buffer starts on a boundary that is a multiple of 4 or make more memory available to this process.

See Also

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

HWCLOCK() HWCLOCK()

Places the current value of the hardware counter in the 64-bit unsigned integer supplied as an argument.

Synopsis

void **hwclock**(*hwtime*);

Parameter Declarations

esize_t **hwtime*;

Return Value

None

Example

The following example illustrates the proper use of **hwclock()**.

```
#include <estat.h>
     .
     .
esize_t start_time, end_time, elapse_time;
     .
     .
hwclock(&start_time);                          /* find starting time */
                                               /* run calculations */
     .
     .
hwclock(&end_time);                            /* find ending time */
elapse_time = esub(end_time, start_time);      /* find elapsed time */
```

HWCLOCK() *(cont.)*

HWCLOCK() *(cont.)*

Environment

Node

Languages

C, Fortran

Description of Parameters

hwtime an output variable that receives the current value of the hardware counter in a 64-bit unsigned integer.

Discussion

Places the current value of the hardware counter in the 64-bit unsigned integer supplied as an argument. If an array is supplied, the subroutine places the least significant 32 bits in the first word. The count rate of the hardware is defined in the parameter *HWHZ* in */usr/include/cube.h*. The type **esize_t** is also defined in */usr/include/cube.h*.

Errors

None

See Also

dclock(), mclock()

INFO: INFOCOUNT(), INFONODE(), INFOPID(), INFOTYPE()

Return information about a pending or received message.

Synopsis

long **infocount**()
long **infonode**()
long **infopid**()
long **infotype**()

Parameter Declarations

None

Return Value

The indicated information is returned. The return value is undefined unless the call is made immediately after a **crecv**(), **cprobe**(), or **msgwait**() is complete, or after an **iprobe**() or **msgdone**() return a `1`, all of which indicate that a message is ready.

infocount returns an integer value specifying the length of the message received (in bytes) after a probe or receive operation completes.

infonode returns the node id of the process which sent the message.

infopid returns the *pid* of the process which sent the message.

infotype returns the type of the message. Message type is specified in the send operation.

INFO: *(cont.)*

INFO: *(cont.)*

Example

The following example uses all four of the **info...()** calls. First, it waits for a message of any type and then uses **infotype()** to determine the type. Then, it uses that information to receive the waiting message. After the receive is complete, it uses **infocount()** to obtain the message length. Next, the example posts an **irecv()** for a message of type `10`, and if it has been received, uses **infonode()** and **infopid()** to find out the node and process id from which the message came.

```
#define BIGNUM 65536

long buf[BIGNUM];
long msg_type, msg_len, msg_id;
long from_node, from_pid;
    .
    .
cprobe(-1);                                  /* Wait for any message to arrive. */
msg_type = infotype();            /* Get the type of the message that just arrived. */
crecv(msg_type, buf, sizeof(buf));          /* Receive the waiting message. */
msglen = infocount();                              /* Get length of message. */
    .
    .
                                          /* Post a receive for a type 10 message. */
msg_id = irecv(10, buf, sizeof(buf));
    .
    .
if(msgdone(msg_id)) {             /* After receive is complete, find out who sent it. */
    from_node = infonode();
    from_pid = infopid();
}
```

Environment

Host, Node

Languages

C, Fortran

INFO: *(cont.)*

INFO: *(cont.)*

Description of Parameters

None

Discussion

Use **infocount()**, **infonode()**, **infopid()**, and **infotype()** to return information about a pending or received message. They return information about the message detected when **cprobe()** completes or when **iprobe()** returns `1`. They must immediately follow the related message-passing statement.

Errors

`infocount: No pid defined`

Use **setpid()** to define a host process id.

`infonode: No pid defined`

Use **setpid()** to define a host process id.

`infopid: No pid defined`

Use **setpid()** to define a host process id.

`infotype: No pid defined`

Use **setpid()** to define a host process id.

See Also

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

IODONE() IODONE()

Determines whether an asynchronous I/O read or write operation is complete.

Synopsis

long **iodone**(*id*)

Parameter Declarations

long *id;*

Return Values

1	indicates that the read or write is complete and that the associated buffer is available for reuse in the case of a write operation, or that the buffer contains valid data, in the case of a read operation. You cannot call **iodone()** or **iowait()** with the same *id* parameter after **iodone()** returns 1 because the read or write operation is complete.
0	indicates that the read or write is not yet complete. In this case, you can use the same *id* again until the operation completes.

Example

This example uses **iodone()** to determine whether an **iwrite()** call has finished the write. If so, the buffer can be used; if not, the buffer must be protected until the write is complete.

```
char *buf;
long id, filedes;
        .
        .
id = iwrite(filedes, buf, sizeof(buf));
        .
        .
if (iodone(id)) {
                                        /* You may now reuse the buffer. */
} else {
                                        /* Buffer still needs to be protected. */
}
```

IODONE() *(cont.)*

IODONE() *(cont.)*

Environment

Node

Languages

C, Fortran

Description of Parameters

id the value returned by an **iread()** or **iwrite()** operation.

Discussion

Use **iodone()** to determine whether an **iread** () or **iwrite()** operation is complete, and when **iodone()** returns a `1`, it releases the *id* number that identifies the asynchronous read or write on which you are checking the status. The *id* is the return value of the **iread()** or **iwrite()** you are checking on. You must use either **iodone()** or **iowait()** after an asynchronous message passing call to ensure that the operation is complete and to release the *id* number.

Errors

```
iodone: Invalid message id
```

Use *id* returned by **iread()** or **iwrite()**.

See Also

iowait(), iread(), iwrite()

IOMODE() IOMODE()

Return the current I/O mode for a file.

Synopsis

long **iomode**(*fildes*)

Parameter Declarations

long *fildes;*

Return Value

`0, 1, 2,` or `3,` depending on the current I/O mode for the designated file. See **setiomode()** for information on I/O modes.

Example

The following example uses **iomode()** to return the current I/O mode of the file designated by the file descriptor contained in the variable *fd* (the file descriptor is returned by the UNIX-compatible **open(), creat()**, or **dup()** calls):

```
        .
        .
mode = iomode(fd);
        .
        .
```

Environment

Node

Languages

C, Fortran

IOMODE() *(cont.)*

IOMODE() *(cont.)*

Description of Parameters

fildes the file descriptor.

Discussion

The **iomode()** call returns an integer from `0` through `3` representing the current I/O mode of the file specified by the *fildes* parameter (see **setiomode()** for information on I/O modes). The *fildes* parameter must contain the value returned by the UNIX-compatible **open()**, **creat()**, or **dup()** call when you open a file.

Errors

`iomode: Bad file number`

Use the file descriptor returned by **open()**, **creat()**, or **dup()**.

`iomode: Not implemented on the SRM`

Can be executed by the node only; not available on the host.

See Also

setiomode()

IOWAIT() IOWAIT()

Wait for completion of an asynchronous I/O read or write operation.

Synopsis

iowait(*id*)

Parameter Declarations

long *id;*

Return Value

None

Example

This example uses **iowait()** to stop further execution of the process until an **iread()** posted earlier has completed before continuing, because the code at that point requires that data.

```
int result;
long id,fd;
        .
        .
id = iread(fd, &result, sizeof(result));
        .                                   /* Perform tasks that do not use 'result'. */
        .
        .
iowait(id);                                 /* Wait for read to complete. */
                                            /* Now you can use the value of 'result'. */
```

Environment

Node

Languages

C, Fortran

IOWAIT() *(cont.)*

IOWAIT() *(cont.)*

Description of Parameters

id is the message id returned when you invoke an **iread()** or **iwrite()** operation.

Discussion

The **iowait()** call allows a node process to wait until an **iread()** or **iwrite()** operation is complete, and when the operation is complete, it releases the id number used by the asynchronous I/O operation to which it refers. Use either **iodone()** or **iowait()** after an asynchronous message passing call to ensure that the operation is complete and that the *id* is freed. The *id* parameter identifies the operation that you are waiting for, and must correspond to the value returned by a previous **iread()** or **iwrite()** call.

When the I/O operation is complete, **iowait()** returns and the associated message buffer is available for reuse in the case of a write operation, or when the buffer contains valid data in the case of a read operation. When **iowait()** returns, it clears the id number used by the read or write for use by a future read or write.

Errors

```
iowait: Invalid message id
```

Use message *id* returned by **iread()** or **iwrite()**.

See Also

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

IPROBE() IPROBE()

Determine whether a message of a selected type is pending.

Synopsis

long **iprobe**(*typesel*)

Parameter Declarations

long *typesel;*

Return Values

1	a message of the selected type is waiting to be received
0	a message of the selected type is not waiting to be received

Example

This example shows a continuous for loop that uses **iprobe()** to test for the availability of two messages (types 1 and 2) and processes the first message to come in. If neither message is available, **flick()** is used to allow another process on the same node to execute before looping again.

```
for (;;) {
    if (iprobe(1)) {                    /* Process type 1 message. */
        .
        .
    break;
    }
    else if (iprobe(2)) {               /* Process type 2 message. */
        .
        .

        break;
        }
    else {
        flick();
    }
}
```

IPROBE() *(cont.)*

IPROBE() *(cont.)*

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

typesel is an integer value that specifies the type(s) of message that you are waiting for. 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 non-force-type message to arrive for the process that initiated the probe operation will be recognized. After the first message has been received, you can use `-1` again to probe for the next message.
- If *typesel* is any negative number other than `-1`, a set of message types will be recognized. Refer to Appendix B for instructions on how to build a *typesel* mask.

Discussion

Use **iprobe()** to determine whether a message of a selected type is available for receipt. You specify the message type with the send operation, and **iprobe()** looks for a message whose *type* matches the type(s) specified in the calling process. If a message of the specified type is waiting in the node to be received, **iprobe()** returns a `1`. If there are no such messages, a `0` is returned. When **iprobe()** returns a `1`, the **info...()** calls can be used to get more information about the message.

This is an asynchronous call and therefore does not block the process waiting for a message to arrive. Use **cprobe()** to block the process until a message of a selected type is available for receipt.

IPROBE() *(cont.)*

IPROBE() *(cont.)*

Errors

`iprobe: No pid defined`

Use **setpid()** to define a host process id.

`iprobe: Too many requests`

Use **msgwait()** for outstanding **isend()** or **irecv()** requests.

See Also

cprobe(), infocount(), infonode(), infopid(), infotype()

IREAD() IREAD()

High-speed, asynchronous read from a file.

Synopsis

long **iread**(*fildes, buf, len*)

Parameter Declarations

long *fildes;*
char **buf;*
long *len;*

Return Value

Returns a non-negative id to be used in **iodone**() or **iowait**().

Example

The following example uses **iread**() to read the contents of a file while the process continues to execute. The value of the variable *fd* is the file descriptor returned by the UNIX **open**(), **dup**(), or **creat**() calls. Then, when the data being read by **iread**() is required by the process, the **iowait**() call is used to block the process until the read is complete and to clear the id for use by a future read or write:

```
int result;
long id,fd;
        .
        .                                   /* Read the file into the result buffer */
id = iread(fd, &result, sizeof(result));
        .                                   /* Perform tasks that do not use 'result'*/
        .
iowait(id);                                 /* Wait for read to complete */
                                            /* Now you can use the value of result */
```

IREAD() *(cont.)*

IREAD() *(cont.)*

Environment

Node

Languages

C, Fortran

Description of Parameters

fildes	the file descriptor.
buf	refers to the buffer where the read data will be stored. The buffer can be any valid data type.
len	is an integer value that specifies the size of the buffer in bytes. Transfer sizes are limited only by memory available for the buffer.

Discussion

The **iread()** call reads data from the file specified by the *fildes* parameter which should contain the value returned by the **open()**, **creat()**, or **dup()** UNIX system calls. The **iread()** call is an asynchronous call that allows the process to continue while the data is being read. When you need to use the read data in processing, use the **iowait()** call to block further processing until the read is complete, and to free the id number for use by a future read or write. The **iodone()** call will also clear the *id* when it returns a `1`.

To determine whether the read operation moved the file pointer to the end-of-file, use the **iseof()** call. The **iread()** call modifies the file pointer immediately, so you can call **lseek()**, **iseof()**, or another read or write call immediately without waiting for the read to finish.

IREAD() *(cont.)*

IREAD() *(cont.)*

Errors

`iread: Invalid argument`

Check arguments.

`iread: Mixed file operations`

In I/O mode 2 or 3, nodes are attempting different operations (reads and writes) to a shared file. Ensure that nodes are performing the same operation.

`iread: I/O error`

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

See Also

cwrite(), cread(), iseof(), iwrite(), iowait(), iodone()
UNIX System V/386 Programmer's Reference: **open(), creat(), dup(), read()**

IRECV() IRECV()

Asynchronously receive a message.

Synopsis

long **irecv**(*typesel, buf, len*)

Parameter Declarations

long *typesel;*
char **buf;*
long *len;*

Return Value

Returns a non-negative message id which is to be used in **msgcancel()**, **msgdone()**, or **msgwait()**.

Example

The following example uses **irecv()** to initiate receipt of a message, and when the information in the message is required by the process, uses **msgwait()** both to block further execution of the process until the receive is complete, and to make the id used for the receive available for subsequent asynchronous message passing calls.

```
int buf[50];
long msg_id, type;
     .
     .
msg_id = irecv(type, buf, sizeof(buf));      /* Initiate message receipt.*/

                /* Perform tasks which are not dependent on the message being received.*/
     .
     .
msgwait(msg_id);                                  /* Wait to guarantee message receipt.*/
                                    /* Now use the contents of 'buf' and/or use info routines.*/
```

IRECV() *(cont.)*

IRECV() *(cont.)*

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

typesel is an integer that specifies the type(s) of message that you are receiving. 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 non-force-type message to arrive for the process that initiated the receive operation will be recognized. After the first message has been received, you can use `-1` again to receive the next message.
- If *typesel* is a negative number other than `-1`, a set of message types will be recognized. Refer to Appendix B for instructions on how to build a *typesel* mask.

buf refers to the buffer where the received message will be stored. Valid buffer entries are any legal data type. Data types should match in send and receive operations. The *buffer* is not available for use until **msgwait()** returns or **msgdone()** indicates that the receive has completed.

len is an integer that specifies the size of the message buffer (in bytes). The maximum number of bytes that you can send in a single message to or from a host process is 256K. There is no limit on message length between nodes.

IRECV() *(cont.)*

IRECV() *(cont.)*

Discussion

Use **irecv()** to post an asynchronous receive that initiates the receipt of a message from a process. It searches for a message whose type matches the type(s) specified in the *typesel* parameter, and immediately returns a message id which is used by **msgdone()** and **msgwait()** to determine whether the receive is complete. **msgwait()** also clears the id assigned to the message as soon as the receive completes, as does **msgdone()** if it returns a `1`. You must use either **msgwait()**, **msgdone()**, or **msgcancel()** to free this id up for future sends and receives because the number of IDs available is limited, and an error results if you use up all the IDs. When the message is received it is stored in a buffer (the *buf* parameter).

After **msgwait()** completes or **msgdone()** returns `1`, you can use the **info...()** calls to get more information about the message. **irecv()** does not affect the values returned by the **info...()** calls; it only initiates the receipt of a message.

irecv() is asynchronous, allowing the calling process to continue even if the desired message is not yet available. It returns as soon as the receive request has been initiated. Use **crecv()** when you want the calling process to be blocked until the desired message is received.

Errors

`irecv: No pid defined`

Use **setpid()** to define a host process id.

`irecv: Invalid length`

Use a non-negative number or a length less than or equal to maximum message length.

`irecv: Too many requests`

Use **msgwait()** for **isend()** or **irecv()** requests that are outstanding.

`irecv: Buffer length exceeds allocation`

Make sure parameter length does not exceed buffer size.

`irecv: Invalid buffer pointer`

Specify a pointer which contains the address of a valid data buffer.

IRECV() *(cont.)*

IRECV() *(cont.)*

See Also

crecv(), infocount(), infonode(), infopid(), infotype(), isend(), isendrecv(), msgcancel(), msgdone(), msgwait()

ISEND() ISEND()

Asynchronously send a message.

Synopsis

long **isend**(*type, buf, len, node, pid*)

Parameter Declarations

long *type;*
char **buf;*
long *len, node, pid;*

Return Value

Returns a non-negative message id which is used by **msgcancel()**, **msgdone()**, or **msgwait()**.

Example

The following example uses **isend()** to send a message of type 1 to process 10 on node 16, and when the buffer is required, uses **msgwait()** both to block further execution of the process until the send is complete, and to make the id used for the send available for subsequent asynchronous message passing calls.

```
char msg[132];
long msg_id, len, node, pid, type;
        .
        .
len = sizeof(msg);
node = 16;
pid = 0;
type = 1;
                                        /* Send type 1 message to process 0 on node 16. */
msg_id = isend(type, msg, len, node, pid);
                                    /* Perform tasks that do not change the contents of msg. */
        .
        .
msgwait(msg_id);
                                        /* Now you can re-use the message buffer. */
```

ISEND() *(cont.)*

ISEND() *(cont.)*

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

type is an integer that identifies the message that you want to send. Type restrictions are described in the Discussion.

buf refers to the buffer that contains the message that you want to send. Buffer entries can be any legal data type. Data types should match in send and receive operations.

len is an integer that specifies the size of the message (in bytes) that you want to send. The maximum number of bytes that you can send in a single message to or from a host process is 256K. There is no limit on message length between nodes. If *len* is greater than the maximum message length, an error message will be issued.

node is an integer value that identifies the node or nodes to which the message is to be sent.

pid for CX nodes, *pid* is an integer value that specifies the process id which is to receive the message. Valid *pids* include any integer value. Negative numbers are reserved for system programs. A node *pid* is assigned when you load a process and a host *pid* is assigned with **setpid**(). For RX nodes, *pid* 0 is the only valid process ID (since only a single process can run on the RX node).

Discussion

Use **isend**() to send a message to a node or host process. Completion of **isend**() does not imply that the message was received by the destination process, only that the send operation was initiated. It returns a message id which is used by **msgdone**() and **msgwait**() to determine whether the send is complete. Use **msgwait**() to clear the id assigned to the message as soon as the receive completes, as does **msgdone**() if it returns a `1`. When the message is sent, the buffer can be used for another message if desired.

ISEND() *(cont.)*

ISEND() *(cont.)*

It is asynchronous, returning as soon as the send request is initiated. Use **csend**() when you want to block the calling process until the message is sent.

It is recommended that you use a *type* with a value from `0` to `999,999,999`. Types `1,073,741,824` to `1,999,999,999` are special force types that bypass the normal flow control mechanisms, do not match the `-1` wildcard, and are discarded if no receive is posted. Force types are intended to be used with **csendrecv**(), **isendrecv**(), and **hsendrecv**(), but can be used by other message passing calls.

NOTE

It is not recommended that you use *types* in other ranges because they are used by the system and could produce unpredictable results.

The buffer can be of any *type*. You should not modify *buf* until **msgwait**() returns or **msgdone**() indicates that **isend**() has completed. Otherwise, corrupted data might be sent.

If *node* is a positive integer, the message is sent to the node with that number. Nodes within a partition are numbered consecutively from 0, and a node number higher than the highest node in the cube causes an error to be returned. To send a message to the host, use the host's id, which is returned by **myhost**().

If node is a negative number up to a certain value, it causes "global" send. If node is `-1`, the message is sent to all nodes. Other negative numbers cause the message to be sent to a subcube composed of a set of the nodes surrounding the sender. The dimension of this cube is calculated as (*d* - **nodedim**() `-1`) , where *d* is the dimension of the subcube, and is less than or equal to **nodedim**().

For RX nodes, if a global send specifies its own process (**mypid**()) it does not receive its own message. For CX nodes, however, if a process other than its own is specified, the sending node also receives the message.

RX nodes can only send messages with a *pid* of 0. Since receive processes ignore *pid*, it is possible to receive a message with a *pid* of any value. However, RX nodes interpret *pid* values other than `0` as an error. To avoid this situation, you should set *pid* to `0` for all processes.

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

Errors

`isend: No pid defined`

Use **setpid()** to define a host process id.

`isend: Invalid type`

Use a non-negative number.

`isend: Invalid length`

Use a non-negative number or a length less than or equal to maximum message length.

`isend: Invalid node`

Use **numnodes()** to determine cube size and **myhost()** to determine host number.

`isend: Invalid buffer pointer`

(C only) Specify a pointer which contains the address of a valid data buffer.

`isend: Not enough memory`

Make sure message buffer starts on a boundary which is a multiple of 4 or make more memory available to this process.

`isend: Too many requests`

Use **msgwait()** or **msgdone()** for **isend()** or **irecv()** requests that are outstanding.

See Also

crecv(), csend(), irecv(), isendrecv(), msgcancel(), msgdone(), msgwait()

ISENDRECV() ISENDRECV()

Send a message and receive a reply.

Synopsis

long **isendrecv**(*type, sbuf, slen, tonode, topid, typesel, rbuf, rlen*)

Parameter Declarations

long *type, typesel;*
char **sbuf, *rbuf;*
long *slen, tonode, topid, rlen;*

Return Value

Returns a non-negative message id which is used by **msgcancel()**, **msgdone()**, or **msgwait()**.

Example

The following example illustrates how to use **isendrecv()**. Part 1 runs on node 0 and uses **isendrecv()** to make a remote procedure call to node 1 and post a receive for the message containing the result. **msgdone()** is used to check for message availability when thef result is required. Part 2 runs on node 1, the that receives the message; it performs the receive operation and returns the result to the first procedure.

Part 1 (runs on node 0):

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

#define SIN_REQUEST_TYPE 100
#define SIN_REPLY_TYPE 200
#define SERVER_NODE 1
#define SERVER_PID  0

double rsin();

long iam;
```

ISENDRECV() *(cont.)*

ISENDRECV() *(cont.)*

```
main()
{
double pi, x, angle;
   iam = mynode();
   pi = acos(-1.0);
   angle = pi/2.0;
   x = rsin(angle);
   printf("\nNode %d: x= %lf for angle= %lf radians\n",
   iam,x,angle);
   killcube(-1,0);
}

double rsin(x)
double x;
{
double  result;
long    msgid,done;

   msgid = isendrecv(SIN_REQUEST_TYPE,&x, sizeof(x),SERVER_NODE,
                     SERVER_PID, SIN_REPLY_TYPE, &result,
                     sizeof(result));
   done = msgdone(msgid);
   printf("\n%d: first done = %d\n",iam,done);
   while((done = msgdone(msgid)) != 1) ;
   printf("%d: second done = %d\n",iam,done);
   return result;
}
```

Part 2:

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

#define SIN_REQUEST_TYPE 100
#define SIN_REPLY_TYPE 200

main()
{
int req_node, req_pid;
double x, result;
```

ISENDRECV() *(cont.)*

ISENDRECV() *(cont.)*

```
    for(;;) {
        crecv(SIN_REQUEST_TYPE, &x, sizeof(x));
        req_node = infonode();
        req_pid = infopid();
        result = sin(x);

        sleep(5);
        csend(SIN_REPLY_TYPE, &result, sizeof(result),
              req_node, req_pid);
    }
}
```

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

type identifies the type of the message that you are sending.

sbuf refers to the buffer that contains the message to be sent.

slen is a positive integer that specifies the size (in bytes) of the message that you wish to send. Messages to or from a host process are limited to a maximum of 256K bytes. There is no limit on message length between nodes.

tonode an integer that defines the node(s) to which the message is to be sent.

topid for CX nodes, *pid* is an integer that specifies the id of the process that is to receive the message. Valid *pids* include any integer value. Negative numbers are reserved for system programs. A node *pid* is assigned when you load a process and a host *pid* is assigned with **setpid()**. For RX nodes, *pid* 0 is the only valid process ID (since only a single process model can run on the RX node).

typesel is an integer value that specifies the type(s) of the reply message. See the Discussion for more information.

ISENDRECV() *(cont.)*

ISENDRECV() *(cont.)*

rbuf refers to the buffer where the reply will be stored.

rlen is an integer value that specifies the size of the reply buffer in bytes.

Discussion

Use **isendrecv**() to send a message to a node or host process, and simultaneously post a receive for the reply. When a message arrives whose *type* matches the type(s) specified in the *typesel* parameter, the calling process receives the message, and stores it in *rbuf*. The calling process continues executing instructions while the send-receive operation is occurring.

When it is executed, **isendrecv**() returns a message id. Use **msgdone**() or **msgwait**() to free these message id numbers and to determine whether the **isendrecv**() operation has completed. You cannot use the **info...**() calls to get information about the reply message, because **isendrecv**() does not disturb the information.

The *type* parameter is a user-defined variable typically used to identify the kind of information contained in the message being sent. Types `0` to `999,999,999` are normal types that can be used by any send or receive call, including this one. Types `1,073,741,824` to `1,999,999,999` are special force types available to the user intended specifically for the "sendrecv" 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. If no receive is posted, as when the receiving process has been killed, a message with a force *type* is discarded. In general, bypassing the normal flow control mechanisms causes no problem because **isendrecv**() guarantees that a receive is posted for the message.

NOTE

Types `1,000,000,000` to `1073741823` and `2,000,000,000` and up are used by the system and should be avoided. Their use may produce unpredictable results.

The buffer for the send and receive may be any legal data type. It is recommended that data types match in send and receive operations.

ISENDRECV() *(cont.)*

ISENDRECV() *(cont.)*

If *tonode* is a positive integer, the message is sent to the node with that number. Nodes within a partition are numbered consecutively from 0, and a node number higher than the highest node in the cube causes an error to be returned. To send a message to the host, use the host's id, which is returned by **myhost()**.

If *tonode* is a negative number up to a certain value, it causes "global" send. If node is `-1`, the message is sent to all nodes. Other negative numbers cause the message to be sent to a subcube composed of a set of the nodes surrounding the sender. The dimension of this cube is calculated as (*d* - **nodedim()** `-1`), where *d* is the dimension of the subcube, and is less than or equal to **nodedim()**.

For RX nodes, if a global send specifies its own process (**mypid()**) it does not receive its own message. For CX nodes, however, if a process other than its own is specified, the sending node also receives the message.

RX nodes can only send messages with a *pid* of 0. Since receive processes ignore *pid*, it is possible to receive a message with a *pid* of any value. However, RX nodes interpret *pid* values other than 0 as an error. To avoid this situation, you should set *pid* to 0 for all processes.

The *typesel* parameter describes the type(s) of the message(s) expected to be received. 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. You can use a force type, as described above.
- If *typesel* is `-1`, the first message to arrive for the process that initiated the receive operation will be recognized. After the first message has been received, you can use `-1` again to receive the next message.
- If *typesel* is any negative number other than `-1`, a set of message types will be recognized. Refer to Appendix B for information on building a *typesel* mask.

ISENDRECV() *(cont.)*

ISENDRECV() *(cont.)*

Errors

`isendrecv: No pid defined`

Use **setpid()** to define a host process id.

`isendrecv: Invalid type`

Use a non-negative number.

`isendrecv: Invalid length`

Use a non-negative number or a length less than or equal to maximum message length.

`isendrecv: Invalid node`

Use **numnodes()** to determine cube size or **myhost()** to determine host number.

`isendrecv: Invalid buffer pointer`

Specify a pointer which contains the address of a valid data buffer (C only).

`isendrecv: Not enough memory`

Make sure message buffer starts on a boundary that is a multiple of 4 or make more memory available to this process.

`isendrecv: Received message too long for buffer`

Make sure buffer is large enough to hold the message.

See Also

iprobe(), **irecv()**, **isend()**, **csendrecv()**, **msgcancel()**, **msgdone()**, **msgwait()**

ISEOF() ISEOF()

Tests CFS reads for end-of-file.

Synopsis

long **iseof**(*fildes*)

Parameter Declarations

long *fildes;*

Return Value

`1` if the file pointer is at the end of the file, and `0` if not.

Example

The following example uses **iseof()** to make sure that all the data in a file is processed. Data is read and processed until the file pointer is at the end of the file. The variable *fd* contains the file descriptor returned by a UNIX **open()**, **creat()**, or **dup()** call :

```
char buffer[40];
        .
        .
while(!iseof(fd)) {                    /* Make sure all data in file is processed */
    cread(fd, buffer, sizeof(buffer));
                                                              /* process data */
        .
        .
}
```

Environment

Node

Languages

C, Fortran

ISEOF() *(cont.)*

ISEOF() *(cont.)*

Description of Parameters

fildes the file descriptor.

Discussion

Use **iseof()** to determine whether you have reached the end of the file you are reading. It returns a `1` if the file pointer for the given *fildes* is at the end of the file. If not, the call returns a `0`. The file is designated with the *fildes* parameter, which must be the value returned by the UNIX-compatible **open()**, **creat()**, or **dup()** call when you open a file. The **iseof()** call can be used with all read and write calls, but is most useful with the **cread()** and **iread()** calls.

Errors

`iseof: Bad file number`

Use the file descriptor returned by **open()**, **creat()** or **dup()**.

`iseof: Not implemented on the SRM`

Applies to CFS files only; cannot be applied to host files.

See Also

cread(), **iread()**

IWRITE() IWRITE()

High-speed asynchronous write to a file.

Synopsis

long **iwrite**(*fildes, buf, len*)

Parameter Declarations

long *fildes;*
char **buf;*
long *len;*

Return Value

Returns a non-negative id to be used in **iodone()** or **iowait()**.

Example

This example uses **iwrite()** to write the contents of *buffer* to the file designated by *fd*. The variable *fd* must contain the value returned by the UNIX-compatible **open()**, **dup()**, or **creat()** calls. Then, when the buffer is required, **iowait()** blocks the procedure until the write is complete clears the *id* for use by future reads or writes.

```
int buffer;
long id, fd;
        .
        .
id = iwrite(fd, &buffer, sizeof(buffer));
        .
        .
iowait(id);
                                        /* You can now reuse the buffer. */
        .
        .
```

IWRITE() *(cont.)*

IWRITE() *(cont.)*

Environment

Node

Languages

C, Fortran

Description of Parameters

fildes	the file descriptor.
buf	refers to the buffer containing the data to be written. The buffer can be any valid data type.
len	is an integer value that specifies the size of the buffer in bytes. Transfer size is limited only by the memory available for the buffer.

Discussion

Use **iwrite()** to perform high-speed asynchronous data writes to the file identified with the *fildes* parameter which is the value returned by the UNIX-compatible **open()**, **dup()**, or **creat()** calls.

With **iwrite()** the process continues executing instructions while the write is being done. When you need to use the write buffer in processing, use the **iowait()** call to block further processing until the write is complete, and to free the id number for use by a future read or write (this is required because the number of IDs available is limited, and using them up causes an error). The **iodone()** call will also clear the id when it returns a `1`.

The **iwrite()** call modifies the file pointer immediately, so you can call **lseek()** or another read or write call right away without waiting for the write to complete.

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

Errors

`iwrite: Invalid argument`

Check arguments.

`iwrite: Mixed file operations`

In I/O mode 2 or 3, nodes are attempting different operations (reads and writes) to a shared file. Ensure that nodes are performing the same operation.

`iwrite: I/O error`

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

`iwrite: Not enough core`

Not enough memory on node for operation.

`iwrite: No space left on device`

Not enough space on device to which you are writing. Create more space in file system, or if you have restricted the disks to which you are writing with **restrictvol()**, re-execute **restrictvol()** to include more drives.

`iwrite: Too many requests`

Use **iowait()** for **iread()** or **iwrite()** requests that are outstanding.

See Also

cread(), cwrite(), iread(), iodone(), iowait()
UNIX System V/386 Programmer's Reference: **open(), creat(), dup(), write()**

KILLCUBE() KILLCUBE()

Terminate and clear out process(es).

Synopsis

killcube(*node, pid*)

Parameter Declarations

long *node, pid;*

Return Value

None

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

node is an integer that identifies a particular node on which you want to terminate processes and flush messages or `-1` to indicate all nodes. Nodes in a partition are numbered consecutively from 0. A host id is invalid.

pid for CX nodes, *pid* is an integer value that specifies the process id that you want to terminate and flush. *Pid* must match a *pid* given for a previous load, or it may be `-1` to terminate and flush all processes and messages related to those processes. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on the RX node).

KILLCUBE() *(cont.)*

KILLCUBE() *(cont.)*

Discussion

Use **killcube()** to kill the specified node process(es) and flush messages related to those processes. It is included for convenience and is equivalent to the sequence:

```
killproc(node, pid);
waitall(node, pid);
flushmsg(-1, node, pid);
```

It is not an error to use **killcube()** to kill a nonexistent process.

Errors

`killcube: Invalid node`

Use **numnodes()** to determine cube size and **myhost()** to determine host number.

`killproc:invalid pid`

Use a non-negative number. For RX nodes, *pid* must be 0.

See Also

load(), **killproc()**, **flushmsg()**, **waitall()**

KILLPROC() KILLPROC()

Terminate a process.

Synopsis

killproc(*node, pid*)

Parameter Declarations

long *node, pid;*

Return Value

None

Example

The following statements are examples of how to use the **killproc**() call on CX nodes.

```
killproc(15, 3);          /* Kill process 3 on node 15. */
killproc(-1, 10);         /* Kill process 10 on all nodes. */
killproc(-1, -1);         /* Kill all processes on all nodes. */
```

For RX nodes, *pid* must always be 0.

Environment

Host, Node

Languages

C, Fortran

KILLPROC() *(cont.)*

KILLPROC() *(cont.)*

Description of Parameters

node is an integer value that identifies the node(s) on which you want to terminate a process. A host id is invalid.

pid For CX nodes, *pid* is an integer value that specifies the id of the process that you want to terminate. *Pid* must match a *pid* given for a previous load, or it may be `-1` to terminate all processes. Valid *pids* include any integer value, but negative numbers other than `-1` are reserved for system programs. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on the RX node).

Discussion

Use **killproc**() to kill the node process specified by *pid* and *node*. It has no effect on messages already sent from or in transit to the process. It causes the selected process to terminate, but does not wait for a process to complete, or flush messages related to that process.

If *node* is a positive integer, the specified process(es) on that node is killed.

If *node* is a negative number up to a certain value, it causes a "global" kill. If *node* is `-1`, the specified process on all nodes in the cube is killed. Other negative numbers kill the process(es) in a set of the nodes surrounding the sender. The dimension of this cube is calculated as (*d* - **nodedim**() `-1`), where *d* is the dimension of the subcube, and is less than or equal to **nodedim**().

Errors

`killproc: Invalid pid`

Use a non-negative number. For RX nodes, *pid* must be 0.

`killcube: Invalid node`

Use **numnodes**() to determine cube size and **myhost**() to determine host number.

See Also

load(), **flushmsg**(), **killcube**(), **waitone**(), **waitall**()

KILLSYSLOG() KILLSYSLOG()

Terminate **syslog** process. This is intended for use by host programs and is not available for node programs.

Synopsis

killsyslog()

Parameter Declarations

None

Return Value

None

Environment

Host

Languages

C, Fortran

Description of Parameters

None

KILLSYSLOG() *(cont.)*

KILLSYSLOG() *(cont.)*

Discussion

Use **killsyslog()** to kill the *syslog* process and reset the standard output to the terminal (*/dev/tty*). It can only be used to kill the *syslog* process started by **setsyslog()**; it cannot be used to kill the *syslog* process started by the **syslog** command.

The purpose of the *syslog* process is to make sure that output from host programs goes through the same file server used by the node programs. To change file servers to the current working environment, the *syslog* process must be killed and restarted by invoking **setsyslog()** to make sure the output goes to the correct file server.

Errors

```
syslog: No active syslog exists
```

To start the *syslog* process, use **setsyslog()**.

```
syslog: There is no attached cube
```

Use **getcube()** to get a cube.

See Also

setsyslog(), **newserver()**

LED() LED()

Turn the node board's green LED (light emitting diode) on or off. This is intended for use by node programs and is not available for host programs.

Synopsis

void **led**(*lstate*)

Parameter Declarations

long *lstate;*

Return Value

None

Environment

Node

Languages

C, Fortran

Description of Parameters

lstate an integer value specifying on or off state of the node board's green LED. The possible settings are:

`1` LED is turned on.

`0` LED is turned off.

LED() *(cont.)*

LED() *(cont.)*

Discussion

Use **led()** to take control of the green LED from the node operating system, and turn it on or off, depending upon the value of *lstate*. Behavior is not defined if values other than `0` or `1` are used.

Errors

None

LOAD() LOAD()

Load a node process.

Synopsis

load(*filename, node, pid*)

Parameter Declarations

char **filename;*
long *node, pid;*

Return Value

None

Example

The following statement loads the file named "node" into all nodes in the current partition, and assigns it a process id of `0` on all nodes.

```
      .
      .
load("node", -1, 0);
      .
      .
```

Environment

Host, Node

Languages

C, Fortran

LOAD() *(cont.)*

LOAD() *(cont.)*

Description of Parameters

filename is a pointer to a character string which specifies the pathname of the program file that you want to load into the node. Valid pathnames are ASCII character strings.

node is an integer value that identifies the node you want to load a process into (use `-1` to select all nodes). Nodes within a partition are numbered consecutively from 0. A host id is invalid. A process can use **mynode()** to obtain its node id after it begins execution.

pid is an integer value that specifies the process id assigned to the loaded node process.

Discussion

Use **load()** to load the file designated by *filename* onto the specified *node(s)*, assign *pid* to the process(es), and begin execution. The host id is invalid because this routine does not load processes into the host. The *node* and *pid* specified in the **load** operation are used in send, receive, and kill operations. RX nodes allow only one process, and *pid* must be 0.

On CX nodes, valid *pids* include any integer value, but negative numbers are reserved for system programs. An error results if you specify a *pid* that is already in use by a previously loaded process on the same node. A *pid* may be reused as soon as a process completes. Up to 20 processes per CX node are allowed. A process can use **mypid()** to obtain its process id after it begins execution.

In an RX node, when a node program executes a **fork()**, a new process is created on any available node. An available node is a node that does not have a process running on it. If a process has been loaded with **load()** and has terminated, then that node is unavailable until it has been waited on with the **waitcube** command, **waitall()**, or **waitone()**

LOAD() *(cont.)* LOAD() *(cont.)*

Errors

`load: There is no attached cube`

Use **getcube**() to allocate a cube and **attachcube**() to change cubes within your program.

`load: Invalid pid`

The *pid* must not be negative. For RX nodes, *pid* must be `0`.

`load: Invalid node`

Node number must be in range. Use **numnodes**() to determine cube size.

`load: Out of process slots`

Use fewer processes.

`load: Pid already in use`

Use a different *pid*.

`load: Not enough memory`

Reduce the size of the process.

`load: Invalid loader record`

Internal error, try again. System administrator may have to **bootcube**.

`load: Invalid loader message`

Internal error, try again. System administrator may have to **bootcube**.

`load: No such file or directory`

Correct the *filename*.

LOAD() *(cont.)*

LOAD() *(cont.)*

```
load: Invalid object file
```

Specify a loadable file.

```
load: There are no cubes allocated
```

Use **getcube()** to allocate a cube.

```
load: No pid defined
```

Use **setpid()** to define a host process id.

See Also

killcube(), mynode(), mypid()

LSIZE() LSIZE()

Increase the size of a file.

Synopsis

long **lsize**(*fildes, offset, whence*)

Parameter Declarations

long *fildes;*
long *offset;*
long *whence;*

Return Value

The file size in bytes is returned if the call is successful; otherwise, it returns a `-1`.

Example

The following statement of **lsize()** increases the size of the file designated by the file descriptor variable *fd* (the value returned by the UNIX **open()**, **creat()**, or **dup()** call) to one million bytes.

```
        .
        .
lsize(fd, 1000000, 0);          /* Increase file size to 1 million bytes.*/
```

Environment

Node

Languages

C, Fortran

LSIZE() *(cont.)*

LSIZE() *(cont.)*

Description of Parameters

fildes the file descriptor returned by the **open()**, **creat()**, or **dup()** UNIX system calls.

offset a long integer value that affects the current size of the file depending upon the value of *whence*.

whence an integer value of `0`, `1`, or `2` that determines how the offset value is used to set the size of the file. These values have the following effects:

`0`	sets the file size to the greater of the current size or *offset*.
`1`	sets the file size to the greater of the current size or the current location of the file pointer plus *offset*.
`2`	sets the file size to the greater of the current size or the current size plus *offset*.

If the new size specified by *offset* and *whence* will be greater than the available disk space, **lsize()** silently allocates the available space and returns the actual new size.

Discussion

The **lsize()** call changes the size of the file designated by the *fildes* parameter, which must be the value returned by the **open()**, **dup()**, or **creat()** call. The *offset* parameter is a value in bytes that, depending upon the value of the *whence* parameter, is used to determine the resulting size of the file.

The major use of **lsize()** is to ensure that enough space is allocated before you do any performance-sensitive calculations. **lsize()** ensures that the disk blocks are allocated to the file from the starting position to the ending position (as determined by *whence* and *offset*). If allocation has been restricted by **restrictvol()**, any new disk blocks allocated will come from the restricted set.

If the call is successful, the call returns the new size of the file in bytes. If the call is not successful, the call will fail and return a `-1`. Using **lsize()** does not affect the position of the file pointer.

For changing the size of large files, use the **esize()** call.

LSIZE() *(cont.)*

LSIZE() *(cont.)*

Errors

`lsize: Invalid size`

Cannot be used in extended file sizes. Use **esize()**.

`lsize: Bad file number`

Use the file descriptor returned by **open()**, **creat()**, or **dup()**.

`lsize: No space left on device`

Not enough space on device to which you are writing. Create more space in file system, or if you have restricted the disks to which you are writing with **restrictvol()**, re-execute **restrictvol()** to include more drives.

`lsize: Not enough core`

Not enough memory on node for operation.

`lsize: Not implemented on the SRM`

Cannot use on the SRM.

See Also

restrictvol(), **esize()**
UNIX System V/386 Programmer's Reference: **open()**, **creat()**, **dup()**, **lseek()**

MASKTRAP() MASKTRAP()

Enable or disable a receive trap. Intended for use by node programs and is not available for host programs.

Synopsis

long **masktrap**(*state*)

Parameter Declarations

long *state;*

Return Value

An integer that is the previous value of *state*.

Example

The following example uses **masktrap()** to prevent critical processing from being interrupted by the execution of the handler procedure called at the completion of the **hrecv()**.

```
hrecv(typesel, buf, len, proc);      /* Initiate receive and set up handler */
masktrap(1);                          /* Temporarily disable receive trap */
    .                                     /* Do critical processing here */
    .
masktrap(0);                          /* Re-enable receive trap */
```

Environment

Node

Languages

C, Fortran

MASKTRAP() *(cont.)*

MASKTRAP() *(cont.)*

Description of Parameters

state Either `1` or `0: 1` blocks receive traps; `0` enables receive traps.

Discussion

Use **masktrap()** to protect critical processing from being interrupted by the execution of the handler procedure called when an **hrecv()**, **hsend()**, or **hsendrecv()** completes. If these complete while traps are masked, the trap will be held until unmasked. Assigning the *state* parameter a value of `1` blocks the receive trap until **masktrap()** is called again, assigning *state* a value of `0`.

Errors

None

See Also

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

MCLOCK() MCLOCK()

Return elapsed time.

Synopsis

unsigned long **mclock()**

Parameter Declarations

None

Return Value

An unsigned long integer containing a relative time value.

Example

The following example illustrates how to use **mclock()** to determine elapsed time of a process.

```
unsigned long start_time, end_time, elapse_time;
    .
    .
start_time = mclock();                          /* find starting time */
                                                /* run calculations */
    .
    .
end_time = mclock();                            /* find ending time */
elapse_time = end_time - start_time;            /* find elapsed time */
```

Environment

Host, Node

Languages

C, Fortran

MCLOCK() *(cont.)*

MCLOCK() *(cont.)*

Description of Parameters

None

Discussion

Use **mclock**() as a simple mechanism to measure time intervals; it returns the value of a counter that reflects relative time in milliseconds. Programs that use **mclock**() should obtain an initial time value and interpret all other time values relative to this initial value.

When **mclock**() is called from a host program, the relative time (in milliseconds) it returns counts only the time that the UNIX system was actually used to execute your host program, any child processes the host program creates, and system operations related to the host program.

On the nodes, the value returned by **mclock**() represents the actual elapsed milliseconds since the node was initialized with the **bootcube** command.

It is recommended that you do not use **mclock**() to synchronize different aspects of process execution because every node maintains its own clock value and values on different nodes may vary after initialization.

On both the host and nodes, the **mclock**() value rolls over approximately every 50 days.

Errors

None

MSGCANCEL() MSGCANCEL()

Cancel an asynchronous send or receive operation.

Synopsis

msgcancel(*id*)

Parameter Declarations

long *id;*

Return Value

None

Example

The following example uses **msgcancel**() to cancel the receipt of a message and clear its id if an error occurs.

```
char buf[];
long msg_id, type;
     .
     .
msg_id = irecv(type, buf, sizeof(buf));
     .
     .                          /* Perform tasks while waiting for receive to complete. */
if (error)                                   /* if error condition , cancel message */
    msgcancel(msg_id);
```

Environment

Host, Node

Languages

C, Fortran

MSGCANCEL() *(cont.)*

MSGCANCEL() *(cont.)*

Description of Parameters

id is the message id returned when you invoke an **isend()** or **irecv()** operation.

Discussion

Use **msgcancel()** to cancel an asynchronous send or receive operation. The *id* parameter identifies the operation to cancel. *Id* must correspond to the value returned by a previous **isend()** or **irecv()**. When **msgcancel()** returns, you do not know whether the send or receive operation completed, but you do know that the operation is no longer active, that the buffer may be reused. You also know that the message id is no longer valid, and so is available for use by further asynchronous send and receive operations, and cannot be used further in **msgdone()** or **msgwait()**.

Errors

```
msgcancel: Invalid message id
```

Use message id returned by **irecv()** or **isend()**.

```
msgcancel: No pid defined
```

Use **setpid()** to define a host process *id*.

See Also

irecv(), **isend()**

MSGDONE() MSGDONE()

Determine whether an asynchronous send or receive operation is complete.

Synopsis

long **msgdone**(*id*)

Parameter Declarations

long *id;*

Return Values

1	indicates that the message is complete and that the associated message buffer is available for reuse, in the case of a send operation, or that the buffer contains valid data in the case of a receive operation. You cannot call **msgdone()** or **msgwait()** with the same id parameter after **msgdone()** returns 1 because the send or receive operation is complete and the id is no longer valid.
0	indicates that the send or receive message is not yet complete. In this case, you can use the same id again until the send or receive operation completes.

Example

This example uses **msgdone()** to determine whether an **isend()** has completed sending the message. If so, it reuses the message buffer, otherwise it does not use it because the message is in the buffer and has not yet been sent.

```
char buf[];
long msg_id, type;
    .
    .
msg_id = isend(type, buf, sizeof(buf), 1, 1);
    .
    .
if (msgdone(msg_id)) {
                                        /* You may now reuse the message buffer. */
} else {
                                        /* Message buffer still needs to be protected. */
}
```

MSGDONE() *(cont.)*

MSGDONE() *(cont.)*

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

id is the message id returned when you invoke an **isend()** or **irecv()** operation.

Discussion

Use **msgdone()** to determine whether an **isend()** or **irecv()** operation is complete. When **msgdone()** returns a `1`, the **isend()** or **irecv()** operation is complete. This means that the associated message buffer is available for reuse in the case of an **isend()**, or the buffer contains valid data in the case of an **irecv()**.

It also releases the id number used by the asynchronous message passing call to which it refers. You must use either **msgdone()** or **msgwait()** after an asynchronous message passing call to ensure that the id is cleared. You cannot call **msgdone()** or **msgwait()** with the same *id* parameter after **msgdone()** returns `1` because the send or receive operation is complete and the id is no longer valid. The *id* parameter identifies the operation you are checking status on. The *id* must correspond to the value returned by a previous **isend()** or **irecv()**.

When **msgdone()** returns `1` for a receive operation, you can use the **info...()** calls get more information about the message.

MSGDONE() *(cont.)*

MSGDONE() *(cont.)*

Errors

`msgdone: Invalid message id`

Use message id returned by **irecv()** or **isend()**.

`msgdone: No pid defined`

Use **setpid()** to define a host process *id*.

`msgdone: Received message too long for buffer`

Make sure the buffer is large enough to hold the message.

See Also

infocount(), **infonode()**, **infopid()**, **infotype()**, **irecv()**, **isend()**, **msgwait()**, **msgcancel()**

MSGWAIT() MSGWAIT()

Wait for completion of an asynchronous send or receive operation.

Synopsis

msgwait(*id*)

Parameter Declarations

long *id;*

Return Value

None

Example

This example uses **msgwait()** to stop further execution of the process until an **irecv()** posted earlier has been completed. It also clears the *id* for use by subsequent sends and receives.

```
int result;
long msg_id, type;
     .
     .
msg_id = irecv(type, &result, sizeof(result));
     .                                  /* Perform tasks that do not use 'result'. */
     .
msgwait(msg_id);
                                        /* Now you can use the value of 'result'. */
```

Environment

Host, Node

Languages

C, Fortran

MSGWAIT() *(cont.)*

MSGWAIT() *(cont.)*

Description of Parameters

id is the message id returned when you invoke an **isend()** or **irecv()**.

Discussion

Use **msgwait()** to block a process until an **isend()** or **irecv()** operation is complete, and when the operation is complete, it releases the *id* number used by the asynchronous message passing call to which it refers. Because a limited number of message id's are available, you must use either **msgdone()** or **msgwait()** after an asynchronous message passing call. This ensures that you do not run out of message IDs, which would cause an error. The *id* parameter identifies the operation that you are waiting for. The *id* must correspond to the value returned by a previous **isend()** or **irecv()** call.

When the message is complete, **msgwait()** returns and the associated message buffer is available for reuse, in the case of a send operation, or the buffer contains valid data, in the case of a receive operation.

When **msgwait()** returns after waiting for an **irecv()** to complete, you can use **info** to get more information about the message. You cannot use the *id* parameter again because the send or receive operation is complete.

Errors

`msgwait: Invalid message id`

Use message id returned by **irecv()** or **isend()**.

`msgwait: No pid defined`

Use **setpid()** to define a host process *id*.

`msgwait: Received message too long`

Make sure the buffer is large enough for buffer to hold the message.

See Also

infocount(), **infonode()**, **infopid()**, **infotype()**, **irecv()**, **isend()**, **msgcancel()**, **msgdone()**

MYHOST() MYHOST()

Obtain the id of the host machine.

Synopsis

long **myhost**()

Parameter Declarations

None

Return Value

The host id is returned.

Example

The following example uses **myhost**() to obtain the host id for a **csend**() operation from a node to the host.

```
char msg[100];
long host_id;
    .
    .
host_id = myhost();
csend(10,msg,sizeof(msg),host_id,1);      /* Send message to host process */
```

Environment

Host, Node

Languages

C, Fortran

MYHOST() *(cont.)*

MYHOST() *(cont.)*

Description of Parameters

None

Discussion

Use **myhost()** to return the host id of the caller's host machine for use in send and receive operations. For host processes, **myhost()** returns the same id as **mynode()** (node id of the calling process). In a **D4** cube, with cubes numbered 0-15, **myhost()** would return `16`.

Errors

```
myhost: No pid defined
```

Use **setpid()** to define a host process id.

See Also

csend(), isend(), mynode()

MYNODE() MYNODE()

Obtain the node id of the calling process.

Synopsis

long **mynode**()

Parameter Declarations

None

Return Value

The node id of the process that initiated the routine is returned.

Example

The following example uses **mynode**() to provide the node id to **killproc**().

```
long this_node;
    .
    .
this_node = mynode();                    /*Obtain my logical node id */
killproc(this_node, 1);                  /* Kill process 1 on this node */
```

Environment

Host, Node

Languages

C, Fortran

MYNODE() *(cont.)*

MYNODE() *(cont.)*

Description of Parameters

None

Discussion

Use **mynode()** to return the node id of the calling process. Nodes within a partition are numbered consecutively from 0. The host's id is the return value of **myhost()**. The node id is used in send and kill operations.

Errors

```
mynode: No pid defined
```

Use **setpid()** to define a host process id.

See Also

myhost(), **mypid()**, **setpid()**, **numnodes()**

MYPID() MYPID()

Obtain the NX/2 process ID of the calling process.

Synopsis

long **mypid**()

Parameter Declarations

None

Return Value

The process id is returned. For RX nodes, *pid* is always 0.

Example

This example uses **mypid**() to return the current process id to be used in a **csend**().

```
char buf[100];
long type;
    .
    .          /* Send message to the processes with the same pid as mine on all nodes */
    .
csend(type, buf, sizeof(buf), -1, mypid());
```

Environment

Host, Node

Languages

C, Fortran

MYPID() *(cont.)*

MYPID() *(cont.)*

Description of Parameters

None

Discussion

Use **mypid()** to returns the cube process id of the calling process. On a node, this is the *pid* that was supplied when the process was loaded. On the host, this *pid* is determined by **setpid()**. (This cube process id should not be confused with the UNIX process id.) The *pids* are used in send and kill operations. For RX nodes, the process ID is always 0.

Errors

```
mypid: No pid defined
```

Use **setpid()** to define a host process id.

See Also

csend(), csendrecv(), flushmsg(), isend(), isendrecv(), load(), killcube(), killproc(), setpid()

NEWSERVER() NEWSERVER()

Start a new file server for the specified cube.

Synopsis

newserver(*cubename*)

Parameter Declarations

char **cubename;*

Return Values

None

Example

Before running a program named **host**, enter the **getcube** command at the system prompt. With no redirection, the node output is the standard output (*stdio*) of the **getcube** command. However, you intend to run your program twice, and you want the node output in two separate files. If you do not use **newserver()** within the program, the node output will go to standard output of the **getcube** command. If you do use **newserver()**, the node output will go to the standard output of the host program, which is the file indicated on the invocation line.

At the system prompt:

```
%getcube
%host > file1
%host > file2
```

Within the host program:

```
static char buffer[] = "defaultname";
     .
     .
newserver(buffer);          /* Now the output will go to the standard output of */
                                                        /* the host program */
```

NEWSERVER() *(cont.)*

NEWSERVER() *(cont.)*

Environment

Host

Languages

C, Fortran

Description of Parameters

cubename refers to the name of the cube, assigned with **getcube**. Valid names include any ASCII string, 15 characters or less. Names longer than 15 characters are truncated to 15 characters. If the parameter is null or points to a null string, the currently attached cube is used. If you specify a cube that has not been allocated with **getcube** or there is no current attached cube, an error message results and the process terminates. Use the **cubeinfo** command or call to find out the names of allocated cubes.

Discussion

When you allocate a cube with **getcube**, a file server to handle I/O for the nodes is automatically created. If you want to redirect the output of your node process to a different file or device at program invocation, you must use **newserver()** within your program to allow the output to go to the desired place. The **newserver** command actually kills the original file server and starts a new one. When you issue **newserver()**, changes that you make to your environment are passed to the new file server.

For information on redirection, refer to the *UNIX System V User's Guide*.

NEWSERVER() *(cont.)*

NEWSERVER() *(cont.)*

Errors

`newserver: Cubename does not exist`

Use **cubeinfo** to determine cube names.

`newserver: Commser not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with the **rebootcube** command.

`newserver: Lifeline not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with the **rebootcube** command.

See Also

cubeinfo(), getcube()

NODEDIM() NODEDIM()

Returns the dimension of the allocated cube.

Synopsis

long **nodedim**()

Parameter Declarations

None

Return Value

An integer from `0` to `7` representing the dimension of the cube.

Example

The following example uses **nodedim**() to return the size of the cube obtained by **getcube**().

```
long alphadim;
    .
    .
getcube ("alpha", "32", NULL, 0);
alphadim = nodedim();                    /* Dimension of alpha is 5 */
```

Environment

Host, Node

Languages

C, Fortran

NODEDIM() *(cont.)*

NODEDIM() *(cont.)*

Description of Parameters

None

Discussion

Use **nodedim**() to return the dimension of the allocated cube. For example, the dimension of a 64-node cube is 6 ($2^6 = 64$). If the number of nodes allocated is 8, **nodedim**() returns 3. Use **numnodes**() to return the number of allocated nodes.

Errors

```
nodedim: No pid defined
```

Use **setpid**() to define a host process id.

See Also

getcube(), **numnodes**()

NUMNODES() NUMNODES()

Obtain the number of nodes in the cube.

Synopsis

long **numnodes()**

Parameter Declarations

None

Return Value

An integer in the range `1-128`, representing the number of nodes in a cube.

Example

The following example gets two cubes, and then uses **numnodes()** to find the highest-numbered node in the cube "beta" and the total number of nodes in the cube "alpha."

```
long last_node, total_nodes;
    .
    .
getcube("alpha", "d3m4", "saxon", 1);
getcube("beta", "32vx", "viking", 1);
setpid(1);

last_node = numnodes() - 1;          /* The last node on cube "beta" is 31. */

attachcube("alpha");

total_nodes = numnodes();       /* Total number of nodes on cube "alpha" is 8. */
```

NUMNODES() *(cont.)*

NUMNODES() *(cont.)*

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

None

Discussion

Use **numnodes()** to return the number of nodes allocated to the cube. For instance, if you allocate a cube of size 16 with **getcube()**, **numnodes()** will return `16` (2^4). Use **getcube()** to assign a number of nodes to an application.

Errors

```
numnodes: No pid defined
```

Use **setpid()** to define a host process id.

See Also

getcube(), **nodedim()**

PLOGON(), PLOGOFF() PLOGON(),PLOGOFF()

Log process related information such as cube allocation, cube deallocation, process initialization, and process completion. Requires root privilege to run.

Synopsis

plogon(*append, noprocs,query, logfile*)
plogoff()

Parameter Declarations

plogon()

short *append, noprocs,query*;
char **logfile*;

plogoff()

None

Return Value

None

Example

This example turns on logging to the file */usr/joelc/mylog*. Booting the iPSC system, turning on/off logging, cube allocation, cube deallocation, process initialization, and process completion are logged.

```
static char logfile[] = "/usr/joelc/mylog";
      .
      .
plogon(0,0,0,logfile);                    /* logging turned on */
      .
      .
plogoff();                                /* logging turned off */
      .
      .
```

PLOGON(), PLOGOFF() *(cont.)*

PLOGON(), PLOGOFF() *(cont.)*

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

append is an integer. When it is not 0, the logging information is appended to *logfile* (if *logfile* exists). When it is 0, the logging information overwrites an existing *logfile*.

noprocs is an integer. When it is not 0, no process loading or termination information is logged. When it is 0, process initialization, and process completion are logged as well as execution of **bootcube**, **plogon**, **plogon()**, **plogoff()**, **plogoff**, **getcube**, **getcube()**, **relcube**, and **relcube()**.

query is an integer that when not 0 causes **plogon()** to to return the name of the current logfile in the parameter *logfile*. No other action is performed.

logfile is a string that identifies the pathname of the log file. If it is null, the default */usr/ipsc/log/process.log* is used.

Discussion

plogon() turns on the logging function, and **plogoff()** turns it off. If already logging to a file, **plogon()** continues logging to the same file or to a new one if a different file is specified.

The process executing **plogon()** or **plogoff()** must be running at root privilege. If the process does not have root privilege, it aborts when it encounters the call.

PLOGON(), PLOGOFF() *(cont.)*

PLOGON(), PLOGOFF() *(cont.)*

Errors

`plogon: Permission denied`

Only the superuser can run **plogon()**.

`plogoff: Permission denied`

Only the superuser can run **plogoff()**.

`plogoff: No active log file exists`

plogoff() cannot find a log function to terminate.

RELCUBE() RELCUBE()

Release a cube.

Synopsis

relcube(*cubename*)

Parameter Declarations

char **cubename;*

Return Value

None

Example

The following example gets a cube with **getcube()**, and later uses **killcube()** to terminate any remaining processes, and finally uses **relcube()** to release the cube.

```
getcube("big", "128", NULL, 0);
     .
     .                                    /* Clean up and release the cube. */
killcube(-1, -1);
relcube("big");
```

Environment

Host

Languages

C, Fortran

RELCUBE() *(cont.)*

RELCUBE() *(cont.)*

Description of Parameters

cubename refers to a character string that is the name of the cube that you want to release. You name a cube when you invoke **getcube**. Valid names include any ASCII string, 15 characters or less. Names longer than 15 characters are truncated to 15 characters. If the parameter is null or points to a null string, the currently attached cube is used. If you specify a cube that has not been allocated with **getcube** or if there is no cube attached, an error message results and the process terminates. Use **cubeinfo()** to find out the names of allocated cubes.

Discussion

Use **relcube()** to release a cube that you allocated when you invoked **getcube()**. It also kills the file server and any processes attached to *cubename* and any node processes running on the cube.

Errors

`relcube: Cubename does not exist`

Use **cubeinfo** to determine cube names.

`relcube: Commser not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with the **bootcube** command.

`relcube: Lifeline not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with the **bootcube** command.

See Also

cubeinfo(), getcube()

RESTRICTVOL() RESTRICTVOL()

Restrict or return the disk volumes to which a file can be allocated.

Synopsis

long **restrictvol**(*fildes, nvol, vollist*)

Parameter Declarations

long *fildes;*
long *nvol;*
long *vollist [ ];*

Return Value

If *nvol* is a positive integer or `0`, and the volume restriction is set successfully, the return value is `0`. If the restriction is not successful, a `-1` is returned and *errno* is written.

If *nvol* is a negative number, the call returns the number of volumes currently available for allocation.

RESTRICTVOL() *(cont.)*

RESTRICTVOL() *(cont.)*

Example

The following example shows a call to **restrictvol()** with `-1` for the *fildes* parameter and one volume to ensure that the file header is created on disk volume 1. The second call to **restrictvol()** restricts the file specified by *fd*, the file descriptor returned by the **open()** call, to disk volumes 1 and 4. Later, following an **lsize()** call increasing the file size, the programmer removes the restriction. After data is written to the file, the programmer uses **restrictvol()** to find out which volumes have actually been used (a maximum array size for *vollist* is used in the example).

```
long fd;
long volumelst[2];
long newvollst[128];
long numvols, status;

volumelst [0] = 1;
volumelst [1] = 4;
status = restrictvol(-1, 1, &volumelst[0]);
     .
     .
fd = open("/cfs/mydata", O_CREAT|O_RDWR, 0644);
status = restrictvol(fd, 2, volumelst);
       /* When you write data to this file, allocation will be restricted to volumes 1 and 4 */
     .
     .


                        /* Remove previous volume restriction following an lsize() call */
status = restrictvol(fd,0,0);
                                    /* Processing follows that includes writes to the file */
     .
     .
                        /* Use the call to return the number of volumes used and to store */
                                                /* the list of volumes in newvollist */
numvols = restrictvol(fd, -1, newvollist);
     .
     .
```

RESTRICTVOL() *(cont.)*

RESTRICTVOL() *(cont.)*

Environment

Node

Languages

C, Fortran

Description of Parameters

fildes — a positive long integer value which must be the file descriptor returned by the **open()**, **creat()**, or **dup()** call when you open a file, or `-1` which indicates that the file header created by the next **open()**, **creat()**, or **dup()** call should be restricted.

nvol — The value of this parameter determines the effect of the call.

- If *nvol* is a positive integer, **restrictvol()** expects to find the values for an array of that size in the *vollist* parameter, and restricts subsequent file allocation to the specified volume(s).
- If *nvol* is `0`, the file may be allocated to all volumes in the system (no restrictions). This is the default.
- If *nvol* is a negative number, the current volume list is put into the variable *vollist*, and the number of volumes to which the specified file is allocated is returned. *vollist* must be large enough to hold all information.

vollist — An array of integers containing the volume numbers to which file allocation is to be restricted if *nvol* is positive. If *nvol* is negative, the volume numbers to which the specified file has been allocated are put into the array. If *nvol* is `0`, the *vollist* parameter is not used.

RESTRICTVOL() *(cont.)*

RESTRICTVOL() *(cont.)*

Discussion

The **restrictvol()** call can be used either to restrict the volumes to which the blocks of a file can be allocated, or to return the volumes available for allocation. Volume restriction must be done on a file-by-file basis, and requires the file descriptor returned by the **open()**, **creat()**, or **dup()** call.

When there is no restriction, file blocks are allocated in a round-robin fashion to volumes in order by number. Volumes with no free space are skipped. CFS allocates file blocks to volumes when a file is opened, when it is written beyond the current end of the file or in a previously unallocated space, and when the **lsize()** call is used to extend the file size.

To restrict the number of volumes, the *nvol* parameter must be set to the number of volumes that you want to use for the file, and the *vollist* parameter must contain the volume number(s) that you want to use. If the restriction is successful, the call returns a `0`.

To return the volumes that a file is residing on, the *nvol* parameter must be any negative number, and the *vollist* parameter must be an empty array at least the size of the number of drives on your system, to avoid error. In this case, **restrictvol()** returns the number of volumes used, and puts the volume numbers into the *vollist* array.

Errors

`restrictvol: Invalid argument`

Check arguments.

`restrictvol: Bad file number`

Use the file descriptor returned by **open()**, **creat()**, or **dup()**.

See Also

lsize()
UNIX System V/386 Programmer's Reference: **open()**, **creat()**, **dup()**

SETIOMODE() SETIOMODE()

Sets the I/O mode as it performs a global synchronization operation.

Synopsis

setiomode(*fildes, mode*)

Parameter Declarations

int *fildes;*
int *mode;*

Return Value

None

Example

The following example uses **setiomode()** to set the I/O mode to 2. In I/O mode 2, reads and writes are global operations that must be performed by all nodes at once, and are performed in order of the node number. Therefore, from the point of the **setiomode()** call, all nodes must have the same sequence of reads and writes:

```
    .
    .                                   /* Create a file on CFS named mydata with open() */
fd = open("/cfs/mydata", O_CREAT|O_RDWR, 0644);
                                        /* set up a buffer named 'buffer' with data */
    .
    .
setiomode(fd, 2);                       /* set the I/O mode of the file to 2 */
                                        /* all nodes write data in buffer to file */
cwrite(fd, buffer, sizeof(buffer));
                                                        /* Do other processing */
    .
    .
lseek(fd, 0, 0);                        /* all nodes move pointer to front of file */
cread(fd, buffer, sizeof(buffer));
```

SETIOMODE() *(cont.)*

SETIOMODE() *(cont.)*

Environment

Node

Languages

C, Fortran

Description of Parameters

fildes the file descriptor.

mode an integer from 0 through 3 that sets the I/O mode of the designated file. Briefly, these modes are defined as follows:

0 **Individual file pointer.** Each process has its own file pointer. Nodes can read or write information anywhere in the file. This is the default mode for all shared files and treats processes on different nodes in the same way that UNIX treats different processes.

1 **Common file pointer.** There is one file pointer for all nodes. I/O requests are serviced or on a first-come, first-serve basis. Nodes can read and write at any time but reads and writes to the file are done one at a time using a common file pointer.

2 **Synchronized, common file pointer, variable length requests.** There is one file pointer for all nodes. I/O requests must be read by all nodes at once, and reads and writes are performed in node order. Records may be variable length.

3 **Synchronized, common file pointer, fixed length requests.** Same as mode 2 except that the amount of data read or written in each of the global requests must be the same.

SETIOMODE() *(cont.)*

SETIOMODE() *(cont.)*

Discussion

The **setiomode()** call sets the I/O mode of the file specified by the *fildes* parameter which must contain the value returned by the **open()**, **creat()**, or **dup()** UNIX system calls. At the same time, **setiomode()** performs a global synchronizing operation like that of the **gsync()** call. Every node must use the same *fildes*, and the file pointer must be in the same position in the file for each node at the time the call is made.

All files start in mode 0. Standard input, standard output, and standard error output are in mode 1 and the mode cannot be changed.

The **iomode()** call can be used to return the current mode of a given file descriptor.

There are four possible I/O modes (0-3), described in detail in the following paragraphs.

Mode 0: Individual File Pointer

Mode 0 is the mode in which all shared files are opened. Shared files are those that more than 1 node accesses in a process. In this mode, all nodes may read and write to any part of the file at any time. This is the fastest mode, except for the time required for seeks. If two nodes write to the same place in the file, the data written by the second node to write overwrites the data already there. Therefore, this mode is most useful in situations where all nodes are only reading a data file or where each node is responsible for data in a specific area of a file. This mode has the highest performance.

Mode 1: Common File Pointer

In this mode, reads and writes to the file can come from any node at any time, and requests are serviced on a first-come, first-serve basis. There is a common file pointer. Because requests from nodes can be performed in any order, the data in the file may not have a consistent order from run to run. This mode is most useful for log files. This mode has lower performance. *stdin*, *stdout*, and *stderr* are always in mode 1.

Mode 2: Synchronized, Common File Pointer, Variable Length Requests

Mode 2 treats reads and writes as global operations, in which all nodes must read or write at once. The amount of data read or written may, however, vary from node to node. For writes, data is written to the file in order by node number. Reads are also done in order by node number. This mode treats reads and writes as synchronizing to the extent that a second request by any one node is blocked until all other nodes have performed the first operation. The only valid use for the **lseek()** or **eseek()** call in this mode is for all nodes to seek to the same position in the file prior to a request. If nodes seek to different positions, an error is returned immediately. This mode offers lower performance.

SETIOMODE() *(cont.)*

SETIOMODE() *(cont.)*

Mode 3: Synchronized Ordered Mode, Fixed Length Requests

Mode 3, like mode 2, requires that all nodes must read or write at once. The difference is that for any given read or write, all nodes must do the same operation with the same buffer size. For writes, data is written to the file in order by node number. Reads are also done in order by node number. The only valid use for the **lseek()** or **eseek()** call in this mode is for all nodes to seek to the same position in the file prior to a request. If nodes seek to different positions or read or write different amounts of data, an error is returned eventually. This is a high performance mode, with the best performance from the following modes (in order): 0, 1, 3, and 2.

Errors

`setiomode: Bad file number`

Use a file descriptor on **open()**, **creat()**, or **dup()**.

`setiomode: Not implemented on the SRM`

Can be executed by the node only; not available on the host.

`setiomode: Bad io mode number`

I/O mode must be set to `0, 1, 2,` or `3`.

`setiomode: File is not synchronized`

In I/O modes 2 or 3, all nodes must set the file pointer to the same location.

See Also

cread(), **cwrite()**, **iomode()**, **iread()**, **iwrite()**
UNIX System V/386 Programmer's Reference: **open()**, **creat()**, **dup()**

SETIPHOST() SETIPHOST()

Set the designated socket node to be the current iphost.

Synopsis

int **setiphost**(*iphostname)*

Parameter Declarations

char **iphostname;*

Return Value

If successful, **setiphost()** returns 0; if **setiphost()** fails, it exits. The underscore version **_setiphost()** returns the slot number of the socket node if successful. This is the number in the *cubeconf* file. The underscore version returns -1 if it fails.

Example

This example makes the socket node called *myiphost* the current iphost.

```
setiphost("myiphost");
```

Environment

Node

Languages

C

SETIPHOST() *(cont.* SETIPHOST() *(cont.)*

Description of Parameters

iphostname specifies the name of the socket node that you want to be the current iphost. This name is defined in the */etc/hosts* file on the SRM.

Discussion

When a node program is loaded, the NX/2 operating system assigns a socket node as its iphost. It chooses the socket node that has been assigned the least number of times. If you want to override this designation, use **setiphost()**.

For example, consider a node server waiting at an **accept()** for a connection from a program running on an ethernet workstation. The program on the workstation must specify the iphost of its node server.

setiphost() is unnecessary no sense if the iPSC/860 system has only one socket node.

Errors

```
setiphost: No such device
```

The call cannot find an entry in */etc/hosts* that corresponds to a working socket node.

See Also

getiphosts()

SETPID() SETPID()

Sets the process id of a host process.

Synopsis

setpid(*pid*)

Parameter Declarations

long *pid;*

Return Value

None

Example

The following example sets the host *pid* for the cube to which you are attached.

```
getcube("alpha", "16vx", NULL, 0);
getcube("beta", "16vx", NULL, 0);

attachcube("alpha");
setpid(0);

attachcube("beta");
setpid(99);
```

Environment

Host

Languages

C, Fortran

SETPID() *(cont.)*

SETPID() *(cont.)*

Description of Parameters

pid specifies the process id that you are assigning to a host process. Valid *pids* include any integer value, but negative numbers are reserved for system programs.

Discussion

Use **setpid()** to set the process id of the host process. You must call **setpid()** before using any of the message-passing calls such as send and receive operations.

You must allocate a cube before you call **setpid()**. If you change the current cube inside your user process by calling **getcube()** or **attachcube()**, you must call **setpid()**. If you attach to a cube you were previously attached to, **setpid()** must be called. The *pid* used previously can be reused in **setpid()**. Use **mypid()** to obtain the *pid* of a host process after it has been set. **setpid()** is not valid for a node process because the *pid* for a node process is set in **load()**, although using it on a node process does not cause an error.

Errors

```
setpid: Pid already in use
```

Select another *pid*.

```
setpid: Pid already set
```

Do not call **setpid()** again.

```
setpid: There is no attached cube
```

Use **cubeinfo** to find out which cubes you own.

```
setpid: Internal cube usage limit
```

Limit of 10 cubes may be allocated. Try again later when system is not as busy.

SETPID() *(cont.)*

SETPID() *(cont.)*

```
setpid: Commser not responding
```

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with **bootcube**.

```
setpid: Lifeline not responding
```

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with **bootcube**.

See Also

getcube(), attachcube(), mypid()

SETSYSLOG() SETSYSLOG()

Start the *syslog* process.

Synopsis

setsyslog(*stdfd*)

Parameter Declarations

long *stdfd;*

Return Values

None

Example

The following example starts the *syslog* process to redirect host process output to *stdio* before loading a program. Assume that when you allocate the cube, you redirect node output to the file *logfile*.

```
%getcube -t1rx > logfile
```

The host program issues a **setsyslog(1)** before loading the node program. The node program just writes to its standard output, which in this case is *logfile*. The host program is as follows:

```
#include <cube.h>
#define NULL 0
main()
{
   setpid(100);
   printf("Before setsyslog\n");
   setsyslog(1);
   printf("After setsyslog\n");
   load("node",-1,0);
 }
```

SETSYSLOG() *(cont.)* SETSYSLOG() *(cont.)*

When this program executes, the hsot's standard output displays

```
Before setsyslog
```

and the file *logfile* contains

```
getcube successful: cube type 1m8rxn20 allocated
After setsyslog
Hello world
```

The output of the node program

Environment

Host

Languages

C, Fortran

Description of Parameters

stdfd is an integer value that indicates where process output is to be sent:

1	redirect output to standard output of the current file server.
2	redirect output to standard error of the current file server.

Discussion

The *syslog* process is a system process used to send the output of a host process to either the standard output or standard error of the current file server of the nodes. Use **setsyslog()** to start the *syslog* process and redirect the standard output of the calling program to the standard input of the *syslog* process. Use **killsyslog()** to stop logging to the file server. For more information on using the **syslog** command, refer to the *iPSC®/2 and iPSC®/860 User's Guide.*

SETSYSLOG() *(cont.)*

SETSYSLOG() *(cont.)*

Errors

`setsyslog: There is no attached cube`

Use **cubeinfo** to find out which cubes you own.

`setsyslog: No pid defined`

Use **setpid()** to define a host process id.

`setsyslog: Active syslog exists`

syslog already in progress.

`setsyslog: Cannot start syslog process`

System will not allow *syslog* process.

See Also

killsyslog(), **syslog** command

WAITALL() WAITALL()

Wait for all the specified processes to complete.

Synopsis

waitall(*node, pid*)

Parameter Declaration

long *node, pid;*

Return Values

None

Example

The following examples illustrate uses of **waitall**().

```
waitall(10, 3);          /* Wait for process 3 on node 10 to complete. */
waitall(15, -1);         /* Wait for all processes on node 15 to complete. */
waitall(-1, -1);         /* Wait for all processes on all nodes to complete. */
```

For RX nodes, *pid* must always be `0`.

Environment

Host, Node

Languages

C, Fortran

WAITALL() *(cont.)*

WAITALL() *(cont.)*

Description of Parameters

node is an integer value that identifies a particular node where a process resides or `-1` can be used to indicate all nodes. Nodes within a partition are numbered consecutively from 0. A host id is invalid.

pid for CX nodes, *pid* is an integer value that specifies the process id (or `-1` to specify all processes). Valid *pids* can be any integer value, but negative numbers other than `-1` are reserved for system programs. The *pid* is assigned when you load a process. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on the RX node). It is not an error to wait for nonexistent processes; **waitall()** simply returns immediately.

Discussion

Use **waitall()** to block further execution of the process until the selected process(es) complete on the selected node(s). Unlike **waitone()**, it will wait for all processes to complete if `-1` is specified.

The procedure does not return until all specified processes on all specified nodes are completed. The completion codes are discarded.

Errors

```
waitall: Invalid node.
```

See Also

waitone(), **load()**, **killcube()**, **killproc()**

WAITONE() WAITONE()

Wait for a specified process to complete. If all processes or nodes are specified, **waitone()** returns as soon as the first process completes.

Synopsis

waitone(*node, pid, cnode, cpid, ccode*)

Parameter Declarations

long *node, pid, *cnode, *cpid, *ccode;*

Return Value

None

Example

The following example uses **waitone()** to supply information to **printf()** when the first node process completes.

```
long cnode, cpid, ccode;
    .
    .
waitone(-1, -1, &cnode, &cpid, &ccode);
printf("Process %d on node %d done with error code %d\n",
cpid, cnode, ccode);
    .
    .
```

Environment

Host, Node

Languages

C, Fortran

WAITONE() *(cont.)*

WAITONE() *(cont.)*

Description of Parameters

node is an integer that is either the number of a node where a process resides or is `-1` to indicate all nodes. A host id is invalid.

pid for CX nodes, *pid* is an integer value that specifies the process id. If `-1` is used to select all processes, **waitone()** returns as soon as the first process completes. An error results and the program terminates if a *pid* is specified that does not correspond to an existing process. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on the RX node). It is not an error to wait for nonexistent processes; **waitall()** simply returns immediately.

cnode, cpid, ccode

When a process with a matching *node* and *pid* completes, the *node, pid,* and completion code of the process are returned in *cnode*, *cpid*, and *ccode*, respectively.

The information returned in *ccode* is equivalent to the status information returned by the UNIX **wait()** call. If the process terminated with a call to **exit()**, *ccode* will contain the least-significant eight bits of the exit code (the argument to **exit()** in the upper eight bits and zero in the lower eight bits). If the process falls off the end of main, the exit code is undefined. If the process is killed or terminated due to an exception, *ccode* will contain the signal number indicating the cause of termination.

On the 386, exceptions map to signal numbers in the following way:

SIGNAL #	EXCEPTION #
12 SIGSYS	NA Invalid system call
09 SIGKILL	NA Process killed
08 SIGFPE	0 Integer divide by zero
05 SIGTRAP	1 Single-step interrupt
09 SIGKILL	2 Non-maskable interrupt
05 SIGTRAP	3 Breakpoint interrupt

WAITONE() *(cont.)*

WAITONE() *(cont.)*

SIGNAL #	EXCEPTION #
11 SIGSEGV	4 Integer overflow exception
11 SIGSEGV	5 Array bounds exception
04 SIGILL	6 Invalid opcode
08 SIGFPE	7 Numeric processor not available
08 SIGFPE	9 Numeric processor overrun
08 SIGFPE	11 Segment not present fault
11 SIGSEV	12 Stack overflow
11 SIGSEV	13 General protection fault
11 SIGSEV	14 Memory fault
08 SIGFPE	NA Vector processor error
08 SIGFPE	16 Floating point exception

Discussion

Use **waitone**() to wait for a process to complete on any of the nodes and provide useful information on how the process terminated. If `-1` is used to select all processes, **waitone**() returns as soon as the first process completes. Similarly, if `-1` is used to select all nodes, **waitone**() returns as soon as the specified process on any node completes. This differs from the **waitall**() which does not return until all selected processes on selected nodes complete.

Errors

```
waitone: No pid defined
```

Use **setpid**() to define a host process id.

See Also

waitall(), **waitcube** command

iPSC®/2 and iPSC®/860 FORTRAN ROUTINES 3

INTRODUCTION

This chapter documents each of the iPSC routines in the two Fortran system interface libraries: host and node. These routines allow you to perform such tasks as accessing and releasing cubes, loading, starting, and killing processes, and passing messages between processes, from within a Fortran program.

Routines for host processes are found in the */usr/lib/libhost.a* library and routines for node processes are found in the */usr/lib/libnode.a* library.

The routines are listed alphabetically. Appendix A provides a summary of the routines with a synopsis, brief description, and whether the routine is available for host, node, or both programs.

attachcube()	**etos()**	**gland()**	**infonode()**	**msgcancel()**
cprobe()	**flick()**	**glor()**	**infopid()**	**msgdone()**
cread()	**flushmsg()**	**glxor()**	**infotype()**	**msgwait()**
crecv()	**forceflush()**	**gopf()**	**iodone()**	**myhost()**
csend()	**forflush()**	**gray()**	**iomode()**	**mynode()**
csendrecv()	**fpsetmask()**	**gsendx()**	**iowait()**	**mypid()**
ctohd()	**gcol()**	**gshigh()**	**iprobe()**	**newserver()**
ctohf()	**gcolx()**	**gslow()**	**iread()**	**nodedim()**
ctohl()	**gdhigh()**	**gsprod()**	**irecv()**	**numnodes()**
ctohs()	**gdlow()**	**gssum()**	**isend()**	**plogon()**
cubeinfo()	**gdprod()**	**gsync()**	**isendrecv()**	**plogoff()**
cwrite()	**gdsum()**	**handler()**	**iseof()**	**relcube()**
dclock()	**getcube()**	**hrecv()**	**iwrite()**	**restrictvol()**
eadd()	**giand()**	**hsend()**	**killcube()**	**setiomode()**
ecmp()	**gilow()**	**hsendrecv()**	**killproc()**	**setpid()**
ediv()	**glor()**	**htocd()**	**killsyslog()**	**setsyslog()**
emod()	**giprod()**	**htocf()**	**led()**	**stoe()**
emul()	**ginv()**	**htoci()**	**load()**	**waitall()**
eseek()	**gihigh()**	**htocs()**	**lsize()**	**waitone()**
esize()	**gisum()**	**hwclock()**	**masktrap()**	
esub()	**gixor()**	**infocount()**	**mclock()**	

The Fortran routines above are documented as follows:

- name of the routine and brief description
- synopsis
- parameter declarations
- return values where applicable
- example
- environment
- languages
- detailed description including input parameters
- error messages returned by this routine
- list of related routines

ATTACHCUBE() ATTACHCUBE()

Make a different cube the current cube.

Synopsis

SUBROUTINE **ATTACHCUBE**(*cubename*)

Parameter Declarations

CHARACTER *cubename*(*)

Return Value

None

Example

This example allocates two cubes (ALPHA and BETA) with **getcube()**, and then changes the current cube first to ALPHA and then to BETA with **attachcube()**.

```
      CHARACTER*15 CUBE1, CUBE2
           .
           .
      CUBE1 = 'ALPHA'
      CUBE2 = 'BETA'

      CALL GETCUBE(CUBE1, '32', ' ', 0)
      CALL GETCUBE(CUBE2, '8VX', 'TRAINON', 1)

C     THE CURRENT CUBE IS 'BETA'
           .
           .
      CALL ATTACHCUBE(CUBE1)

C     THE CURRENT CUBE IS NOW 'ALPHA'
           .
           .
      CALL ATTACHCUBE (CUBE2)

C     THE CURRENT CUBE IS NOW 'BETA' AGAIN
```

ATTACHCUBE() *(cont.)*

ATTACHCUBE() *(cont.)*

Environment

Host

Languages

C, Fortran

Description of Parameters

cubename refers to a character string that specifies the name of the cube that you want to attach.

Discussion

Use **attachcube()** to change a process's current cube to another cube you previously allocated. When you call **attachcube()**, you are making the specified *cubename* your current cube and all subsequent message passing and loader calls made by the process that called **attachcube()** will apply to *cubename*.

Use **attachcube()** to run applications in several cube partitions. You can only be attached to one cube at a time. Use **getcube()** to allocate a cube and **attachcube()** to change cubes within your program.

Names of cubes belonging to a single user must be unique. Names are assigned with **getcube()**. If the parameter is null or refers to a null string, the default *cubename* becomes "defaultname." If you specify a cube that has not been allocated, an error message results and the process will be terminated. Use **cubeinfo()** to find out the names of allocated cubes.

ATTACHCUBE() *(cont.)*

ATTACHCUBE() *(cont.)*

Errors

`attachcube: Cubename does not exist`

Use **cubeinfo()** to determine cube names.

`attachcube: Commser not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with the **bootcube** or **rebootcube** command.

`attachcube: Lifeline not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with the **bootcube** or **rebootcube** command.

See Also

cubeinfo(), getcube(), relcube()

CPROBE() CPROBE()

Wait for a message to arrive. Blocks the calling process until the message is available for receipt.

Synopsis

SUBROUTINE **CPROBE**(*typesel*)

Parameter Declarations

INTEGER *typesel*

Return Value

None

Example

This example uses **cprobe()** to block any further action until a message of type 10 is available to be received. After using **infocount()** to ensure that the message is of the proper length, **crecv()** is used to receive the message.

```
PARAMETER(ITYPE=10)

CHARACTER MSGBUF*80
      .
      .
CALL CPROBE(ITYPE)
IF (INFOCOUNT().LE.LEN(MSGBUF)) THEN
  CALL CRECV (ITYPE, MSGBUF, LEN(MSGBUF))
ENDIF
```

CPROBE() *(cont.)*

CPROBE() *(cont.)*

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

typesel is an integer value that specifies the type(s) of message(s) you are waiting for. 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 the probe operation will be recognized. After the first message has been received, you can use `-1` again to probe for the next message.
- If *typesel* is any negative number other than `-1`, a set of message types will be recognized. Refer to Appendix B for information on building a *typesel* mask.

Discussion

Use **cprobe()** as a synchronous call that causes a process to wait until a message of the selected type (or types) is available to be received. The type must match the type specified in the **csend()** or **isend()** sending the message. When **cprobe()** returns, you can use **crecv()** or **irecv()** to initiate the receipt of the desired message. Use the **info...()** calls to get more information about the message.

Use **iprobe()** rather than **cprobe()** when you want to determine whether a message of a selected type is pending but do not want the calling process to be blocked if the message is not available.

CPROBE() *(cont.)*

CPROBE() *(cont.)*

Errors

`cprobe: No pid defined`

Use **setpid()** to define a host process id.

`cprobe: Too many requests`

Use **msgwait()** for outstanding **isend()** or **irecv()** requests.

`cprobe: Invalid type`

Cannot probe for force-type messages.

See Also

infocount(), infonode(), infopid(), infotype(), iprobe()

CREAD() CREAD()

High speed synchronous read from a CFS file. Blocks the calling process until the read is complete.

Synopsis

SUBROUTINE **CREAD**(*unit, buf, len*)

Parameter Declarations

INTEGER *unit*
INTEGER *buf(*)*
INTEGER *len*

Return Value

None

Example

The following example uses **cread**() to read the contents of a file after opening it with **open**():

```
      DIMENSION BUFFER(1000)
C     OPEN A FILE ON CFS NAMED MYDATA
      OPEN(UNIT=3, FILE='/cfs/mydata', FORM = 'UNFORMATTED')
C     READ ONE BUFFER'S WORTH OF DATA FROM THE FILE
      CALL CREAD(3, BUFFER, BUFLEN)
```

Environment

Node

Languages

C, Fortran

CREAD() *(cont.)*

CREAD() *(cont.)*

Description of Parameters

unit the file descriptor.

buf refers to the buffer where the read data will be stored. The buffer can be any valid data type.

len an integer value that specifies the size of the buffer in bytes. Transfer sizes are limited only by the available memory for the buffer.

Discussion

Use **cread()** to perform high-speed synchronous data reads from the file identified with the *unit* parameter which you assigned to the file with the unit parameter of **open()**. To use **cread()**, you must use **open()** and set the FORM parameter to `'UNFORMATTED'`.

For a given file, you can use either the iPSC system reads and writes (**cread()**, **iread()**, **cwrite()**, **iwrite()**), or the standard Fortran **read()** and **write()**. An attempt to access a file with a mixture of the two kinds causes an error. Files automatically created with **read()** or **write()** without using **open()** will not have the correct format for the high-speed iPSC system reads and writes, but can be accessed with **read()** and **write()**.

With **cread()**, the process is blocked from executing further instructions until the read is complete and the read data is in the buffer you designate. If you do not want the calling process blocked during the read, use **iread()**.

Use **iseof()** to determine whether the read operation moved the file pointer to the end-of-file.

CREAD() *(cont.)*

CREAD() *(cont.)*

Errors

`cread: Invalid argument`

Check arguments.

`cread: Mixed file operations`

In I/O mode 2 or 3, nodes are attempting different operations (reads and writes) to a shared file. Ensure that nodes are performing the same operation.

`cread: I/O error`

Make sure file is open and of the proper format.

See Also

cwrite(), iread(), iseof(), iwrite()
UNIX System V/386 Programmer's Reference: **open(), creat(), dup(), read()**

CRECV() CRECV()

Receive a message and wait for the receive to complete before proceeding.

Synopsis

SUBROUTINE **CRECV**(*typesel, buf, len*)

Parameter Declarations

INTEGER *typesel*
INTEGER *buf(*)*
INTEGER *len*

Return Value

None

Example

This example contains a **crecv()** statement that receives the first message to arrive and stores it in buffer *BUF1*. The buffer can be of any type; in this case it is declared to be of type **CHARACTER**. The buffer length is declared as 256K bytes.

```
      INTEGER BUFLEN1
      PARAMETER(BUFLEN1=256)
      CHARACTER BUF1*BUFLEN1

C     RECEIVE THE FIRST MESSAGE TO ARRIVE AND
C     STORE IT IN 'BUF1'.

      CALL CRECV(-1, BUF1, BUFLEN1)
```

CRECV() *(cont.)*

CRECV() *(cont.)*

Environment

Node, Host

Languages

C, Fortran

Description of Parameters

typesel is an integer value that specifies the type(s) of message you are waiting for. 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 non-force-type to arrive for the process that initiated the receive operation will be recognized. After the first message has been received, you can use `-1` again to receive the next message.
- If *typesel* is any negative number other than `-1`, a set of message types will be recognized. Refer to Appendix B for information on building a *typesel* mask.

buf refers to the buffer where the received message will be stored. The buffer can be any valid data type. It is recommended that data types match in send and receive operations.

len is an integer value that specifies the size of the message buffer in bytes. The maximum number of bytes that you can send in a single message to or from a host process is 256K bytes. There is no limit on message length between nodes.

CRECV() *(cont.)*

CRECV() *(cont.)*

Discussion

Use **crecv()** to initiate the receipt of a message, and then cause the calling process to wait for a message whose type matches the type(s) specified in the *typesel* parameter. When the message is received, it is stored in *buf* and the calling process resumes execution. You can use the **info...()** calls to get more information about the message after it is received.

Use **irecv()** when you do not want the calling process to be blocked.

Errors

`crecv: No pid defined`

Use **setpid()** to define a host process id.

`crecv: Invalid length`

Use a non-negative number or a length less than or equal to maximum message length.

`crecv: Too many requests`

Use **msgwait()** for **isend()** or **irecv()** requests that are outstanding.

`crecv: Buffer length exceeds allocation`

Make sure parameter length does not exceed buffer size.

`crecv: Invalid buffer pointer`

Specify a pointer which contains the address of a valid data buffer (C only).

`crecv: Received message too long for buffer`

Make sure the buffer is large enough to hold the message.

`crecv: Invalid node`

Use **numnodes()** to determine cube size and **myhost()** to determine host number.

CRECV() *(cont.)*

CRECV() *(cont.)*

See Also

csend(), csendrecv(), infocount(), infonode(), infopid(), infotype(), irecv(), isend()

CSEND() CSEND()

Send a message. Blocks until execution is complete.

Synopsis

SUBROUTINE **CSEND**(*type, buf, len, node, pid*)

Parameter Declarations

INTEGER *type*
INTEGER *buf(*)*
INTEGER *len, node , pid*

Return Value

None

Example

This example definesThe messages "HELLO NODE 0" and "HELLO NODE 1". Node 0 sends a message to node 1 and receives a reply. Node 1 does likewise.

```
      INCLUDE 'fcube.h'
      CHARACTER SMSG*100
      CHARACTER RMSG*100
      INTEGER IAM,MSGLEN
      IAM = MYNODE()
      IF(IAM .EQ. 0) THEN
         SMSG = 'HELLO NODE 1'
         MSGLEN = 12
         CALL CSEND(100, SMSG, MSGLEN,1,0)
         CALL CRECV(200, RMSG, MSGLEN)
      ELSE
         SMSG = 'HELLO NODE 0'
         MSGLEN = 12
         CALL CSEND(200, SMSG, MSGLEN,0,0)
         CALL CRECV(100, RMSG, MSGLEN)
      ENDIF
      WRITE(*,1000) RMSG
1000  FORMAT(1X,A12)
      END
```

CSEND() *(cont.)*

CSEND() *(cont.)*

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

type	an integer that identifies the type of the message that you are sending.
buf	refers to the buffer that contains the message to be sent. The buffer may be any legal data type. It is recommended that data types match in send and receive operations.
len	a positive integer that specifies the size (in bytes) of the message that you wish to send. Messages to or from a host process are limited to a maximum of 256K bytes. There is no limit on message length between nodes.
node	an integer that defines where the message is to be sent.
pid	an integer that specifies the id of the process that is to receive the message. For the RX node (which supports the single process model), *pid* must be set to `0`. Values other than `0` return an error. The CX node supports the multiple process model and can be any integer.

CSEND() *(cont.)*

CSEND() *(cont.)*

Discussion

Use **csend()** to send a message to a node or host process and cause the calling process to wait until the message is sent. Completion of **csend()** does not imply that the message was received by the destination process, only that the message was sent and that the buffer (*buf*) is available for reuse.

It is recommended that you use a type with a value from `0` to `999,999,999`. Types `1,073,741,824` to `1,999,999,999` are special force types that bypass the normal flow control mechanisms, do not match the `-1` wildcard, and are discarded if no receive is posted. Force types are intended to be used with **csendrecv()**, **isendrecv()**, and **hsendrecv()**, but can be used by other message passing calls.

NOTE

It is not recommended that you use types in other ranges because they are used by the system and could produce unpredictable results.

If *node* is a positive integer, the message is sent to the node with that number. Nodes within a partition are numbered consecutively from 0, and a node number higher than the highest node in the cube causes an error to be returned. To send a message to the host, use the host's id, which is returned by **myhost()**.

If *node* is a negative number up to a certain value, it causes a "global" send. If *node* is `-1`, the message is sent to all nodes. Other negative numbers cause the message to be sent to a subcube composed of a set of the nodes surrounding the sender. The dimension of this cube is calculated as (*d* - **nodedim()** `-1`) , where *d* is the dimension of the subcube, and is less than or equal to **nodedim()**.

For RX nodes, if a global send specifies its own process (**mypid()**) it does not receive its own message. For CX nodes, however, if a process other than its own is specified, the sending node also receives the message.

For the CX node, Valid *pids* include any integer value. Negative numbers are reserved for system programs. A node *pid* is assigned when you load a process and a host *pid* is assigned with **setpid()**. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on the RX node).

To initiate a send request when you want the calling process to continue during the send operation, use **isend()**. To define a *pid*, use the **setpid()** call.

CSEND() *(cont.)*

CSEND() *(cont.)*

Errors

`csend: No pid defined`

Use **setpid()** to define a host process id.

`csend: Invalid type`

Use a non-negative number.

`csend: Invalid length`

Use a non-negative number or a length less than or equal to maximum message length.

`csend: Invalid node`

Use **numnodes()** to determine cube size or **myhost()** to determine host number.

`csend: Invalid buffer pointer`

Specify a pointer which contains the address of a valid data buffer.

`csend: Not enough memory`

Make sure message buffer starts on a boundary which is a multiple of four or make more memory available to this process.

See Also

cprobe(), crecv(), csendrecv(), iprobe(), irecv(), isend()

CSENDRECV() CSENDRECV()

Send a message and receive a reply. The calling process is blocked until the reply is received.

Synopsis

INTEGER FUNCTION **CSENDRECV**(*type, sbuf, slen, tonode, topid, typesel, rbuf, rlen*)

Parameter Declarations

INTEGER *type, typesel*
INTEGER *sbuf(*), rbuf(*)*
INTEGER *slen, tonode , topid, rlen*

Return Value

Length of the received message.

Example

The following example illustrates how to use **csendrecv**(). Part 1 runs on node 0 and uses **csendrecv**() to make a remote procedure call and post a receive for the message containing the result. Part 2 runs on node 1 and receives the message; it performs the receive operation, and returns the result to the first procedure.

Part 1: (runs on node 0)

```
      INCLUDE 'fcube.h'
      DOUBLE PRECISION RSIN, PI, X , ANGLE
      INTEGER IAM

      IAM = MYNODE()
      PI  = ACOS(-1.0D00)
      ANGLE = PI/2.0D00
      X  = RSIN(ANGLE)
      WRITE(*,1000) CHAR(10),IAM,X,ANGLE
1000  FORMAT(1X,A1,'NODE = ', I2':  X =', D15.5,
     X       ' FOR ANGLE ',D15.5)
      END
```

CSENDRECV() *(cont.)*

CSENDRECV() *(cont.)*

```
      DOUBLE PRECISION FUNCTION RSIN(X)
            DOUBLE PRECISION X, RESULT
            INTEGER LEN

            LEN = CSENDRECV(100, X,8,1,0,200,RESULT,8)
            RSIN = RESULT
            END
```

Part 2: (runs on node 1)

```
            INCLUDE 'FCUBE.H'

            DOUBLE PRECISION X, RESULT

      10    CONTINUE
            CALL CRECV(100,X,8)
            RESULT = SIN(X)
            CALL CSEND(200, RESULT, 8,0,0)
            GOTO 10
            END
```

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

type	a non-negative integer that identifies the type of the message that you are sending.
sbuf	refers to the buffer that contains the message to be sent.
slen	a positive integer that specifies the size (in bytes) of the message that you wish to send. Messages to or from a host process are limited to a maximum of 256K bytes. There is no limit on message length between nodes.
tonode	an integer that determines the node(s) to which the message is to be sent.

CSENDRECV() *(cont.)*

CSENDRECV() *(cont.)*

topid an integer that specifies the id of the process that is to receive the message. Valid *pids* for RX nodes include any integer value. Negative numbers are reserved for system programs. A node *pid* is assigned when you load a process and a host *pid* is assigned with **setpid()**. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on the RX node).

typesel is an integer value that specifies the type(s) of the reply message.

rbuf refers to the buffer where the reply message will be stored. The buffer can be any valid data type. It is recommended that data types match in send and receive operations.

rlen an integer value that specifies the size of the reply message buffer in bytes.

Discussion

Use **csendrecv()** as a synchronous call that simultaneously sends a message to a node or host process and posts a receive for the reply. The process is blocked from continuing until the return message is received. When a message whose type matches the type(s) specified in the *typesel* parameter arrives, the calling process receives the message, stores it in *rbuf* and the calling process resumes execution. You cannot use the **info...()** calls with this call; it does not affect the information.

This call is intended to be used for remote procedure calls. If you do not want to block the procedure while the send-receive operation is occurring, use **isendrecv()**.

The *typesel* parameter is a user-defined variable that identifies the kind of information contained in the message being sent. Types `0` to `999,999,999` are normal types that can be used by any send or receive call, including this one. Types `1,073,741,824` to `1,999,999,999` are special force types available to the user intended specifically for the "sendrecv" calls. Force types have special properties:

- A message with a force type bypasses the normal flow control mechanisms and is not delayed by clogged message buffers on the node.
- 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.

CSENDRECV() *(cont.)*

CSENDRECV() *(cont.)*

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.

The buffer for the send and receive may be any legal data type. It is recommended that data types match in send and receive operations.

If *tonode* is a positive integer, the message is sent to the node with that number. Nodes within a partition are numbered consecutively from 0, and a node number higher than the highest node in the cube causes an error to be returned. To send a message to the host, use the host's id, which is returned by **myhost()**.

If *tonode* is a negative number up to a certain value, it causes a "global" send. If node is `-1`, the message is sent to all nodes. Other negative numbers cause the message to be sent to a subcube composed of a set of the nodes surrounding the sender. The dimension of this cube is calculated as (*d* - **nodedim()** `-1`) , where *d* is the dimension of the subcube, and is less than or equal to **nodedim()**.

The *typesel* parameter describes the type(s) of the message(s) expected to be received. 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. You can use a force type, as described above.
- If *typesel* is `-1`, the first message to arrive for the process that initiated the receive operation will be recognized. After the first message has been received, you can use `-1` again to receive the next message.
- If *typesel* is any negative number other than `-1`, a set of message types will be recognized. Refer to Appendix B for information on building a *typesel* mask.

CSENDRECV() *(cont.)*

CSENDRECV() *(cont.)*

Errors

`csendrecv: No pid defined`

Use **setpid()** to define a host process id.

`csendrecv: Invalid type`

Use a non-negative number.

`csendrecv: Invalid length`

Use a non-negative number or a length less than or equal to maximum message length.

`csendrecv: Invalid node`

Use **numnodes()** to determine cube size or **myhost()** to determine host number.

`csendrecv: Buffer length exceeds allocation`

Make sure parameter length does not exceed buffer size.

`csendrecv: Invalid buffer pointer`

Specify a pointer which contains the address of a valid data buffer.

`csendrecv: Not enough memory`

Make sure message buffer starts on a boundary that is a multiple of four or make more memory available to this process.

`csendrecv: Received message too long for buffer`

Make sure buffer is large enough to hold the message.

See Also

cprobe(), crecv(), hsendrecv(), iprobe(), irecv(), isend(), isendrecv()

CTOH*x*(), HTOC*x*() CTOH*x*(),HTOC*x*()

Convert node byte order to remote workstation byte order, or remote workstation byte order to node byte order.

Synopsis

CTOHD(*sv*,*n*) byte swap cube to host **double precision**
HTOCD(*sv*,*n*) byte swap host to cube **double precision**

CTOHF(*sv*,*n*) byte swap cube to host **real**
HTOCF(*sv*,*n*) byte swap host to cube **real**

CTOHL(*sv*,*n*) byte swap cube to host **integer*4**
HTOCL(*sv*,*n*) byte swap host to cube **integer*4**

CTOHS(*sv*,*n*) byte swap cube to host **integer*2**
HTOCS(*sv*,*n*) byte swap host to cube **integer*2**

Parameter Declarations

CTOHD() and **HTOCD()**:	**DOUBLE PRECISION** *sv(*)* **INTEGER** *n*
CTOHF() and **HTOCF()**:	**REAL** *sv(*)* **INTEGER** *n*
CTOHL() and **HTOCL()**:	**INTEGER*4** *sv(*)* **INTEGER** *n*
CTOHS() and **HTOCS()**:	**INTEGER*2** *sv(*)* **INTEGER** *n*

Return Value

None

CTOH*x*(), HTOC*x*() *(cont.)*

CTOH*x*(), HTOC*x*() *(cont.)*

Example

The following example is a fragment of a host program that sends a message (of type *NODETYPE* and consisting of the array *DATAIN*) from the host to all nodes, and then receives a message back (the same type, but consisting of the array *DATAOUT*). Sometimes the byte order in which data is stored on the remote host is different than the byte order of data on the nodes. In these cases, **HTOCL()** changes the byte order of *DATAIN* before it goes to the nodes, and **CTOHL()** changes the byte order back to the remote host's byte order after the message from the nodes is received:

```
      INTEGER DATAIN(100)
      INTEGER DATAOUT(100)
      INTEGER NODETYPE, NODEPID
        .
        .
C  SWAP BYTES TO NODE ORDER BEFORE SENDING MESSAGE TO NODE 0
      CALL HTOCL(DATAIN,100)
      CALL CSEND(NODETYPE, DATAIN, LEN(DATAIN), 0, NODEPID)
        .
        .
      CALL CRECV(NODETYPE, DATAOUT, LEN(DATAOUT))
C  SWAP BYTES BACK TO HOST ORDER BEFORE USING THE CONTENTS
C    OF THE MESSAGE BUFFER

      CALL CTOHL(DATAOUT,100)
        .
        .
```

Environment

Host

Languages

C, Fortran

CTOH*x*(), HTOC*x*() *(cont.)*

CTOH*x*(), HTOC*x*() *(cont.)*

Description of Parameters

sv starting address of variable(s) to swap . Individual variables of any type, elements of structures, or arrays can be used as long as the address is specified. When you want to swap an entire array, use *n* for the number of elements and *sv* as the array starting address.

n number of *sv*'s to swap. If you want to swap multiple elements that are the same type and are also contiguous in memory, use *n* to specify the number of elements to swap. Otherwise, specify `1`.

Discussion

Remote development machines that do not use a 286™ or 386™ microprocessor as its main processor may have a different internal byte order representation and byte alignment of integer and floating point values than the nodes. If so, you need to use these utilities in host programs whenever you send or receive data between node and host programs. These utilities are provided to convert byte order between the node and the host. These calls must be used in the host program, because the node expects to receive that data in the proper byte order.

Therefore, before sending data from remote host to node with one of the send operations (**csend**() or **isend**()), use one of the **HTOC***x* calls depending on the data type, and after receiving data from the node program via **crecv**() or **irecv**(), change the byte order back to the remote host byte order with one of the **CTOH***x* calls.

The byte-swapping utilities are provided for source code compatibility, and the utilities have no effect on programs running on the SRM and remote development machines that do not require byte-order change.

Errors

None

See Also

csend(), **crecv**(), **irecv**(), **isend**()
For C programs only, **createstruc**()

CUBEINFO() CUBEINFO()

Obtain information about allocated cubes. This call is for use by host programs; it is not available for node programs.

Synopsis

INTEGER FUNCTION **CUBEINFO**(*ct, numslots, global*)

Parameter Declarations

INTEGER *numslots, global*
CHARACTER*16 *ct (slotsize, numslots)*

Return Value

An integer containing the number of cubes for which information is available. If the number of cubes allocated was greater than the value you specify for *numslots*, this value will be larger than *numslots*. However, the number of cubes for which information is returned to the buffer table never exceeds *numslots*.

Example

The following fragment is an example of how you might use the **cubeinfo()** call in a program. In this example, the number of slots is the maximum number of cubes allowed in a system (10). The **cubeinfo()** call is called with the *global* parameter set to 2, to get information about all allocated cubes on the current System Resource Manager (SRM). This returns the number of cubes found (storing it in the variable *numcubes*) and also fills in the table *ctbl* with the cube information for each of the cubes. It then does a string compare to make sure that the cubename of one of allocated cubes is the one that you want to use in the subsequent processing steps (in this case, the cube's name is "alpha").

```
      PROGRAM MAIN
      INCLUDE 'fcube.h'
      CHARACTER *16 CTBL(SLOTSIZE,10)
      DATA NUMSLOTS/10/
      NUMCUBES = CUBEINFO(CTBL, NUMSLOTS, 2)
      DO 10 I=1,NUMCUBES
        IF (CTBL(CUBENAME,I) .EQ. 'ALPHA'
            .
            .
10    CONTINUE
```

CUBEINFO() *(cont.)*

CUBEINFO() *(cont.)*

Notes

The *ct* parameter refers to a buffer table that will hold the returned cube information values. It is declared as a two-dimensional array of character strings. *SLOTSIZE* is a fixed value that is defined in */usr/include/fcube.h*. *NUMSLOTS* is a user-defined parameter.

A number of parameters are defined in the *fcube.h* INCLUDE file to help you define and access the buffer table. These parameters are:

```
INTEGER CUBENAME, USERNAME, SRMNAME, HOSTNAME, TTY, CUBETYPE
INTEGER CINDEX, TTY1, TTY2, TTY3, TTY4, SLOTSIZE

PARAMETER (CUBENAME=1)
PARAMETER (USERNAME= 2)
PARAMETER (SRMNAME= 3)
PARAMETER (HOSTNAME= 4)
PARAMETER (TTY= 5)
PARAMETER (CUBETYPE= 6)
PARAMETER (CINDEX= 7)
PARAMETER (TTY1= 8)
PARAMETER (TTY2= 9)
PARAMETER (TTY3= 10)
PARAMETER (TTY4= 11)
PARAMETER (SLOTSIZE= 11)
```

The fields of the buffer table contain the following information:

CUBENAME Name of the cube.

USERNAME Name of user.

SRMNAME Name of the SRM to which the cube is attached.

HOSTNAME Name of the host where **getcube()** was invoked. Name is either the remote system name or *SRMNAME*.

TTY TTY where **getcube()** was invoked.

CUBETYPE Type of this cube.

CINDEX System table index (a unique identifier for the cube, not normally used in applications programs).

TTY1-TTY4 Names of terminals attached to the cube. Unused entries are set to all blanks.

CUBEINFO() *(cont.)* CUBEINFO() *(cont.)*

A user logged into the same host on different *ttys* may be attached to the same cube from each *tty*. Up to four *ttys* may be attached to the same cube. Strings for attached *ttys* are of format `'0'` for no *tty* and a string containing the full pathname on the system for valid *ttys*.

Environment

Host

Languages

C, Fortran

Description of Parameters

ct specifies a buffer table where cube information is to be returned.

numslots is the number of elements in the table to which cube information is returned. Each element contains the cube information for a single cube.

global is an integer value from `0` through `4` that determines the cubes for which information is returned. The possible settings are:

`0`	returns information about the current attached cube.
`1`	returns information about all cubes that you own that were allocated from the current host
`2`	returns information about all the cubes on the system from which the command was executed. If executed on an SRM, information is returned for all cubes allocated from that SRM; if executed on a remote development machine, supplies information on all cubes that have been allocated from that workstation is returned.
`3`	returns information about how cubes are allocated on all SRMs. This is equivalent to setting *global* to `2` on each SRM on the network. This value is not allowed on the SRM.

CUBEINFO() *(cont.)*

CUBEINFO() *(cont.)*

4 requires that the call have the additional parameter *srmname*. This call should only be used in a program to be run on a remote host, and allows you to specify the name of a particular SRM in a network. The call becomes:

cubeinfo(*ct, numslots, global, srmname*)

where the parameter *srmname* in C is a pointer to a character string, and in Fortran is a character string, naming the SRM of interest.

Discussion

Use **cubeinfo()** to obtain information about allocated cubes. The return value is the number of cubes matching the *global* parameter. In addition, **cubeinfo()** fills a slot in a buffer with an array of information about each of these cubes. The parameter *numslots* limits the number of cubes for which information will be placed in a slot. If there are fewer selected cubes than *numslots*, the remaining slots will be cleared to contain all zeros. If there are no cubes found, **cubeinfo()** returns a `0` and the buffer is cleared.

Errors

`cubeinfo: Commser not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with the **bootcube** or **rebootcube** command.

`cubeinfo: Lifeline not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with the **bootcube** or **rebootcube** command.

`cubeinfo: Global value invalid`

Use a value of `0, 1, 2, 3,` or `4` for global.

See Also

attachcube(), getcube(), relcube()

CWRITE() CWRITE()

High speed synchronous write to a CFS file. Blocks the calling process until the write is complete.

Synopsis

SUBROUTINE **CWRITE**(*unit, buf, len*)

Parameter Declarations

INTEGER *unit*
INTEGER *buf(*)*
INTEGER *len*

Return Value

None

Example

The following example uses **cwrite()** to write the contents of the designated buffer to a file. First, the file is opened with **open()**, setting the format to `'UNFORMATTED'` (required for **cwrite()**). After setting up the buffer, **cwrite()** is used to write the data in the buffer into the file.

```
          .
          .
      INTEGER BUFFER(BUFLEN)
C     CREATE A FILE ON CFS NAMED MYDATA
      OPEN(UNIT=3, FILE='/cfs/mydata', FORM = 'UNFORMATTED')
C     FILL A BUFFER NAMED "BUFFER" WITH DATA
          .
          .
C     WRITE CONTENTS OF BUFFER TO FILE
      CALL CWRITE(3, BUFFER, BUFLEN)
          .
          .
```

CWRITE() *(cont.)*

CWRITE() *(cont.)*

Environment

Node

Languages

C, Fortran

Description of Parameters

unit	the unit number assigned when the file was opened.
buf	refers to the buffer where the data to be written is stored. The buffer can be any valid data type.
len	an integer value that specifies the buffer size in bytes. Transfer size is limited only by memory available for the buffer.

Discussion

Use **cwrite()** to perform high-speed synchronous data writes to the file identified with the *unit* parameter which you assigned to the file with the *unit* parameter of **open()**. To use **cwrite()**, you must use **open()** and set the FORM parameter to `'UNFORMATTED'`.

For a given file, you can use either the iPSC system reads and writes (**cread()**, **iread()**, **cwrite()**, **iwrite()**), or the standard Fortran **read()** and **write()**. An attempt to access a file with a mixture of the two kinds causes an error. Files automatically created with **read()** or **write()** without using **open()** will not have the correct format for the high-speed iPSC system reads and writes, but can be accessed with **read()** and **write()**.

With **cwrite()**, the process is blocked from executing further instructions until the write is complete. If you do not want the calling process blocked during the read, use **iwrite()**.

CWRITE() *(cont.)*

CWRITE() *(cont.)*

Errors

`cwrite: Invalid argument`

Check arguments.

`cwrite: Mixed file operations`

In I/O mode 2 or 3, nodes are attempting different operations (reads and writes) to a shared file. Ensure that nodes are performing the same operation.

`cwrite: I/O error`

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

`cwrite: No such unit`

Unit must be an integer from `1` through `100`.

`cwrite: Too may open files`

Only 100 files can be open at one time for any process.

`cwrite: Not enough core`

Not enough memory on node for operation.

`cwrite: Unformatted I/O to FORMATTED file`

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

`cwrite: No space left on device`

Not enough space on device to which you are writing. Create more space in file system, or if you have restricted the disks to which you are writing with **restrictvol()**, re-execute **restrictvol()** to include more drives.

CWRITE() *(cont.)*

CWRITE() *(cont.)*

`cwrite: Attempt to mix standard and CFS I/O calls`

Fortran **read()** and **write()** cannot be used on the same file as CFS I/O calls.

`cwrite: Attempt to write to READONLY file`

Check file attributes.

See Also

cread(), iread(), iwrite()
iPSC®/860 Fortran Language Reference Manual: **OPEN(), WRITE()**

DCLOCK() DCLOCK()

Returns double precision time in seconds since booting the cube.

Synopsis

DOUBLE PRECISION **DCLOCK**( )

Parameter Declarations

None

Return Value

Double precision time (in seconds)

Example

The following example illustrates the proper use of **dclock**().

```
      DOUBLE PRECISION STIME, ETIME, TTIME
          .
          .
C FIND STARTING TIME
      STIME = DCLOCK()
C EXECUTE PROCESS
          .
          .
C FIND ENDING TIME
      ETIME = DCLOCK()
C FIND ELAPSED TIME
      TTIME = ETIME - STIME
```

DCLOCK() *(cont.)*

DCLOCK() *(cont.)*

Environment

Node

Languages

C, Fortran

Discussion

The **dclock()** subroutine returns double precision time in seconds since the cube was booted. In this form, there is enough precision to maintain 100ns accuracy (which is the count rate for the RX node board counter) for a period of approximately 14 years. On the CX node, the maximum precision will be 1ms.

In Fortran, the intrinsic called *SECOND* returns the time in seconds.

Errors

None

See Also

hwclock(), mclock()

EADD(), ECMP(), EDIV(), EMOD(), EMUL(), ESUB()

Performs mathematical operations on 64-bit integers.

Synopsis

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*

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)*

EADD(), ECMP(), EDIV(), EMOD(), EMUL(), ESUB() *(cont.)*

Return Values

ecmp() returns:

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

All others have no return values.

Example

The following example shows a few typical mathematical operations:

```
      INTEGER E2(2)
      INTEGER E1(2), N, Q, R
C  E1 IS 8G BYTES
      E1(1) = 0
      E1(2) = 2
      E2(1) = 1
      E2(2) = 0

C  INCREMENT E1 by 1
      CALL EADD(E1, E2, E1)
C  DECREMENT E1 BY 1
      CALL ESUB(E1, E2, E1)

C  MULTIPLY E1 by 2
      N = 2
      CALL EMUL(E1, N, E1)

C  DIVIDE E1 by 100
      N = 100
      CALL EDIV(E1, N, Q)

C  REMAINDER
      CALL EMOD(E1, N, R)
```

EADD(), ECMP(), EDIV(), EMOD(), EMUL(), ESUB() *(cont.)*

This example compares two extended numbers *e1* and *e2*.

```
INTEGER E1(2), E2(2)
E1(1) = 3
E1(2) = 4
E2(1) = 0
E2(2) = 2
IF (ECMP(E1, E2) .EQ. 1) THEN
  PRINT *, 'E1 GREATER THAN E2'
          .
          .
```

Environment

Node

Languages

C, Fortran

Description of Parameters

e1, e2	extended number values, implemented as an array of 2 integers
n	integer value
eresult	resulting extended number
result	resulting integer number

EADD(), ECMP(), EDIV(), EMOD(), EMUL(), ESUB() *(cont.)*

Discussion

The following routines perform mathematical functions on 64-bit integers. These routines support the extended file sizes in CFS.

eadd()	Adds two extended numbers
ecmp()	Compares two extended numbers
ediv()	Divides extended number by integer
emod()	Gets remainder of **ediv()**
emul()	Multiplies extended number with integer
esub()	Subtracts two extended numbers

Errors

`ediv: Invalid size`

Quotient should fit in an integer.

`emod: Invalid size`

Quotient should fit in an integer.

See Also

etos(), stoe()

ESEEK() ESEEK()

Move file pointer in extended files.

Synopsis

SUBROUTINE **ESEEK**(*unit, offset, whence, newpos*)

Parameter Declarations

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

Return Values

None

Example

The following example uses **eseek()** to move the file pointer to the beginning of a file.

```
INTEGER OFFSET(2), NEWPOS(2)
          .
          .
OFFSET(1)=0
OFFSET(2)=0
CALL ESEEK(3,OFFSET,0,NEWPOS)
```

Environment

Node

Languages

C, Fortran

ESEEK() *(cont.)*

ESEEK() *(cont.)*

Description of Parameters

unit the unit you assign with the **open**() call.

offset an integer vector of length two that affects the current size of the file depending upon the value of *whence*.

whence an integer value of `0`, `1`, or `2` that determines how the offset value is used to set the size of the file. These values have the following effects:

`0`	sets the file pointer to *offset*.
`1`	sets the file pointer to the current location of the file pointer plus *offset*.
`2`	sets the file pointer to the current size plus *offset*.

newpos an integer vector of length two which is the file pointer location as measured in bytes from the beginning of the file.

Discussion

Use **eseek**() to move the read/write file pointer in the file designated by the *unit* parameter, which should be the value you assigned the *unit* parameter of **open**(). The *offset* parameter is a value in bytes that depends upon the value of the *whence* parameter. If *whence* is `0`, the pointer is set to *offset* bytes from the beginning of the file. If *whence* is `1`, the pointer is set to the *offset* added to the current location of the file pointer. If *whence* is `2`, the pointer is set to the size of the file plus *offset* (the units are bytes.)

If successful, the location of the new file pointer is returned in units of bytes.

ESEEK() *(cont.)*

ESEEK() *(cont.)*

Errors

`eseek: Bad file number`

Use the unit number assigned by **open()**.

`eseek: No such unit`

Unit must be a value no larger than `100`.

`eseek: Mixed file operations`

In I/O mode 2 or 3, nodes are attempting different operations (reads and writes) to a shared file. Ensure that nodes are performing the same operation.

`eseek: Not implemented on the SRM`

Available only for CFS files.

See Also

eseek(), eadd(), lseek(), cread(), cwrite(), iread(), iwrite()
iPSC®/860 Fortran Language Reference Manual: **OPEN()**

ESIZE() ESIZE()

Increase the size of a file.

Synopsis

SUBROUTINE **ESIZE**(*unit, offset, whence, newsize*)

Parameter Declarations

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

Return Values

None

Example

The following example increases the size of the file designated by the variable *unit1* (the value you assigned the *unit* variable of the Fortran **open()** routine) to 4G bytes.

```
      INTEGER UNIT1
      INTEGER OFFSET(2), NSIZE(2)
C  LOW BITS
      OFFSET(1) = 0
C  HIGH BITS
      OFFSET(2) = 1
C  INCREASE FILE SIZE TO 4G BYTES
      CALL ESIZE(UNIT1,OFFSET,0,NSIZE)
```

ESIZE() *(cont.)*

ESIZE() *(cont.)*

Environment

Node

Languages

C, Fortran

Description of Parameters

unit the unit you assign the file with the **open**() call.

offset value of an integer vector of length 2 that affects the current size of the file depending upon the value of *whence*.

whence an integer value of `0, 1,` or `2` that determines how the offset value is used to set the size of the file. These values have the following effects:

`0`	sets the file size to the greater of the current size or *offset*.
`1`	sets the file size to the greater of the current size or the current location of the file pointer plus *offset*.
`2`	sets the file size to the greater of the current size or the current size plus *offset*.

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

newsize an integer vector of length 2 representing the file size in bytes

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

Discussion

The **esize**() call changes the size of the file designated by the *unit* parameter, which should be the value you assigned the *unit* parameter of the **open**() call when you opened the file. The *offset* parameter is a value in bytes that, depending upon the value of the *whence* parameter, is used to determine the resulting size of the file. If *whence* is `0`, the file size is the *offset* value. If *whence* is `1` and *offset* is greater than the file size, then the resulting file size is the *offset* added to the current location of the file pointer. However, this is not true if *offset* is less than the file size. If *whence* is `2`, the file size is the current file size plus *offset*.

The primary use of **esize**() is to ensure that enough space is allocated before you do any sensitive calculations. If *offset* is positive, **esize**() ensures that the disk blocks are allocated to the file from the starting position to the ending position (as determined by *whence*, *newsize*, and *offset*) If allocation has been restricted by **restrictvol**(), any new disk blocks allocated will come from the restricted set.

Errors

`esize: Bad file number`

Use the file descriptor returned by **open**().

`esize: No space left on device`

Not enough space on device to which you are writing. Create more space in file system, or if you have restricted the disks to which you are writing with **restrictvol**(), re-execute **restrictvol**() to include more drives.

`esize: Not enough core`

Not enough memory on node for operation.

`esize: Not implemented on the SRM`

Available only for CFS files.

See Also

eseek(), **eadd**(), **restrictvol**(), **lsize**()
iPSC®/860 Fortran Language Reference Manual: **OPEN**()

ETOS(), STOE() ETOS(),STOE()

Converts between extended numbers and strings.

Synopsis

SUBROUTINE **ETOS**(*e, s*)
SUBROUTINE **STOE**(*s, e*)

Parameter Declarations

ETOS()	**INTEGER** *e(2)*
	CHARACTER *s(*)*
STOE()	**CHARACTER** *s(*)*
	INTEGER *e(2)*

Return Values

None

Example

The following example displays a 5G-byte value in the form of a string. Then it uses that string to construct an extended integer.

```
      INTEGER E(2), EE(2)
      CHARACTER*20 S
      E(1) = 1073741824
      E(2) = 1
      CALL ETOS(E, S)
      PRINT *, '5G BYTES ', S
C     DISPLAYS THE STRING 5368709120
      CALL STOE(S,EE)
      PRINT *,EE(1),' ',EE(2)
      END
```

ETOS(), STOE() *(cont.)*

ETOS(), STOE() *(cont.)*

Environment

Node, Host

Languages

C, Fortran

Description of Parameters

e	extended number.
s	character string.

Discussion

The **etos()** routine converts an integer vector of length two to a character string. The **stoe()** routine converts a string of characters to an integer vector of length two. These calls support the extended file sizes in CFS.

Errors

`etos: Invalid size`

Size should be less than 2^{60} or overflow occurs.

See Also

eadd(), esub(), ediv(), emul(), ecmp(), stoe()

FLICK() FLICK()

Relinquish node processor to another process.

Synopsis

SUBROUTINE **FLICK**()

Parameter Declarations

None

Return Value

None

Example

The following example illustrates the proper use of **flick**(). Two asynchronous receives are posted, each of which return a message id (*ID1* and *ID2*). After some other processing, you enter a DO-loop that tests to see if either message has been received, and if so, process the one received. If not, however, you use **flick**() to defer execution of this process, to allow another process on the same node to execute while waiting for these messages to be received.

```
      INTEGER NODE, PID
      INTEGER BUF1(2), BUF2(10)
      INTEGER ID1, ID2

      ID1 = IRECV(101, BUF1, LEN(BUF1))
      ID2 = IRECV(102, BUF2, LEN(BUF2))
C     DO SOME OTHER PROCESSING HERE.
          .
          .
C     NOW, WAIT FOR EITHER MESSAGE TO BE RECEIVED.

10     IF ((MSGDONE(ID1).NE.0) THEN
```

FLICK() *(cont.)*

FLICK() *(cont.)*

```
      .
C     PROCESS MESSAGE 1
      .
       ELSE IF(MSGDONE(ID2).NE.0) THEN
      .
C     PROCESS MESSAGE 2
      .
       ELSE
       CALL FLICK()
       ENDIF
       GOTO 10
```

Environment

Host, Node

Languages

C, Fortran

Discussion

You should use the **flick()** routine when the current process has no useful work to do (e.g., when a process is waiting for any of several **irecv**'s to complete). It is recommended that a process never "busy loop" without flicking. The **flick()** call is a delay mechanism, not a synchronization mechanism. The scheduler does not know the conditions for which a process is waiting.

CX Node Description

The **flick()** call temporarily relinquishes the node processor to another process on the same node. The scheduler runs processes on a node in round-robin fashion, running each process until it blocks in a **csend()**, **crecv()**, **cprobe()**, or **msgwait()**, or for approximately 50 milliseconds, or until the process gives up the processor with the **flick()** routine. The execution of the process is not deferred indefinitely. Instead, the process will resume execution during its next scheduled time frame. The **flick()** call simply allows other node processes to execute. The **csend()**, **crecv()**, **cprobe()**, and **msgwait()** operations also defer execution of a calling process until they complete their task.

FLICK() *(cont.)*

FLICK() *(cont.)*

RX Node Description

The **flick()** call relinquishes the node processor until an interrupt of any kind occurs within NX. It is then scheduled again for processing. The execution of the process can be deferred up to a maximum of 10 ms. Using **flick()** in this way can provide higher performance, because part of the interrupt overhead is eliminated.

Host Description

On a host machine, **flick()** performs no operation. It is included so that you can write programs that call **flick()** and still be able to run them on a host machine.

Errors

None

See Also

UNIX System V Programmer's Reference Manual: **sleep()**

FLUSHMSG() FLUSHMSG()

Flush specified messages from the system.

Synopsis

SUBROUTINE **FLUSHMSG**(*typesel, node, pid*)

Parameter Declarations

INTEGER *typesel, node, pid*

Return Value

None

Example

This example sends two synchronous messages followed by one asynchronous message to process 0 on node 1. It then flushes all messages of all types waiting to be received for that process and node. Issuing **msgcancel()** to cancel the asynchronous message (*MSG3*) before the **flushmsg()** call ensures that all messages are canceled.

```
      CHARACTER MSG1, MSG2, MSG3
      INTEGER NODE, PID, MSGID
            .
            .
      NODE = 1
C     FOR RX NODES, PID MUST BE 0.
      PID = 0

      CALL CSEND(10, MSG1, LEN(MSG1), NODE, PID)
      CALL CSEND(11, MSG2, LEN(MSG2), NODE, PID)
      MSGID = ISEND(12, MSG3, LEN(MSG3), NODE, PID)
            .
            .
      CALL MSGCANCEL(MSGID)
      CALL FLUSHMSG(-1, NODE, PID)
```

FLUSHMSG() *(cont.)*

FLUSHMSG() *(cont.)*

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

typesel is an integer value that specifies the type(s) of message that you want to flush. The type depends on the type specified in the send. 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, messages of the specified type will be flushed. All other messages will be ignored.
- If *typesel* is `-1`, all messages to the node process will be flushed.
- If *typesel* is any negative number other than `-1`, a set of message types will be flushed. Refer to Appendix B for information on how to build a *typesel* mask.

node is a positive integer that identifies the receiving node or `-1` to indicate all nodes and the host. Nodes within a partition are numbered consecutively from 0. If you want to flush messages just from the host, use **myhost()** to determine the host's id.

pid For CX nodes, *pid* is an integer that specifies the receiving process id or `-1` to indicate all processes. Valid *pids* include any integer value, but negative numbers other than `-1` are reserved for system programs. A node *pid* is assigned when you load a process and a host *pid* is assigned with **setpid()**. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on the RX node).

FLUSHMSG() *(cont.)*

FLUSHMSG() *(cont.)*

Discussion

Use **flushmsg()** to flush all messages of the specified type from system buffers. All messages of the type(s) selected by the *typesel* in the selected node(s) that have been sent to the selected *pid(s)* will be eliminated from the system.

Use of **flushmsg()** has no effect on processes. It also has no effect on messages in transit to processes which do not have the selected *pid*, even if they were sent by a process which does have a matching *pid*. That is, **flushmsg()** flushes messages by the destination *pid*, not the source *pid*. If a particular node is selected, only messages sent to that node which have arrived on that node but have not been received are affected. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on the RX node).

To ensure that all messages are flushed, use **flushmsg()** in conjunction with **msgcancel()** to ensure that all specified messages are flushed.

Errors

```
flushmsg: invalid node
```

Use **numnodes()** to determine cube size and **myhost()** to determine host number.

See Also

msgcancel(), **killcube()**

FORCEFLUSH() FORCEFLUSH()

Forces all buffered I/O to be flushed when an exception occurs.

Synopsis

SUBROUTINE **FORCEFLUSH**()

Parameter Declarations

None

Return Values

None

Example

The following node program sets up a signal handler to be executed when an exception occurs. If an exception occurs, the signal handler is called to flush all outstanding I/O buffers.

```
      INTEGER A(100)
      INTEGER R, OLDMASK
      R = 0
      DO 100 I=1,100
         A(I) = I
100   CONTINUE
      OLDMASK =  FPSETMASK(4)
      CALL FORCEFLUSH()
      OPEN(UNIT = 12,FILE = '/USR/YOU/REMOTE/FGLOBAL/MYFLUSH',
     X     STATUS = 'NEW')
      WRITE(12,7) (A(I), I=1,100)
7     FORMAT(10(I3,1X))
C     THE NEXT STATEMENT WILL CAUSE A DIVIDE-BY-ZERO
C     EXCEPTION, BUT BECAUSE FORCEFLUSH HAS BEEN INVOKED,
C     ALL 100 ELEMENTS WILL BE WRITTEN TO THE FILE
      R = 1/R
      END
```

FORCEFLUSH() *(cont.)*

FORCEFLUSH() *(cont.)*

Environment

Node

Languages

Fortran

Description of Parameters

None

Discussion

This routine sets up a signal handler to be executed when an exception occurs. If an exception occurs, then before the program terminates, the signal handler is called to flush all outstanding I/O buffers. This routine may be used with the subroutine **fpsetmask()** to complete all I/O in the event of a floating-point exception.

The **forceflush()** call needs to be called anytime before the exception occurs. Note that writes to the terminal are not buffered.

Errors

None

See Also

fpsetmask()

FORFLUSH() FORFLUSH()

Forces completion of all buffered I/O on a file specified by its unit number.

Synopsis

SUBROUTINE **FORFLUSH**(*iunit*)

Parameter Declarations

INTEGER *iunit*

Return Values

None

Example

The following example forces all buffered I/O in *A(100)* to the file *JUNK6*

```
      INTEGER A(100)

      DO 100 I=1,100
         A(I) = I
100   CONTINUE

      OPEN(8,FILE = 'JUNK',STATUS = 'NEW')
      WRITE(8,700) (A(I), I = 1,100)
700   FORMAT(10(I3,1X))
      CALL FORFLUSH(8)
      END
```

Environment

Node

Languages

Fortran

FORFLUSH() *(cont.)*

FORFLUSH() *(cont.)*

Description of Parameters

iunit the unit file number.

Discussion

FORFLUSH() forces all buffered I/O in *iunit* to the associated file.

Errors

```
lsize: Bad file number
```

Use the file descriptor returned by **open()**.

See Also

forceflush()

FPSETMASK() FPSETMASK()

Set the exception mask.

Synopsis

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

Parameter Declarations

INTEGER *mask*

Return Values

None

Example

This example sets the 387™ exception mask.

```
      MASK = 1 + 2 + 4 + 8
      OLDMASK =  FPSETMASK(MASK)
C     ENABLE FLOATING-POINT EXCEPTIONS FOR
C     INVALID OPERATIONS, DENORMALIZATION, DIVIDE-BY-ZERO,
C     AND OVERFLOW EXCEPTIONS.
```

Environment

Node

Languages

Fortran

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

Description of Parameters

mask the following bits correspond to enabled floating-point exceptions:

0	All floating-point interrupts disabled (the default).
1	Invalid operation (e.g., sqrt (-(.0))
2	Denormalization exception. Refer to the IEEE standard for an explanation of the Denormalization exception.
4	Divide-by-zero exception.
8	Overflow exception.
16	Underflow exception.

Discussion

This call sets the exception mask. It is available on both host , 386 nodes, and i860 nodes, but not on SX nodes. For 386 nodes, look at */usr/include/ieeefp.h* for a list of exceptions. For i860 nodes, look at */usr/i860/include/include/ieeefp.h* for a list of exceptions.

Errors

None

See Also

UNIX System V/386 Programmers Reference Manual: **fpgetround(3c)**

GCOL() GCOL()

Global concatenation operation.

Synopsis

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

Parameter Declaration

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

Return Values

None

Example

This example assumes that a vector of length *N* is distributed among the nodes of the cube, where *X(M)* is a piece of the vector on a given node. Note that *M* need not have the same value on all nodes. After computing the dot product and summing the global contributions, the vector must be normalized and then the entire vector must be available to all nodes, using **gcol()** to collect the vector. The array *Y* in each node then contains all of the vector *X*. The order of *X's* in *Y* is by increasing node number. This example is simplified, and the way in which the norm is computed in this example can lead to problems with numerical underflow or overflow. Safer and more complicated implementations are possible and ought to be used in practice.

GCOL() *(cont.)*

GCOL() *(cont.)*

```
      INTEGER I, MYNODE, CNT, DPSIZE
      DOUBLE PRECISION X(M), Y(N), DOT, NORM, DUMMY
      PARAMETER (DPSIZE = 8)
C     Assume that X, M, and N have been initialized
C     Compute the local dot product
      DOT = 0.0
      DO 1 I = 1, M
         DOT = DOT + X(I)*X(I)
1     CONTINUE
C     Sum global contributions
      CALL GDSUM (DOT, 1, DUMMY)
C     Compute global NORM
      NORM = DSQRT(DOT)
C     Do the local normalization
      DO 2 I = 1, M
         X(I) = X(I)/NORM
2     CONTINUE
C     Collect vector
      CALL GCOL(X,M*DPSIZE,Y, N*DPSIZE, CNT)
```

Environment

Node

Languages

C, Fortran

Description of Parameters

x	refers to the input buffer to be used in the operation. Note that *x* may be of any type.
xlen	the length of input buffer *x* (in bytes).
y	refers to the output buffer to be used in the operation (*y* contains the desired result). Note that *y* must be of the same data type as *x*.
ylen	the length of output buffer *y* (in bytes).
ncnt	refers to the number of bytes returned in *y*.

GCOL() *(cont.)*

GCOL() *(cont.)*

Discussion

Use **gcol()** to collect and concatenate a contribution from each node in the current cube and to ensure that contributions are received in order. The *x* and *y* parameters can be of any data type as long as they are of the same data type. The result is returned in *y* to every node. All participating processes must have the same *pid*. This is a global operation, meaning that all nodes in the cube must issue this command before the execution of the process can continue. For RX nodes, *pid* 0 is the only valid process ID (since only a single process can run on the RX node).

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

Errors

`gcol: Received message too long for buffer`

Make sure buffer is long enough.

`gcol: Invalid buffer pointer`

Specify a pointer which contains the address of a valid buffer.

`gcol: Invalid length`

Use a non-negative number for *xlen* or *ylen*

`gcol: Buffer length exceeds allocation`

Make sure buffers are large enough and length parameters are not greater than buffer size.

GCOLX() GCOLX()

Global concatenation operation for contributions of known length.

Synopsis

SUBROUTINE **GCOLX**(*x, xlens, y*)

Parameter Declaration

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

Return Values

None

Example

This example assumes that a vector of length *N* is distributed among the 16 nodes of the cube, where *X(M)* is a piece of the vector on a given node. Note that *M* need not have the same value on all nodes (but for this example, we assume that it does). After computing the dot product and summing the global contributions, the vector must be normalized and then the entire vector must be available to all nodes, using **gcolx()** to collect the vector. The contents of the *XLENS* array used by **gcolx()** is easy to calculate, since we assume *X* to be the same length on every node. The array *Y* in each node then contains all of the vector *X*. The number of bytes returned in *Y* is simply the sum of the elements in *XLENS*. The order of *X*'s in *Y* is by increasing node number. This example is simplified, and the way in which the norm is computed in this example can lead to problems with numerical underflow or overflow. Safer and more complicated implementations are possible and ought to be used in practice.

```
      INTEGER I, DPSIZE, NUM_NODES
      DOUBLE PRECISION X(M), Y(N), DOT, NORM, DUMMY
      PARAMETER (DPSIZE = 8)
      PARAMETER (NUM_NODES = 16)
      INTEGER XLENS(NUM_NODES), XLEN
C     Assume that X, M, and N have been initialized
C     Compute the local dot product
      DOT = 0.0
      DO 1 I = 1, M
          DOT = DOT + X(I)*X(I)
```

GCOLX() *(cont.)*

GCOLX() *(cont.)*

```
1     CONTINUE
C  Sum global contributions
      CALL GDSUM (DOT, 1, DUMMY)
C  Compute global NORM
      NORM = DSQRT(DOT)
C  Do the local normalization
      DO 2 I = 1, M
          X(I) = X(I)/NORM
2     CONTINUE
      XLEN = M*DPSIZE
C  Get the length of all contributions into XLENS.
      DO 3 I = 1, NUM_NODES
          XLENS(I) = XLEN
3     CONTINUE
C  Collect the vector
      CALL GCOLX(X,XLENS,Y)
```

Environment

Node

Languages

C, Fortran

Description of Parameters

x	refers to the input buffer to be used in the operation. Note that *X* may be of any type.
xlens	an array containing the length (in bytes) of the input buffer *X* expected from each node.
y	refers to the output buffer to be used in the operation (*Y* contains the desired result). Note that *Y* must be of the same data type as *X*.

GCOLX() *(cont.)*

GCOLX() *(cont.)*

Discussion

Use **gcolx()** to globally collect and concatenate a contribution of specified length from each node in node number order. By providing the expected length of each contribution, **gcolx** improves the speed of this operation compared to **gcol()** due to the reduced overhead of calculating where each contribution belongs in the output buffer. The elements in *XLENS* are in increasing node number order. The *X* and *Y* parameters can be of any data type as long as they are of the same data type. The result is returned in *Y* to every node. All participating processes must have the same *pid*. This is a global operation, meaning that all nodes in the cube must issue this command before the execution of the process can continue.

If the lengths of the contributions from all the nodes is unknown, use the **gcol()** routine.

Errors

`gcolx: Received message too long for buffer`

Make sure buffer is long enough.

`gcolx: Invalid length`

Use a non-negative number for *XLENS*.

`gcolx: Buffer length exceeds allocation`

Make sure buffers are large enough and length parameters are not greater than buffer size.

See Also

gcol()

GETCUBE() GETCUBE()

Allocate a cube. This is intended for host programs and is not available for node programs.

Synopsis

SUBROUTINE **GETCUBE**(*cubename, cubetype, srmname, keep, account*)

Parameter Declarations

CHARACTER *cubename*(*)*
CHARACTER *cubetype*(*)*
CHARACTER *srmname*(*)*
INTEGER *keep*
CHARACTER *account*(*)*

Return Values

None

Example

This example gets a cube named "alpha," to be made up of 16 nodes of type vector (vx). By assigning no name to the host connected to the cube you want, the name defaults to that of the System Resource Manager. A *keep* value of 0 releases the cube and terminates the node processes when the calling process exits or is killed.

```
CALL GETCUBE('ALPHA', '16vx', '', 0,'')
```

Environment

Host

Languages

C, Fortran

GETCUBE() *(cont.)*

GETCUBE() *(cont.)*

Description of Parameters

cubename specifies the name of the cube that you want to allocate. Valid names include any ASCII string, 15 characters or less. Names longer than 15 characters will be truncated to 15 characters. If the name is a null string, "defaultname" is used. If you specify a name that you (or anyone using your login name) are already using, it is an error. Use **cubeinfo()** or the **cubeinfo** command to find out the names of allocated cubes.

cubetype specifies the size and type of the cube you want to allocate. Specify size first followed by type in a contiguous string (no spaces). Without this option, you get the largest available cube. The number of nodes allocated will always be a power of two. If the size you specify is not a power of two, it will be rounded up.

Size can be specified in one of two ways:

n	where *n* is the number of nodes (for example, 8) or
d*n*	where **d** indicates dimension and *n* is the size of the dimension (for example, d3)

Specifying no type gets a cube of any type. Other choices are:

rx	i860-based compute nodes
cx	386-based compute nodes
vx	nodes with vector companion board
sx	nodes with SX installed
f	nodes with 387 coprocessor installed
b	disable messsage buffering
m1	nodes with at least 1M byte of memory each
m4	nodes with at least 4M bytes of memory each
m8	nodes with at least 8M bytes of memory each
m16	nodes with at least 16M bytes of memory each
m32	nodes with at least 32M bytes of memory each

GETCUBE() *(cont.)*

GETCUBE() *(cont.)*

	m64	nodes with 64M bytes of memory each
	n*s*	gets a continuous set of nodes starting with slot *s*

srmname specifies the name of the host connected to the cube that you want to allocate. If you are on a System Resource Manager, the name defaults to your System Resource Manager's name. Other *srmnames* are ignored. If you are on a remote host, specify the *srmname* that you want the cube to be allocated from. If *srmname* is not specified on a remote host, **getcube**() will select an available System Resource Manager for you from the list in the */usr/ipsc/lib/srms* file. The *srmname* must be the network host name of a System Resource Manager with an attached cube to be valid.

keep is one of two settings that allow you to choose either to keep the cube only until the process is terminated or until you release it. The possible settings are:

`0`	cube is released and all node processes are terminated when the process that invoked **getcube** exits or is killed.
`1`	cube is not released until the owner releases the cube with **relcube**() or the **relcube** command.

account is a pointer to a character string. Unlike the C version, you cannot put in 0 as a null pointer. This call interprets two contiguous single quotes as a null string.

Discussion

Use **getcube**() to allocate a new cube, assigning a name, type, host, and lifetime to the cube. It also starts a file server that will handle I/O (read, write and print) operations for the allocated cube. You assign names to cubes when you invoke **getcube**() so that the system will know which cube a program will be connected to. All cubes allocated by a single user must have unique names.

The *cubetype* parameter specifies the size and, if desired, the type of cube you want allocated. Cubes are always allocated such that the number of nodes is a power of two. Thus, in addition to the higher powers of two, you may allocate a cube composed of 0 or 1 node. In addition to the type of node, you may also specify the minimum required memory amount, so that the nodes allocated will have at least the required memory, but may have more.

So, `2` specifies a two-node cube in which the nodes can be of any type, `16vx` specifies a 16-node vector cube, and `d3m4` specifies an eight-node cube in which each node has at least 4M bytes of memory per node. To get a cube composed of different types of nodes, specify the different sizes and types, separated by a slash (/), in one string. For example, `4vx/4m8` gets four vector nodes and four nodes with at least 8M bytes of memory each.

GETCUBE() *(cont.)*

GETCUBE() *(cont.)*

You are automatically attached to the last cube allocated. Subsequent message passing and loader calls made by the process which called **getcube()** will apply to the newly allocated cube. Use **attachcube()** when you want to access a different allocated cube.

All the cubes that you own are automatically released when you log out of the SRM unless you invoke your program with **nohup**. To invoke your program with **nohup** enter:

```
nohup progname &
```

Logging out of a remote host does not automatically release cubes.

If any of the parameters are not specified, default values are used. With no arguments, **getcube()** will reserve a cube for you with the name, "defaultname." The number of nodes reserved for this cube will be the largest integral power of two that is possible given the existing set of unallocated nodes.

If the cube cannot be allocated, an error message results and the process will terminate. Use **cubeinfo()** to determine the size (as well as other information) of the cube that was allocated.

When **getcube()** is executed, it calls the routine called **account_hook()**. Exactly what **account_hook()** does is specific to your installation. The default version of **account_hook()** does no accounting. If you change **account_hook()** to write to standard output, the writes go to */usr/ipsc/commser.log*. The parameter *account* is passed to **account_hook()** as an account name. If *account* is null, the parameter passed is the string *defaultaccountname*.

The source for **account_hook()** is provided. Modifying **account_hook()** is a superuser activity and should only be performed by the system administrator. The system administrator may modify **account_hook()** to perform other actions that control and monitor node usage. For example, one possible use of **account_hook()** is to verify the user requesting node allocation and deny allocation to users with insufficient privilege.

Errors

`getcube: Internal cube usage limit`

Limit of 10 cubes may be allocated. Try again later when system is not as busy.

`getcube: permission denied`

If the accounting routines reject a user, this message is given.

GETCUBE() *(cont.)*

GETCUBE() *(cont.)*

`getcube: Commser not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with the **bootcube** or **rebootcube** command.

`getcube: Lifeline not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with the **bootcube** or **rebootcube** command.

`getcube: Cubename already exists`

Use a different name.

`getcube: No SRM that matched request was found`

Remote host error only. Could not find an SRM with a cube of the type requested.

`getcube: Commser out of buffers`

Communication server out of buffers.

`getcube: Cubetype not available`

The cubetype is present but currently in use.

`getcube: Cubetype not installed`

The cubetype is not present.

`getcube: Cubetype not valid`

The request is invalid.

See Also

attachcube(), cubeinfo(), newserver(), numnodes(), relcube()

GINV() GINV()

Returns the position of an element in the binary-reflected Gray code sequence.

Synopsis

INTEGER FUNCTION **GINV** (*j*)

Parameter Declarations

INTEGER *j*

Return Value

An integer interpreted as the corresponding position of the gray code value used as input.

Example

This example returns the position of the gray code value 6 and stores it in the variable *X*:

```
INTEGER x
X = GINV(6)
```

The returned value in this case will be a 4.

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

j a positive integer value.

GINV() *(cont.)*

GINV() *(cont.)*

Discussion

The **ginv()** takes a Gray code element (positive integer) as input and returns the position of the element in the binary-reflected Gray code sequence. This is the inverse of **gray()**. For the inverse function, obtaining the corresponding gray code value with an integer value as input, use the **gray()** function. The following table shows the gray code translation for the integer values from `0` through `7`.

```
000   001   011   010   110   111   101   100   Binary Gray code

 0     1     3     2     6     7     5     4    Decimal Gray code

 0     1     2     3     4     5     6     7    Position of each element
```

Errors

None

See Also

gray()

GOPF() GOPF()

Makes a global operation of a user-defined associative and commutative function.

Synopsis

SUBROUTINE **GOPF**(*x, xlen, work, f*)

Parameter Declarations

integer *x(*)*
integer *xlen*
integer *work(*)*
external *f*

Return Value

None

Example

Use of **gopf()** gives an arbitrary predefined associative and commutative function the same properties as the other global communications calls such as **gdsum()**. The function must have two parameters (the input value, and an array for the contributions of other nodes). The following is an example of such a function, called **max_node**, which finds the maximum value among the rows of a matrix and the number of the node that is the owner of the row. Following that is a code fragment that calls **gopf()** to exercise this function. In this example, it is assumed that a matrix is distributed by rows among the processors, and this code finds which element of a distributed vector is the largest (in absolute value).

GOPF() *(cont.)*

GOPF() *(cont.)*

The following functions is separately compiled to an object and that object is linked with the main program.

```
      INTEGER FUNCTION MAX_NODE(MINE,WORK)
      INTEGER N,IAM
      REAL MINE(2), WORK(2)

      IAM = MYNODE()
      IF ( (MINE(1) .LT. WORK(1)) )
     X THEN
         MINE(1) = WORK(1)
         MINE(2) = WORK(2)
       ENDIF
      IF ( (MINE(1) .EQ. WORK(1)) .AND. (MINE(2) .GT. WORK(2)))
     X THEN
         MINE(2) = WORK(2)
      ENDIF
      RETURN
      END
```

Here is the main routine.

```
      INCLUDE 'fcube.h'

      EXTERNAL MAX_NODE
      REAL MINE(2), WORK(2)
      INTEGER N, MAX_NODE, MAX_LOC, IAM
      PARAMETER(N=10)
      REAL X(N)
      IAM = MYNODE()

      DO 100 I=1,N
         X(I) = REAL(I)
100   CONTINUE
      IF(IAM .EQ. 1)X(5) = 25.

      MINE(1) = ABS(X(1))
      MINE(2) = IAM
      MAX_LOC = 1

      DO 200 I=2,N
         IF(MINE(1) .LT. ABS(X(I))) THEN
            MINE(1) = ABS(X(I))
            MAX_LOC = I
         ENDIF
```

GOPF() *(cont.)*

GOPF() *(cont.)*

```
200   CONTINUE

      CALL GOPF(MINE,8,WORK,MAX_NODE)
      IF(MINE(2) .EQ. IAM)  THEN
         PRINT *,CHAR(10),' MAX NODE IS ',IAM
         PRINT *,'MAX VALUE IS ',MINE(1)
      ENDIF
      END
```

Environment

Node

Languages

C, Fortran

Description of Parameters

x	refers to the input vector to be used in the operation. When the operation completes, it contains the final result. Note that *x* can be any type.
xlen	the length of the vector *x* in bytes.
work	refers to an array which is used to receive contributions from other nodes. The length of work must be at least *xlen*.
f	the function to be called. It is called as *f(x, work)*, which sets *x* to whatever function of *x* and *work* is desired.

Discussion

Use **gopf()** to give some function *f()* the same global properties that the other global communications routines, such as **gdsum()**, have. These properties are:

- All nodes must issue **gopf()**, which in turn calls the function
- Each node calculates the result, which is stored in *x*. The work array is used to receive messages from other nodes.
- The result is returned in *x* to all nodes.

GOPF() *(cont.)*

GOPF() *(cont.)*

The function *f()* must be a commutative and associative function of two vectors, returning its result in the first vector. It must have two parameters:

x	the input vector to be used in the operation. When the operation is complete, it contains the final result.
work	an array which is used to receive the other contributions. The length of work must be at least *xlen*.

All participating processes must have the same *pid*. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on RX nodes).

Errors

None

GRAY() GRAY()

Returns the binary-reflected gray code for an integer.

Synopsis

INTEGER FUNCTION **GRAY**(*j*)

Parameter Declarations

INTEGER *j;*

Return Value

An integer that is the binary-reflected Gray code for the input.

Example

To return the gray code value corresponding to a value of `4` and store it in the variable *X*, use the statements:

```
INTEGER X
X = GRAY(4)
```

In this example, the returned value is `6`.

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

j is a positive long integer value.

GRAY() *(cont.)*

GRAY() *(cont.)*

Discussion

The gray code is a useful tool in determining where a particular node is in a given application topology (ring, mesh, tree, or linear array) and in optimally assigning tasks to nodes. **gray** () takes a positive integer (*j*) as input and returns the gray code element corresponding to *j* in the binary-reflected gray code sequence. For the inverse function, obtaining the position from a gray code value, use the **ginv**() call. The following table shows the gray code translation for values from `0` through `7`.

```
000   001   011   010   110   111   101   100   Binary Gray code

 0     1     3     2     6     7     5     4    Decimal Gray code

 0     1     2     3     4     5     6     7    Position of each element
```

Errors

None

See Also

ginv()

GSENDX() GSENDX()

Send a vector to a list of nodes.

Synopsis

SUBROUTINE **GSENDX**(*type, x, xlen, nodenums, nlen*)

Parameter Declaration

integer *type*
integer *x(*)*
integer *xlen*
integer *nodenums(*)*
integer *nlen*

Return Values

None

Environment

Node

Languages

C, Fortran

Description of Parameters

type	message type. Must be the same for all participating processes. There must be no other messages of this type in the system.
x	refers to the input vector to be sent. Note that *x* can be of any type.
xlen	the number of bytes of data in vector *x*.
nodenums	refers to the list of node numbers to which data is to be sent.
nlen	the number of node numbers in the list.

GSENDX() *(cont.)*

GSENDX() *(cont.)*

Discussion

Use **gsendx()** to send a vector to a set of nodes specified by *nodenums*. All the nodes that are to receive the vector must call **crecv()**, **irecv()**, or **hrecv()** to receive the message, using the type specified for **gsendx()** above. All participating processes must have the same *pid*. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on RX nodes).

Errors

`gsendx: Invalid node`

Use **numnodes()** to determine cube size and **myhost()** to determine host node number.

`gsendx: Invalid buffer pointer`

Specify a pointer which contains the address of a valid vector.

`gsendx: Invalid length`

Use a non-negative number for *len* or *nlen*.

`gsendx: Buffer length exceeds allocation`

Make sure vector is large enough.

See Also

csend(), **isend()**

GSYNC() GSYNC()

Global synchronization operation.

Synopsis

SUBROUTINE **GSYNC**()

Parameter Declarations

None

Environment

Node

Languages

C, Fortran

Discussion

Use **gsync()** to synchronize node processes. This is not a precise synchronization; when a node process executes **gsync()**, it waits until all other nodes in the cube have called **gsync()** before continuing to execute the process. All participating processes must have the same *pid*. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on RX nodes).

Errors

None

See Also

waitall(), waitone()

GxAND() GxAND()

Global AND operation for integer or logical vectors or scalars.

Synopses

Integer (bitwise operation):

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

Logical (logical operation):

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

Return Values

None

Environment

Node

Languages

C, Fortran

Description of Parameters

x	the input vector or scalar to be used in the operation. When the operation is complete, it contains the final result.
n	the number of elements in *x*.

GxAND() *(cont.)*

GxAND() *(cont.)*

work an array which is used to receive the contributions from other nodes. The length of work must be at least *n*.

Discussion

Use **giand()** or **gland()** to calculate the bitwise or logical AND of each integer or logical component (depending on which routine you use) of *x* across all nodes. The result is returned in *x* to every node. Each node receives the results from all the other nodes in the partition in the work array. All participating processes must have the same *pid*. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on RX nodes).

This is a global operation, meaning that all nodes in a partition must execute this operation before the process can continue on each node. When *x* is a vector, each element of the resulting vector represents the accumulation of the corresponding index of the input vector from each node.

Errors

`gxand: Received message too long for buffer`

Make sure *n* is the same for all nodes.

`gxand: Received message too short for buffer`

Make sure *n* is the same for all nodes.

`gxand: Invalid buffer pointer`

Specify a pointer which contains the address of a valid vector.

`gxand: Invalid length`

Use a non-negative number for *n*.

`gxand: Buffer length exceeds allocation`

Make sure vector is large enough.

G*x*HIGH() G*x*HIGH()

Global MAX operation for double precision, integer, or real vectors or scalars.

Synopses

Double precision:

SUBROUTINE **GDHIGH**(*x, n, work*)
double precision *x(*)*
integer *n*
double precision *work(*)*

Integer:

SUBROUTINE **GIHIGH**(*x, n, work*)
integer *x(*)*
integer *n*
integer *work(*)*

Real:

SUBROUTINE **GSHIGH**(*x, n, work*)
real *x(*)*
integer *n*
real *work(*)*

Return Values

None

Example

This example uses **gshigh()** to find the largest value among a list of floating point numbers. The list is distributed among the processors in straightforward domain decomposition; specifically, *list(i)* on processor *p* represents the $(i+B*p)^{th}$ element of the global list, where *B* is the number of elements on each processor.

GxHIGH() *(cont.)*

GxHIGH() *(cont.)*

```
C Returns the largest value among all values of the global list
      REAL FUNCTION FINDMAX(X, NELEM)
      INTEGER NELEM, INDX, K
      REAL X(NELEM)
      REAL FMAX, TEMP
      INDX = 1
C Find largest value for this process
      DO 10 K=2, NELEM
          IF (X(K) .GT. X(INDX)) INDX = K
10    CONTINUE
      FMAX = X(INDX)
C Check across all processors
      CALL GSHIGH(FMAX, 1, TEMP)
      FINDMAX = FMAX
      RETURN
      END
```

Environment

Node

Languages

C, Fortran

Description of Parameters

x	the input vector or scalar to be used in the operation. When the operation is complete, it contains the final result.
n	the number of elements in *x*.
work	an array which is used to receive the contributions from other nodes. The length of work must be at least *n*.

G*x*HIGH() *(cont.)*

G*x*HIGH() *(cont.)*

Discussion

Use **gdhigh()**, **gihigh()**, or **gshigh()** to calculate the maximum value of the double precision, integer, or real component (depending on which routine you use) of *x* across all nodes. The result is returned in *x* to every node. The work array is used to receive messages from other nodes. All participating processes must have the same *pid*. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on RX nodes).

This is a global operation meaning that all nodes in a cube must execute this operation before the process can continue on each node. When *x* is a vector, each element of the resulting vector represents the accumulation of the corresponding index of the input vectors from each node.

Errors

`gxhigh: Received message too long for buffer`

Make sure *n* is the same for all nodes.

`gxhigh: Received message too short for buffer`

Make sure *n* is the same for all nodes.

`gxhigh: Invalid buffer pointer`

Specify a pointer which contains the address of a valid vector.

`gxhigh: Invalid length`

Use a non-negative number for *n*.

`gxhigh: Buffer length exceeds allocation`

Make sure value of *n* does not exceed control vector size.

GxLOW() GxLOW()

Global vector MIN operation for double precision, integer, or real vectors or scalars.

Synopses

Double precision:

SUBROUTINE **GDLOW**(*x, n, work*)
double precision *x(*)*
integer *n*
double precision *work(*)*

Integer:

SUBROUTINE **GILOW**(*x, n, work*)
integer *x(*)*
integer *n*
integer *work(*)*

Real:

SUBROUTINE **GSLOW**(*x, n, work*)
real *x(*)*
integer *n*
real *work(*)*

Return Values

None

G*x*LOW()*(cont.)* G*x*LOW()*(cont.)*

Example

This example uses **gslow()** to find the smallest value among a list of real numbers. The list is distributed among the processors in straightforward domain decomposition; specifically, *x(i)* on processor *p* represents the $(i+B*p)^{th}$ element of the global list, where *B* is the number of elements on each processor.

```
C Returns the smallest value among all values of the global list
      REAL FUNCTION FINDMIN(X, NELEM)
      INTEGER NELEM, INDX, K
      REAL X(NELEM)
      REAL FMIN, TEMP
      INDX = 1
C Find largest value for this process
      DO 10 K=2, NELEM
        .       IF (X(K) .LT. X(INDX)) INDX = K
10    CONTINUE
      FMIN = X(INDX)
C Check across all processors
      CALL GSLOW(FMIN, 1, TEMP)
      FINDMIN = FMIN
      RETURN
      END
```

Environment

Node

Languages

C, Fortran

Description of Parameters

x	the input vector or scalar to be used in the operation. When the operation is complete, it contains the final result.
n	the number of elements in *x*.

GxLOW() *(cont.)* GxLOW() *(cont.)*

work an array which is used to receive the contributions from other nodes. The length of work must be at least *n*.

Discussion

Use **gdlow()**, **gilow()**, or **gslow()** to calculate the minimum value of the double precision, integer, or real component (depending on which routine you use) of *x* across all nodes. The result is returned in *x* to every node. The work array is used to receive messages from other nodes. All participating processes must have the same *pid*. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on RX nodes).

This is a global operation meaning that all nodes in a cube must execute this operation before the process can continue on each node. When *x* is a vector, each element of the resulting vector represents the accumulation of the corresponding index of the input vectors from each node.

Errors

`gxlow: Received message too long for buffer`

Make sure *n* is the same for all nodes.

`gxlow: Received message too short for buffer`

Make sure *n* is the same for all nodes.

`gxlow: Invalid buffer pointer`

Specify a pointer which contains the address of a valid vector.

`gxlow: Invalid length`

Use a non-negative number for *n*.

`gxlow: Buffer length exceeds allocation`

Make sure value of *n* does not exceed control vector size.

GxOR() GxOR()

Global OR operation for integer or logical vectors or scalars.

Synopses

Integer (bitwise operation):

SUBROUTINE **GIOR**(*x, n, work*)
integer *x(*)*
integer *n*
integer *work(*)*

Logical (logical operation):

SUBROUTINE **GLOR**(*x, n, work*)
logical *x(*)*
integer *n*
logical *work(*)*

Return Values

None

Environment

Node

Languages

C, Fortran

Description of Parameters

x	the input vector or scalar to be used in the operation. When the operation is complete, it contains the final result.
n	the number of elements in *x*.

GxOR() *(cont.)*

GxOR() *(cont.)*

work an array which is used to receive the contributions from other nodes. The length of work must be at least *n*.

Discussion

Use **gior()** or **glor()** to calculate the bitwise or logical OR of each integer or logical component (depending on which routine you use) of *x* across all nodes. The result is returned in *x* to every node. The work array is used to receive messages from other nodes. All participating processes must have the same *pid*. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on RX nodes).

This is a global operation, meaning that all nodes in a cube must execute this operation before the process can continue on each node. When *x* is a vector, each element of the resulting vector represents the accumulation of the corresponding index of the input vectors from all nodes.

Errors

`gxor: Received message too long for buffer`

Make sure *n* is the same for all nodes.

`gxor: Received message too short for buffer`

Make sure *n* is the same for all nodes.

`gxor: Invalid buffer pointer`

Specify a pointer which contains the address of a valid vector.

`gxor: Invalid length`

Use a non-negative number for *n*.

`gxor: Buffer length exceeds allocation`

Make sure value of *n* does not exceed control vector size.

G*x*PROD() G*x*PROD()

Global multiplication operation for double precision, integer, or real vectors or scalars.

Synopses

Double precision:

SUBROUTINE **GDPROD**(*x, n, work*)
double precision *x(*)*
integer *n*
double precision *work(*)*

Integer:

SUBROUTINE **GIPROD**(*x, n, work*)
integer *x(*)*
integer *n*
integer *work(*)*

Real:

SUBROUTINE **GSPROD**(*x, n, work*)
real *x(*)*
integer *n*
real *work(*)*

Return Values

None

Environment

Node

Languages

C, Fortran

GxPROD() *(cont.)*

GxPROD() *(cont.)*

Description of Parameters

x	the input vector or scalar to be used in the operation. When the operation is complete, it contains the final result.
n	the number of elements in *x*.
work	an array which is used to receive the contributions from other nodes. The length of work must be at least *n*.

Discussion

Use **gdprod()**, **giprod()**, or **gsprod()** to calculate the product of the double precision, integer, or real component (depending on which routine you use) of *x* across all nodes. The result is returned in *x* to every node. The work array is used to receive messages from other nodes. All participating processes must have the same *pid*. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on RX nodes).

This is a global operation, meaning that all nodes in a cube must execute this operation before the process can continue on each node. When *x* is a vector, each element of the resulting vector represents the accumulation of the corresponding elements of the input vectors from other nodes.

Errors

```
gxprod: Received message too long for buffer
```

Make sure *n* is the same for all nodes.

```
gxprod: Received message too short for buffer
```

Make sure *n* is the same for all nodes.

```
gxprod: Invalid buffer pointer
```

Specify a pointer which contains the address of a valid vector.

GxPROD() *(cont.)*

GxPROD() *(cont.)*

```
gxprod: Invalid length
```

Use a non-negative number for *n*.

```
gxprod: Buffer length exceeds allocation
```

Make sure value of *n* does not exceed control vector size.

GxSUM() GxSUM()

Global sum operation for double precision, integer, or real vectors or scalars.

Synopses

Double precision:

SUBROUTINE **GDSUM**(*x, n, work*)
double precision *x(*)*
integer *n*
double precision *work(*)*

Integer:

SUBROUTINE **GISUM**(*x, n, work*)
integer *x(*)*
integer *n*
integer *work(*)*

Real:

SUBROUTINE **GSSUM**(*x, n, work*)
real *x(*)*
integer *n*
real *work(*)*

Return Values

None

G*x*SUM() *(cont.)*

G*x*SUM() *(cont.)*

Example

This example uses **gdsum**() to find the sum of the dot products of a vector of length *N* that is distributed among the nodes of the cube, where *X*(*M*) is a piece of the vector on a given node. Note that *M* need not have the same value on all nodes.

```
      INTEGER I
      DOUBLE PRECISION X(M), DOT, DUMMY
C Assume that X and M have been initialized
C Compute the local dot product
      DOT = 0
      DO 1 I = 1, M
          DOT = DOT + X(I)*X(I)
1     CONTINUE
C Sum global contributions
      CALL GDSUM(DOT, 1, DUMMY)
```

Environment

Node

Languages

C, Fortran

Description of Parameters

x	the input vector or scalar to be used in the operation. When the operation is complete, it contains the final result.
n	the number of elements in *x*.
work	an array which is used to receive the contributions from other nodes. The length of work must be at least *n*.

G*x*SUM() *(cont.)*

G*x*SUM() *(cont.)*

Discussion

Use **gdsum()**, **gisum()**, or **gssum()** to calculate the sum of the double precision, integer, or real component (depending on which routine you use) of *x* across all nodes. The result is returned in *x* to every node. The work array is used to receive messages from other nodes. All participating processes must have the same *pid*. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on RX nodes).

This is a global operation, meaning that all nodes in a partition must execute this operation before the process can continue on each node. When *x* is a vector, each element of the resulting vector represents the accumulation of the corresponding element of the input vectors from each node.

Errors

`gxsum: Received message too long for buffer`

Make sure *n* is the same for all nodes.

`gxsum: Received message too short for buffer`

Make sure *n* is the same for all nodes.

`gxsum: Invalid buffer pointer`

Specify a pointer which contains the address of a valid vector.

`gxsum: Invalid length`

Use a non-negative number for *n*.

`gxsum: Buffer length exceeds allocation`

Make sure value of *n* does not exceed control vector size.

GxXOR() GxXOR()

Global exclusive OR operation for integer or logical vectors or scalars.

Synopses

Integer (bitwise operation):

SUBROUTINE **GIXOR**(*x, n, work*)
integer *x(*)*
integer *n*
integer *work(*)*

Logical (logical operation):

SUBROUTINE **GLXOR**(*x, n, work*)
logical *x(*)*
integer *n*
logical *work(*)*

Return Values

None

Environment

Node

Languages

C, Fortran

Description of Parameters

x	the input vector or scalar to be used in the operation. When the operation is complete, it contains the final result.
n	the number of elements in *x*.

GxXOR() *(cont.)*

GxXOR() *(cont.)*

work an array which is used to receive the contributions from other nodes. The length of work must be at least *n*.

Discussion

Use **gixor()** or **glxor()** to calculate the bitwise or logical exclusive OR of each integer or logical component (depending on which routine you use) of *x* across all nodes. The result is returned in *x* to every node. The work array is used to receive messages from other nodes. All participating processes must have the same *pid*. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on RX nodes).

This is a global operation, meaning that all nodes in a cube must execute this operation before the process can continue on each node. When *x* is a vector, each element of the resulting vector represents the accumulation of the corresponding index of the input vectors from all nodes.

Errors

`gxxor: Received message too long for buffer`

Make sure *n* is the same for all nodes.

`gxxor: Received message too short for buffer`

Make sure *n* is the same for all nodes.

`gxxor: Invalid buffer pointer`

Specify a pointer which contains the address of a valid vector.

`gxxor: Invalid length`

Use a non-negative number for *n*.

`gxxor: Buffer length exceeds allocation`

Make sure value of *n* does not exceed control vector size.

HANDLER() HANDLER()

Provide a user-written handler for program failures.

Synopsis

SUBROUTINE **HANDLER** (*etype, proc*)

Parameter Declarations

INTEGER *etype*
EXTERNAL *proc*

Return Value

None

Example

The following example for a CX node casues an exception handler *dbz* to be executed if an attempt to divide by zero is made (exception 0).

```
PROGRAM HANDLER
EXTERNAL DBZ
INTEGER U,L,ANS
U=1
L=0

CALL HANDLER(0,DBZ)
ANS = U/L
PRINT *,ANS
END
```

The handler must be written in C. The name of the C function must have an appended underscore as follows.

```
long dbz_(ecode)
long ecode;
{
    printf(" %d: Alas, a divzero \n");
    exit(1);
}
```

HANDLER() *(cont.)*

HANDLER() *(cont.)*

Environment

Node

Languages

C, Fortran

Description of Parameters

etype — the type of exception for which a user-written handler is provided. The *etype* is an integer between 0 and 31. The "Discussion" section of this manual page describes the meanings of these integers.

proc — the procedure executed when the specified exception occurs. This procedure must be user-written and must be a C procedure with one argument. The argument is the hardware error code generated by the hardware depending upon the exception type selected.

For types other than 11-14, there is no code, and the argument passed to the procedure will be 0. For type 31, the argument passed is the current **mclock()** value. For exception handler types 11-14, the argument will be the hardware error code explained in the *Intel 386™ Programmer's Reference Manual.* For RX node operation, the only implemented exception types are 3, 6, 13, and 14. For these exception types, the argument passed to the procedure is 0.

Discussion

Use **handler()** to allow a node process to execute a user-written exception handler when a specific exception occurs. The handler must be written in C and have one argument, as discussed above. Only the process calling **handler()** can use the exception handler to which it refers.

For CX node operation, use the *proc* parameter as described above in the Description of Parameters. For the RX node, the only implemented exception types are 3, 6, 13, and 14. The preferred method of handling signals is to use the UNIX **signal()** call. Refer to the *UNIX System V Programmer's Reference Manual* for a discussion.

HANDLER() *(cont.)*

HANDLER() *(cont.)*

If an exception occurs for which no handler is specified, the default action is to print a message and kill the process. Exception types 0-31 indicate program failure. Exception types 2, 8, and 17-31 are reserved. Available hardware exception types are as follows:

0 **Divide Error.** Occurs if an integer quotient is too large or an attempt is made to divide by zero.

1 **Single Step.** Allows programs to execute one instruction at a time.

Hardware exception types *(cont.)*:

3 **Break Point.** Occurs when the one-byte breakpoint instruction (INT 3) is executed.

4 **Overflow.** Occurs when the **INTO** instruction is executed when the overflow bit of the flags register is set.

5 **Bounds Check.** Occurs when the **BOUND** instruction is executed when the specified array index is found to be invalid with respect to the given array bounds.

6 **Invalid Opcode.** Occurs if execution of an invalid opcode is attempted. May also occur when the effective address given by certain instructions is a register rather than a memory location.

7 **Processor Extension Not Available.** Occurs when an 80387 floating point instruction is attempted on a node which does not have an 80387.

9 **Processor Extension Segment Overrun.** Occurs when a processor extension memory operand is not completely contained in a single segment.

11 **Segment or Call Gate Not Present.** Occurs when an attempt is made to load a nonpresent segment or to use a control descriptor that is marked not-present.

13 **General Protection Fault.** Occurs when a violation occurs which is not covered by another interrupt.

14 **Page Fault.** Occurs when the application makes a memory reference to a location which has not been allocated to the process.

16 **Processor Extension Error.** Occurs after the numeric instruction that caused the error. On the iPSC system, an exception 16 is always a floating point error.

HANDLER() *(cont.)*

HANDLER() *(cont.)*

Errors

`handler: Invalid handler type`

Specify one of the exception types listed.

See Also

386™ Programmer's Reference Manual
860™ Programmer's Reference Manual
UNIX System V Programmer's Reference Manual: **signal()**

HRECV() HRECV()

Provide user-written receive handler for receive traps.

Synopsis

SUBROUTINE **HRECV** (*typesel, buf, len, proc*)

Parameter Declarations

INTEGER *typesel*
INTEGER *buf(*)*
INTEGER *len*
EXTERNAL *proc*

Return Value

None

Example

This example posts an **hrecv()** to receive message type 101 and on receipt, execute an exception handler named "handleproc." However, following the posting of **hrecv()** is a section of code that must not be interrupted, so **masktrap(0)** is called to prevent the receive handler from being executed until the end of the critical processing, and then is called again to permit normal **hrecv()** operation. If the message type becomes available during the critical processing, it will wait until **masktrap(0)** is called, and then execute "handleproc" immediately.

```
      CHARACTER BUF*1000
      INTEGER OLDMASK
      EXTERNAL HANDLEPROC

      CALL HRECV(101, BUF, LEN(BUF), HANDLEPROC)
C HANDLEPROC WILL BE CALLED WHEN THE RECEIVE IS COMPLETE

      OLDMASK = MASKTRAP(1)
C PROCESSING HERE WILL NOT BE INTERRUPTED

      OLDMASK = MASKTRAP(0)
C HANDLEPROC CAN BE EXECUTED AS SOON AS MESSAGE IS AVAILABLE
```

HRECV() *(cont.)*

HRECV() *(cont.)*

NOTE

The handler must be written in C and have four arguments. If you are calling the handler from a Fortran program, its name must be followed by an underline character. The first line of the handler and the declaration of the parameters would have the following form:

```
handlername_(type, count, node, pid)
long type, count, node, pid;
```

The **hrecv()** routine passes the four values for the parameters to the handler for you so the **info...()** calls are not needed within the handler. The parameters are the message type (*type*), the length of the message (*count*), the node (*node*), and the process id (*pid*).

The handler routine must not do standard input/output. Instead you must use routines that directly invoke UNIX system primitives. For example, you will get a segmentation violation if your C handler attempts to perform a **printf**. Use a **write** to standard output instead.

Environment

Node

Languages

C, Fortran

HRECV() *(cont.)*

HRECV() *(cont.)*

Description of Parameters

typesel is an integer value that specifies the type (or types) of message that you are waiting for. 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 the receive operation will be recognized. After the first message has been received, you can use `-1` again to receive the next message.

If *typesel* is any negative number other than `-1`, a set of message types will be recognized. Refer to Appendix B for instructions on how to build a *typesel* mask.

buf specifies the buffer where the received message will be stored. The buffer can be any legal data type in the language in which you are writing. Data types should match in send and receive operations. *Buf* is not available for use until *proc* is called indicating that the receive has completed.

len is an integer that specifies the size of the message buffer (in bytes) pointed to by *buf*. The maximum number of bytes that you can send in a single message to or from a host process is 256K bytes. There is no limit on message length between nodes.

proc is the procedure executed when the receive is complete. This procedure must be written by the user and must be a C procedure with four arguments.

Discussion

Use **hrecv()** to post a request to receive a message into the buffer, and when receipt of the message is complete, immediately execute a receive trap handler procedure. The name of the handler is the *proc* parameter.

The receive trap handler must be written by the user in C, and must have four arguments: *type*, *count*, *node*, and *pid*. The **hrecv()** call contains **info...()** calls that return information about the message that was received and passes the returned values to the arguments of the handler for you.

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

It is possible to post several **hrecv()** requests simultaneously, with separate handlers or with a common handler. Because the handler is executed as soon as the receive is complete, you can ensure that a critical section of code is not interrupted by the execution of the handler by issuing the **masktrap()** call to protect that section of code. The example illustrates this situation.

If a message too long to fit in the buffer comes in, the receive is completed and the message is lost with no error returned. To detect this situation, check the count argument in the handler.

Errors

`hrecv: Buffer length exceeds allocation`

Make sure parameter length does not exceed buffer size.

`hrecv: Invalid length`

Use a non-negative number of a length less than or equal to maximum message length.

`hrecv: Not enough memory`

Reduce the size of the process.

`hrecv: Too many requests`

Use **msgwait()** or **msgdone()** for **isend()** or **irecv()** requests that are outstanding, or use a force type.

`hrecv: Invalid buffer pointer`

(C only) Specify a pointer which contains the address of a valid data buffer or correct the *proc* parameter.

See Also

crecv(), irecv(), handler(), hsend(), hsendrecv()

HSEND() HSEND()

Send a message and provide user-written exception handler on completion of the send.

Synopsis

SUBROUTINE **HSEND**(*type, buf, len, node, pid, proc*)

Parameter Declarations

INTEGER *type*
INTEGER *buf(*)*
INTEGER *len, node, pid*
EXTERNAL *proc*

Return Value

None

Example

This example uses **hsend()** to send message type 101 and execute an exception handler named "HANDLEPROC" as soon as the send is complete.

```
      CHARACTER BUF*1000
      EXTERNAL HANDLEPROC

      CALL HSEND(101, BUF, LEN(BUF), NODEID, 0, HANDLEPROC)
C HANDLEPROC WILL BE CALLED WHEN THE SEND IS COMPLETE
```

HSEND() *(cont.)*

HSEND() *(cont.)*

NOTE

The handler must be written in C and have four arguments. If you are calling the *proc* from a Fortran program, the name of the *proc* must be followed by an underline. The first line of the *proc* and the declaration of the parameters would have the following form:

```
proc_(type, count, node, pid)
long type, count, node, pid
```

hsend() passes the four values for the parameters: the message type (*type*), the length of the message (*count*), the destination node (*node*), and the destination process id (*pid*) to the handler for you.

The handler routine must not do standard input/output. Instead you must use routines that directly invoke UNIX system primitives. For example, you will get a segmentation violation if your C handler attempts to perform a **printf**. Use a **write** to standard output instead.

Environment

Node

Languages

C, Fortran

Description of Parameters

type	a user-defined integer that identifies the type of the message that you are sending.
buf	refers to the buffer that contains the message to be sent. The buffer may be of any legal data type.

HSEND() *(cont.)*

HSEND() *(cont.)*

len a positive integer that specifies the size (in bytes) of the message that you wish to send. Messages to or from a host process are limited to a maximum of 256K bytes. There is no limit on message length between nodes.

node an integer that defines the node(s) to which the message is to be sent

pid an integer that specifies the id of the process that is to receive the message. Valid *pids* include any integer value. Negative numbers are reserved for system programs. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on RX nodes). A node *pid* is assigned when you load a process and a host *pid* is assigned with **setpid()**.

proc the procedure executed when the send is complete. This procedure must be written by the user and must be a C procedure with four arguments.

Discussion

Use **hsend()** as an asynchronous send that sends a message to the specified node and process id and executes a handler procedure as soon as the send is complete. The name of the handler is the *proc* parameter for the call.

The handler must be written by the user in C, and must have four arguments: *type*, *count*, *node*, and *pid*. The **hsend()** call passes values to the arguments of the handler for you.

It is possible to post several **hsend()** requests simultaneously, with separate handlers or with a common handler. Because the handler is executed as soon as the send is complete, you can ensure that a critical section of code is not interrupted by the execution of the handler by issuing the **masktrap()** call to protect that section of code.

It is recommended that you use a *type* with a value from `0` to `1,000,000`. Types `1,073,741,824` to `1,999,999,999` are special force types that bypass the normal flow control mechanisms, do not match the `-1` wildcard, and are discarded if no receive is posted. Force types are intended to be used with **csendrecv()**, **isendrecv()**, and **hsendrecv()**, but can be used by other message passing calls.

NOTE

It is not recommended that you use types in other ranges because they are used by the system and could produce unpredictable results.

HSEND() *(cont.)*

HSEND() *(cont.)*

If *node* is a positive integer, the message is sent to the node with that number. Nodes within a partition are numbered consecutively from 0, and a node number higher than the highest node in the cube causes an error to be returned. To send a message to the host, use the host's id, which is returned by **myhost()**.

If *node* is a negative number up to a certain value, it causes a "global" send. If node is `-1`, the message is sent to all nodes. Other negative numbers cause the message to be sent to a subcube composed of a set of the nodes surrounding the sender. The dimension of this cube is calculated as (*d* - **nodedim**() `-1`), where *d* is the dimension of the subcube, and is less than or equal to **nodedim()**.

For RX nodes, if a global send specifies its own process (**mypid**()), it does not receive its own message. For CX nodes, however, if a process other than its own is specified, the sending node also receives the message.

Errors

`hsend: No pid defined`

Use **setpid**() to define a host process id.

`hsend: Invalid type`

Use a non-negative number.

`hsend: Invalid length`

Use a non-negative number or a length less than or equal to maximum message length.

`hsend: Invalid node`

Use **numnodes**() to determine cube size or **myhost**() to determine host number.

`hsend: Invalid buffer pointer`

Specify a pointer which contains the address of a valid data buffer.

`hsend: Not enough memory`

Make sure message buffer starts on a boundary that is a multiple of four or make more memory available to this process.

HSEND() *(cont.)*

HSEND() *(cont.)*

See Also

hrecv(), masktrap(), csend(), isend()

HSENDRECV() HSENDRECV()

Send a message and receive a reply; on receipt of the reply, executes a user-defined trap handler.

Synopsis

SUBROUTINE **HSENDRECV**(*type, sbuf, slen, tonode, topid, typesel, rbuf, rlen, proc*)

Parameter Declarations

INTEGER *type, typesel*
INTEGER *sbuf(*), rbuf(*)*
INTEGER *slen, tonode , topid, rlen*
EXTERNAL *proc*

Return Value

None

Example

This example uses **hsendrecv()** both to send a message and to post a receive for the result. On receipt, it will execute an exception handler named **hndl**. This examples consists of two parts. Part 1 runs on node 0, and part 2 runs on node 1. Node 0 sends a message to node 1 requesting action. When node 0 receives the answer, it also calls **hndl**.

Part 1: (runs on node 0)

```
      INCLUDE 'fcube.h'
      EXTERNAL HNDL
      DOUBLE PRECISION RSIN, PI, X , ANGLE
      INTEGER IAM,STIME,TIME

      IAM = MYNODE()
      PI  = ACOS(-1.0D00)
      ANGLE = PI/2.0D00
      X  = RSIN(ANGLE)
      WRITE(*,1000) CHAR(10),IAM,X,ANGLE
1000  FORMAT(1X,A1,'NODE = ', I2':  X =', D15.5,
     X' FOR ANGLE ',D15.5)
      END
```

HSENDRECV() *(cont.)*

HSENDRECV() *(cont.)*

```
      DOUBLE PRECISION FUNCTION RSIN(X)
      DOUBLE PRECISION X, RESULT
      INTEGER LEN
      CALL HSENDRECV(100, X,8,1,0,200,RESULT,8,HNDL)
      STIME = MCLOCK()
100   TIME = MCLOCK() - STIME
      IF (TIME .LE. 500) GOTO 100
         RSIN = RESULT
      END
```

Part 2: (runs on node 1)

```
      INCLUDE 'FCUBE.H'
      DOUBLE PRECISION X, RESULT

10    CONTINUE
      CALL CRECV(100,X,8)
      RESULT = SIN(X)
      CALL CSEND(200, RESULT, 8,0,0)
      GOTO 10
      END
```

The handler, which must be written in C, is as follows:

```
hndl_(type,count,node,pid)
long type,count,node,pid;
{
   write(1,"hello",5);
}
```

HSENDRECV() *(cont.)*

HSENDRECV() *(cont.)*

NOTE

The handler must be written in C and have four arguments. If you are calling the handler from a Fortran program, the name of the handler must be followed by an underline character. The first line of the handler and the declaration of the arguments would have the following form:

```
procname_ (type, count, node, pid)
long type, count, node, pid
```

The **hsendrecv()** routine passes the four values for the parameters to the handler so the **info...()** calls are not needed within the handler. The parameters are the message type (*type*), the length of the message (*count*), the node (*node*), and the process id (*pid*).

The handler routine must not do standard input/output. Instead you must use routines that directly invoke UNIX system primitives. For example, you will get a segmentation violation if your C handler attempts to perform a **printf**. Use a **write** to standard output instead, as shown in the example.

Environment

Node

Languages

C, Fortran

HSENDRECV() *(cont.)*

HSENDRECV() *(cont.)*

Description of Parameters

type identifies the type of the message that you are sending. Types are described in the Discussion.

sbuf refers to the buffer that contains the message to be sent.

slen is a positive integer that specifies the size (in bytes) of the message that you wish to send. Messages to or from a host process are limited to a maximum of 256K bytes. There is no limit on message length between nodes.

tonode an integer that determines the node(s) to which the message is to be sent.

topid for CX, is an integer that specifies the id of the process that is to receive the message. Valid *pids* include any integer value. Negative numbers are reserved for system programs. A node *pid* is assigned when you load a process and a host *pid* is assigned with **setpid()**. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on RX nodes).

typesel is an integer value that specifies the type(s) of message you are waiting to receive. This parameter is described further in the Discussion.*rbuf* refers to the buffer where the reply message will be stored.

rlen is an integer value that specifies the size of the receive buffer in bytes.

proc is the procedure executed when the receive is complete. This procedure must be written by the user and must be a C procedure with four arguments.

Discussion

Use **hsendrecv()** to send a message to a node or host process and simultaneously post a receive for the reply, and when receipt of the message is complete, immediately execute a receive trap handler procedure. The name of the handler is the *proc* parameter. The receive trap handler (the *proc* parameter) must be written by the user in C, and must have four arguments: *type, count, node,* and *pid.* **hsendrecv()** passes these values to the arguments of the handler for you.

It is possible to post several **hsendrecv()** requests simultaneously, with separate handlers or with a common handler. Because the handler is executed as soon as the receive is complete, you can ensure that a critical section of code is not interrupted by the execution of the handler by issuing the **masktrap()** call to protect that section of code.

If a message that is too long to fit in the buffer comes in, the receive is completed and the message is lost with no error returned. To detect this situation, check the count argument in the handler.

HSENDRECV() *(cont.)*

HSENDRECV() *(cont.)*

When a message whose type matches the type(s) specified in the *typesel* parameter, the calling process receives the message, and stores it in *rbuf*. The calling process continues executing instructions while the send-receive operation is occurring. You can use the **info...()** calls to get more information about the reply message.

The type parameter is a user-defined variable typically used to identify the kind of information contained in the message being sent. Types `0` to `1,000,000,000` are normal types that can be used by any send or receive call, including this one. Types `1,073,741,824` to `1,999,999,999` are special force types available to the user intended specifically for the "sendrecv" 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. If no receive is posted, as when the receiving process has been killed, a message with a force type is discarded. In general, bypassing the normal flow control mechanisms causes no problem because hsendrecv() guarantees that a receive is posted for the message.

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. The buffer for the send and receive may be any legal data type. It is recommended that data types match in send and receive operations. RX nodes can only send messages with *pid*s of 0. Since receive processes ignore *pid*, it is possible to receive a message with a *pid* of any value. However, RX nodes interpret *pid* values other than 0 as an error. To avoid this situation, you should set *pid* to 0 for all processes.

The *typesel* parameter describes the type(s) of the message(s) expected to be received. The *typesel* (type selector) allows you to select a specific message type or a set of message types based on a 32-bit mask.

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

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. You can use a force type, as described above.
- If *typesel* is `-1`, the first message to arrive for the process that initiated the receive operation will be recognized. After the first message has been received, you can use `-1` again to receive the next message.
- If *typesel* is any negative number other than `-1`, a set of message types will be recognized. Refer to Appendix B for information on building a *typesel* mask.

Errors

`hsendrecv: No pid defined`

Use **setpid()** to define a host process id.

`hsendrecv: Invalid type`

Use a non-negative number.

`hsendrecv: Invalid length`

Use a non-negative number or a length less than or equal to maximum message length.

`hsendrecv: Invalid node`

Use **numnodes()** to determine cube size or **myhost()** to determine hos* number.

`hsendrecv: Buffer length exceeds allocation`

Make sure parameter length does not exceed buffer size.

`hsendrecv: Invalid buffer pointer`

Specify a pointer which contains the address of a valid data buffer.

HSENDRECV() *(cont.)*

HSENDRECV() *(cont.)*

```
hsendrecv: Not enough memory
```

Make sure message buffer starts on a boundary that is a multiple of 4 or make more memory available to this process.

See Also

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

HWCLOCK() HWCLOCK()

Places the current value of the hardware counter in the 64-bit unsigned integer supplied as an argument.

Synopsis

SUBROUTINE **HWCLOCK**(*hwtime*)

Parameter Declarations

INTEGER *hwtime(2)*

Return Value

None

Example

The following example illustrates the proper use of **hwclock()**.

```
      INTEGER*4 STIME(2), ETIME(2), TTIME(2)
          .
          .
C FIND STARTING TIME
      CALL HWCLOCK(STIME)
C EXECUTE PROCESS
          .
          .
C FIND ENDING TIME
      CALL HWCLOCK(ETIME)
C FIND ELAPSED TIME
      CALL ESUB(ETIME, STIME, TTIME)
```

HWCLOCK() *(cont.)*

HWCLOCK() *(cont.)*

Environment

Node

Languages

C, Fortran

Description of Parameters

hwtime an output variable that receives the current value of the hardware counter in a 64-bit unsigned integer.

Discussion

Places the current value of the hardware counter in the 64-bit unsigned integer supplied as an argument. If an array is supplied, the subroutine places the least-significant 32 bits in the first word. The count rate of the hardware is defined in the parameter *hwhz* in */usr/include/fcube.h*.

Errors

None

See Also

dclock(), mclock()

INFO: INFOCOUNT(), INFONODE(), INFOPID(), INFOTYPE()

Return information about a pending or received message.

Synopsis

INTEGER FUNCTION **INFOCOUNT()**

INTEGER FUNCTION **INFONODE()**

INTEGER FUNCTION **INFOPID()**

INTEGER FUNCTION **INFOTYPE()**

Parameter Declarations

None

Return Value

The indicated information is returned. The return value is undefined unless the call is made immediately after a **crecv()**, **cprobe()**, or **msgwait()** is complete, or after an **iprobe()** or **msgdone()** return a `1`, all of which indicate that a message is ready.

The return values for each of the calls are:

infocount returns an integer value specifying the length of the message received (in bytes) after a probe or receive operation completes.

infonode returns the node id of the process which sent the message.

infopid returns the *pid* of the process which sent the message.

infotype returns the type of the message. Message type is specified in the send operation.

INFO: *(cont.)*

INFO: *(cont.)*

Example

The following example uses all four of the **info...()** calls. First, it waits for a message of any type and then uses **infotype()** to determine the type. Then, it uses that information to receive the waiting message. After the receive is complete, it uses **infocount()** to return the message length. Next, the example posts an **irecv()** for a message of type `10`, and if it has been received, uses **infonode()** and **infopid()** to find out the node and process id from which the message came.

```
      PARAMETER(BIGNUM=65536)
      INTEGER BUF(BIGNUM)
      INTEGER MSGTYPE, MSGLEN, MSGID
      INTEGER FROMNODE, FROMPID
         .
         .
C  WAIT FOR ANY MESSAGE TO ARRIVE, AND GET ITS TYPE

      CALL CPROBE(-1)
      MSGTYPE = INFOTYPE()

C  RECEIVE THE WAITING MESSAGE AND GET ITS LENGTH IN BYTES.

      CALL CRECV(MSGTYPE, BUF, 4*BIGNUM)
      MSGLEN = INFOCOUNT()
         .
         .
C  POST A RECEIVE FOR A TYPE 10 MESSAGE AND FIND ITS SOURCE
      MSGID = IRECV(10, BUF, 4*BIGNUM)
         .
         .
      IF (MSGDONE(MSGID) .EQ. 1) THEN
        FROMNODE = INFONODE()
        FROMPID = INFOPID()
      END IF
```

INFO: *(cont.)*

INFO: *(cont.)*

Discussion

Use **infocount()**, **infonode()**, **infopid()**, and **infotype()** to return information about a pending or received message. They return information about the message detected when **cprobe()** completes or when **iprobe()** returns `1`. They return information about the message received when **crecv()** or **msgwait()** return or when **msgdone()** for a previous **irecv()** returns a `1`.

They must immediately follow the related message-passing statement.

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

None

INFO: *(cont.)*

INFO: *(cont.)*

Errors

`infocount: No pid defined`

Use **setpid()** to define a host process id.

`infonode: No pid defined`

Use **setpid()** to define a host process id.

`infopid: No pid defined`

Use **setpid()** to define a host process id.

`infotype: No pid defined`

Use **setpid()** to define a host process id.

See Also

crecv(), cprobe(), irecv(), iprobe(), msgdone(), msgwait()

IODONE() IODONE()

Determines whether an asynchronous I/O read or write operation is complete.

Synopsis

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

Parameter Declarations

INTEGER *id*

Return Values

0	indicates that the read or write is not yet complete.
1	indicates that the read or write is complete and that the associated buffer is available for reuse, in the case of a write operation, or that the buffer contains valid data, in the case of a read operation.

Example

This example uses **iodone()** to determine whether **iwrite()** has finished the write. If so, the buffer can be used; otherwise the buffer must be protected until the write is complete.

```
      DOUBLE PRECISION BUF(1000)
      INTEGER ID, UNIT
           .
           .
      ID = IWRITE(UNIT, BUF, 1000*8)
           .
           .
      IF (IODONE(ID) .EQ. 1) THEN
C YOU MAY NOW RE-USE THE BUFFER.
      ELSE
C THE BUFFER STILL NEEDS TO BE PROTECTED.
      END IF
```

IODONE() *(cont.)*

IODONE() *(cont.)*

Environment

Node

Languages

C, Fortran

Description of Parameters

id the value returned by an **iread()** or **iwrite()** operation.

Discussion

Use **iodone()** to determine whether an **iread()** or **iwrite()** operation is complete, and when **iodone()** returns a `1`, it releases the *id* number that identifies the asynchronous read or write on which you are checking the status. The *id* is the return value of the **iread()** or **iwrite()** you are checking on. You must use either **iodone()** or **iowait()** after an asynchronous message passing call to ensure that the operation is complete and to release the id number.

Errors

```
iodone: Invalid message id
```

Use *id* returned by **iread()** or **iwrite()**.

See Also

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

IOMODE() IOMODE()

Return the current I/O mode for a file.

Synopsis

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

Parameter Declarations

INTEGER *unit;*

Return Value

`0, 1, 2,` or `3`, depending on the current I/O mode for the designated file. See **setiomode()** for information on I/O modes.

Example

The following example uses **iomode()** to return the current I/O mode of the file designated by the *unit* parameter `3`, assumed to be the value assigned to the *unit* parameter of the Fortran **open()** statement when you opened the file:

```
      .
      .
IMODE = IOMODE(3)
      .
      .
```

Environment

Node

Languages

C, Fortran

IOMODE() *(cont.)*

IOMODE() *(cont.)*

Description of Parameters

unit is a positive integer value that was assigned to the *unit* parameter of the **open()** statement when the file was opened.

Discussion

Use **iomode()** to determine the current I/O mode of the file specified by *unit*. This is an integer from `0` through `3`. See **setiomode()** for information on I/O modes. The *unit* parameter must contain the value assigned to the *unit* parameter of the **open()** statement when the file was opened.

Errors

`iomode: Bad file number`

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

`iomode: Not implemented on the SRM`

Can be executed by the node only; not available on the host.

`iomode: No such unit`

Unit must be a value no larger than 100.

`iomode: Fortran runtime error: Unit not open`

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

See Also

setiomode()

IOWAIT() IOWAIT()

Wait for completion of an asynchronous I/O read or write operation.

Synopsis

SUBROUTINE **IOWAIT**(*id*)

Parameter Declarations

INTEGER *id*

Return Value

None

Example

This example uses **iowait()** to stop further execution of the process until an **iread()** posted earlier has been completed before continuing because the code at that point requires that data.

```
      INTEGER RESULT
      INTEGER ID,UNIT
          .
          .

      ID = IREAD(UNIT, RESULT, 4)

C     PERFORM TASKS THAT DO NOT USE 'RESULT'.
          .
          .
C     STOP UNTIL THE MESSAGE IS RECEIVED.

      CALL IOWAIT(ID)

C     NOW YOU CAN USE 'RESULT'.
```

IOWAIT() *(cont.)*

IOWAIT() *(cont.)*

Environment

Node

Languages

C, Fortran

Description of Parameters

id is the message id returned when you invoke an **iread()** or **iwrite()** operation.

Discussion

The **iowait()** call allows a node process to wait until an **iread()** or **iwrite()** operation is complete, and when the operation is complete, it releases the id number used by the asynchronous I/O operation to which it refers. Use either **iodone()** or **iowait()** after an asynchronous message passing call to ensure that the operation is complete and that the id is freed. The *id* parameter identifies the operation that you are waiting for, and must correspond to the value returned by a previous **iread()** or **iwrite()** call.

When the I/O operation is complete, **iowait()** returns and the associated message buffer is available for reuse in the case of a write operation, or when the buffer contains valid data in the case of a read operation. When **iowait()** returns, it clears the id number used by the read or write for use by a future read or write.

Errors

```
iowait: Invalid message id
```

Use message id returned by **iread()** or **iwrite()**.

See Also

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

IPROBE() IPROBE()

Determine whether a message of a selected type is pending.

Synopsis

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

Parameter Declarations

INTEGER *typesel*

Return Values

0	a message of the selected type is not waiting to be received
1	a message of the selected type is waiting to be received

Example

This example shows a continuous for loop that uses **iprobe()** to test for the availability of two messages (types 1 and 2) and processes the first message to come in. If neither message is available, **flick()** is used to allow another process on the same node to execute before looping again.

```
C     LOOP CONTINUOUSLY UNTIL A TYPE 1 OR TYPE 2 MESSAGE
C     ARRIVES, THEN PROCESS ACCORDINGLY.
10     IF (IPROBE(1) .EQ. 1) THEN
C     PROCESS TYPE 1 MESSAGE.
            .
            .
          GOTO 20
       ELSE IF (IPROBE(2) .EQ. 1) THEN
C     PROCESS TYPE 2 MESSAGE.
            .
            .
         GOTO 20
       ELSE
         CALL FLICK()
       END IF
       GOTO 10
20     CONTINUE
```

IPROBE() *(cont.)*

IPROBE() *(cont.)*

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

typesel is an integer value that specifies the type(s) of message that you are waiting for. 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 non-force-type message to arrive for the process that initiated the probe operation will be recognized. After the first message has been received, you can use `-1` again to probe for the next message.
- If *typesel* is any negative number other than `-1`, a set of message types will be recognized. Refer to Appendix B for instructions on how to build a *typesel* mask.

Discussion

Use **iprobe()** to determine whether a message of a selected type is available for receipt. You specify the message type with the send operation, and **iprobe()** looks for a message whose *type* matches the type(s) specified in the calling process. If a message of the specified type is waiting in the node to be received, **iprobe()** returns a `1`. If there are no such messages, a `0` is returned. When **iprobe()** returns a `1`, the **info...()** calls can be used to get more information about the message.

This is an asynchronous call and therefore does not block the process waiting for a message to arrive. Use **cprobe()** to block the process until a message of a selected type is available for receipt.

IPROBE() *(cont.)*

IPROBE() *(cont.)*

Errors

`iprobe: No pid defined`

Use **setpid()** to define a host process id.

`iprobe: Too many requests`

Use **msgwait()** for outstanding **isend()** or **irecv()** requests.

See Also

cprobe(), infocount(), infonode(), infopid(), infotype()

IREAD() IREAD()

High-speed, asynchronous read from a file.

Synopsis

INTEGER FUNCTION **IREAD**(*unit, buf, len*)

Parameter Declarations

INTEGER *unit*
INTEGER *buf(*)*
INTEGER *len*

Return Value

Returns a non-negative id to be used in **iodone()** or **iowait()**.

Example

The following example uses **iread()** to read the contents of a file while the process continues to execute. The value for the *unit* parameter is the value you assigned the *unit* parameter in **open()**. Then, when the data is required by the process, **iowait()** is used to block the process until the read is complete and to clear the id for use by future reads or writes:

```
      INTEGER RESULT
      INTEGER ID,UNIT1
      UNIT1 = 3
      OPEN(UNIT=UNIT1, FILE='/CFS/MYDATA', FORM='UNFORMATTED')
         .
         .
      ID = IREAD(UNIT1, RESULT, 4)

C PERFORM TASKS THAT DO NOT USE 'RESULT'.
         .
         .
C STOP UNTIL THE MESSAGE IS RECEIVED.
      CALL IOWAIT(ID)
C NOW YOU CAN USE 'RESULT'
```

IREAD() *(cont.)*

IREAD() *(cont.)*

Environment

Node

Languages

C, Fortran

Description of Parameters

unit the value you assign the file with the *unit* parameter of **open()**.

buf refers to the buffer where the read data will be stored. The buffer can be any valid data type.

len is an integer value that specifies the size of the buffer in bytes. Transfer sizes are limited only by memory available for the buffer.

Discussion

Use **iread()** to perform high-speed asynchronous data reads from the file identified with the *unit* parameter which you assigned to the file with the *unit* parameter of **open()**. To use **cread()**, you must use **open()** and set the FORM parameter to `'UNFORMATTED'`.

For a given file, you can use either the iPSC system read and writes (**cread()**, **iread()**, **cwrite()**, **iwrite()**), or the Fortran **read()** and **write()**. An attempt to access a file with a mixture of the two kinds causes an error. Files automatically created with **read()** or **write()** without using **open()** will not have the correct format for the high-speed iPSC system reads and writes, but can be accessed with **read()** and **write()**. (These files will be assigned a name of the form `fnnn.unit`, where `nnn` is the node number and `unit` is the value of the unit parameter).

When **iread()** is executed, it is asynchronous in that it allows the process to continue executing instructions while the read is being done. When you need to use the read data in processing, use **iowait()** to block further processing until the read is complete, and to free the id number for use by a future read or write (this is required because the number of IDs available is limited, and using them up causes an error). You can also use **iodone()** which returns a `1` and clears the id if the read is complete.

To determine whether the read operation moved the file pointer to the end-of-file, use **iseof()**. The **iread()** routine modifies the file pointer immediately, so you can execute **lseek()**, **iseof()**, or another read or write routine immediately without waiting for the read to finish.

IREAD() *(cont.)*

IREAD() *(cont.)*

Errors

`iread: Invalid argument`

Check arguments.

`iread: Mixed file operations`

In I/O mode 2 or 3, nodes are attempting different operations (reads and writes) to a shared file. Ensure that nodes are performing the same operation.

`iread: I/O error`

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

`iread: No such unit`

Unit must be a value no larger than 100.

`iread: Too many open files`

Only 100 files can be open at one time for any process.

`iread: Not enough core`

Not enough memory on node for operation.

`iread: Unformatted I/O to FORMATTED file`

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

`iread: Attempt to mix standard and CFS I/O call`

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

`iread: Too many requests`

Use **iowait()** for **iread()** or **iwrite()** requests that are outstanding.

IREAD() *(cont.)*

IREAD() *(cont.)*

See Also

cread(), cwrite(), infocount(), iseof(), iwrite(), iodone(), iowait()
iPSC®/860 Fortran Language Reference Manual: **OPEN(), READ()**

IRECV() IRECV()

Asynchronously receive a message.

Synopsis

INTEGER FUNCTION **IRECV**(*typesel, buf, len*)

Parameter Declarations

INTEGER *typesel*
INTEGER *buf(*)*
INTEGER *len*

Return Value

Returns a non-negative message id which is to be used in **msgcancel()**, **msgdone()**, or **msgwait()**.

Example

The following example uses **irecv()** to initiate receipt of a message, and when the information in the message is required by the process, uses **msgwait()** both to block further execution of the process until the receive is complete, and to make the id used for the receive available for subsequent asynchronous message passing functions.

```
      INTEGER BUF(50)
      INTEGER MSGID, TYPE
          .
          .
C  INITIATE RECEIPT OF A MESSAGE.

      MSGID = IRECV(TYPE,BUF,50*4)

C  PERFORM TASKS NOT DEPENDENT ON MESSAGE RECEIPT
          .
          .
C  NOW WAIT TO GUARANTEE THE RECEIPT OF A MESSAGE
      CALL MSGWAIT(MSGID)

C  NOW USE CONTENTS OF 'BUF' AND/OR INFO ROUTINES
```

IRECV() *(cont.)*

IRECV() *(cont.)*

Environment

Host/Node

Languages

C, Fortran

Description of Parameters

typesel is an integer that specifies the type(s) of message that you are receiving. 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 non-force-type message to arrive for the process that initiated the receive operation will be recognized. After the first message has been received, you can use `-1` again to receive the next message.
- If *typesel* is a negative number other than `-1`, a set of message types will be recognized. Refer to Appendix B for instructions on how to build a *typesel* mask.

buf refers to the buffer where the received message will be stored. Valid buffer entries are any legal data type. Data types should match in send and receive operations. The *buffer* is not available for use until **msgwait()** returns or **msgdone()** indicates that the receive has completed.

len is an integer that specifies the size of the message buffer (in bytes). The maximum number of bytes that you can send in a single message to or from a host process is 256K bytes. There is no limit on message length between nodes.

IRECV() *(cont.)*

IRECV() *(cont.)*

Discussion

Use **irecv()** to post an asynchronous receive that initiates the receipt of a message from a process. It searches for a message whose type matches the type(s) specified in the *typesel* parameter, and immediately returns a message id which is used by **msgdone()** and **msgwait()** to determine whether the receive is complete. **msgwait()** also clears the id assigned to the message as soon as the receive completes, as does **msgdone()** if it returns a `1`. You must use either **msgwait()**, **msgdone()**, or **msgcancel()** to free this id up for future sends and receives because the number of IDs available is limited, and an error results if you use up all the IDs. When the message is received it is stored in a buffer (the *buf* parameter).

After **msgwait()** completes or **msgdone()** returns `1`, you can use the **info...()** calls to get more information about the message. **irecv()** does not affect the values returned by the **info...()** calls; it only initiates the receipt of a message.

irecv() is asynchronous, allowing the calling process to continue even if the desired message is not yet available. It returns as soon as the receive request has been initiated. Use **crecv()** when you want the calling process to be blocked until the desired message is received.

Errors

`irecv: No pid defined`

Use **setpid()** to define a host process id.

`irecv: Invalid length`

Use a non-negative number or a length less than or equal to maximum message length.

`irecv: Too many requests`

Use **msgwait()** for **isend()** or **irecv()** requests that are outstanding.

`irecv: Buffer length exceeds allocation`

Make sure parameter length does not exceed buffer size.

`irecv: Invalid buffer pointer`

Specify a pointer which contains the address of a valid data buffer.

IRECV() *(cont.)*

IRECV() *(cont.)*

See Also

crecv(), infocount(), infonode(), infopid(), infotype(), isend(), isendrecv(), msgcancel(), msgdone(), msgwait()

ISEND() ISEND()

Asynchronously send a message.

Synopsis

INTEGER FUNCTION **ISEND**(*type, buf, len, node, pid*)

Parameter Declarations

INTEGER *type*
INTEGER *buf(*)*
INTEGER *len, node, pid*

Return Value

Returns a non-negative message id which is used by **msgcancel()**, **msgdone()**, or **msgwait()**.

Example

The following example uses **isend()** to send a message of type 1 to process 10 on node 16, and when the buffer is required, uses **msgwait()** both to block further execution of the process until the send is complete, and to make the id used for the send available for subsequent reads and writes.

```
      CHARACTER MSG*132
      INTEGER MSGID, MSGLEN, NODE, PID, TYPE
          .
          .
      MSGLEN = LEN(MSG)
      NODE = 16
      PID = 10
      TYPE = 1
C SEND A TYPE 1 MESSAGE TO PROCESS 10 ON NODE 16
      MSGID = ISEND(TYPE, MSG, MSGLEN, NODE, PID)
          .
          .
C PERFORM TASKS THAT DO NOT CHANGE CONTENTS OF MSG
          .
          .
      CALL MSGWAIT(MSGID)
C NOW YOU CAN RE-USE THE MESSAGE BUFFER
```

ISEND() *(cont.)*

ISEND() *(cont.)*

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

type is an integer that identifies the message that you want to send. Type restrictions are described in the Discussion.

buf refers to the buffer that contains the message that you want to send. Buffer entries can be any legal data type. Data types should match in send and receive operations.

len is an integer that specifies the size of the message (in bytes) that you want to send. The maximum number of bytes that you can send in a single message to or from a host process is 256K. There is no limit on message length between nodes. If *len* is greater than the maximum message length, an error message will be issued.

node is an integer value that identifies the node or nodes to which the message is to be sent.

pid for CX nodes, is an integer value that specifies the process id which is to receive the message. Valid *pids* include any integer value. Negative numbers are reserved for system programs. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on RX nodes). A node *pid* is assigned when you load a process and a host *pid* is assigned with **setpid()**.

ISEND() *(cont.)*

ISEND() *(cont.)*

Discussion

Use **isend()** to send a message to a node or host process. Completion of **isend()** does not imply that the message was received by the destination process, only that the send operation was initiated. It returns a message id which is used by **msgdone()** and **msgwait()** to determine whether the send is complete. Use **msgwait()** to clear the id assigned to the message as soon as the receive completes, as does **msgdone()** if it returns a `1`. When the message is sent, the buffer can be used for another message if desired.

This is asynchronous, returning as soon as the send request is initiated. Use **csend()** when you want to block the calling process until the message is sent.

It is recommended that you use a *type* with a value from `0` to `999,999,999`. Types `1,073,741,824` to `1,999,999,999` are special force types that bypass the normal flow control mechanisms, do not match the `-1` wildcard, and are discarded if no receive is posted. Force types are intended to be used with **csendrecv()**, **isendrecv()**, and **hsendrecv()**, but can be used by other message passing calls.

NOTE

It is not recommended that you use *types* in other ranges because they are used by the system and could produce unpredictable results.

The buffer can be of any *type*. You should not modify *buf* until **msgwait()** returns or **msgdone()** indicates that **isend()** has completed. Otherwise, corrupted data might be sent.

If *node* is a positive integer, the message is sent to the node with that number. Nodes within a partition are numbered consecutively from 0, and a node number higher than the highest node in the cube causes an error to be returned. To send a message to the host, use the host's id, which is returned by **myhost()**.

If node is a negative number up to a certain value, it causes "global" send. If node is `-1`, the message is sent to all nodes. Other negative numbers cause the message to be sent to a subcube composed of a set of the nodes surrounding the sender. The dimension of this cube is calculated as (*d* - **nodedim()** `-1`) , where *d* is the dimension of the subcube, and is less than or equal to **nodedim()**.

For RX nodes, if a global send specifies its own process (**mypid()**), it does not receive its own message. For CX nodes, however, if a process other than its own is specified, the sending node also receives the message. RX nodes can only send messages with *pid*s of `0`. Since receive processes ignore *pid*, it is possible to receive a message with a *pid* of any value. However, RX nodes interpret *pid* values other than `0` as an error. To avoid this situation, you should set *pid* to 0 for all processes.

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

Errors

`isend: No pid defined`

Use **setpid()** to define a host process id.

`isend: Invalid type`

Use a non-negative number.

`isend: Invalid length`

Use a non-negative number or a length less than or equal to maximum message length.

`isend: Invalid node`

Use **numnodes()** to determine cube size and **myhost()** to determine host number.

`isend: Invalid buffer pointer`

(C only) Specify a pointer which contains the address of a valid data buffer.

`isend: Not enough memory`

Make sure message buffer starts on a boundary which is a multiple of 4 or make more memory available to this process.

`isend: Too many requests`

Use **msgwait()** or **msgdone()** for **isend()** or **irecv()** requests that are outstanding.

See Also

crecv(), csend(), irecv(), isendrecv(), msgcancel(), msgdone(), msgwait()

ISENDRECV() ISENDRECV()

Send a message and receive a reply.

Synopsis

INTEGER FUNCTION **ISENDRECV**(*type, sbuf, slen, tonode, topid, typesel, rbuf, rlen*)

Parameter Declarations

INTEGER *type, typesel*
INTEGER *sbuf(*), rbuf(*)*
INTEGER *slen, tonode, topid, rlen*

Return Value

Returns a non-negative message id which is used by **msgcancel()**, **msgdone()**, or **msgwait()**.

Example

The following example illustrates how to use **isendrecv()**. Part 1, which runs on node 0, is a simple procedure that uses **isendrecv()** to make a remote procedure call and post a receive for the message containing the result. When result is required, **msgwait()** is used to block the process until result is received. Part 2, which runs on node 2, is a portion of the server running on the receiving node that receives the message, performs the operation, and sends back the result to the first procedure.

ISENDRECV() *(cont.)*

ISENDRECV() *(cont.)*

Part 1: (runs on node 0)

```
      INCLUDE 'fcube.h'

      DOUBLE PRECISION RSIN, PI, X , ANGLE
      INTEGER IAM

      IAM = MYNODE()
      PI  = ACOS(-1.0D00)
      ANGLE = PI/2.0D00
      X  = RSIN(ANGLE)
      WRITE(*,1000) CHAR(10),IAM,X,ANGLE
1000  FORMAT(1X,A1,'NODE = ', I2':  X =', D15.5,
     x' FOR ANGLE ',D15.5)
      END

      DOUBLE PRECISION FUNCTION RSIN(X)
      DOUBLE PRECISION X, RESULT
      INTEGER ID

      ID = ISENDRECV(100, X,8,1,0,200,RESULT,8)
      CALL MSGWAIT(ID)
      RSIN = RESULT
      END
```

Part 2: (runs on node 1)

```
      INCLUDE 'fcube.h'

      DOUBLE PRECISION X, RESULT

10    CONTINUE
      CALL CRECV(100,X,8)
      RESULT = SIN(X)
      CALL CSEND(200, RESULT, 8,0,0)
      GOTO 10
      END
```

ISENDRECV() *(cont.)*

ISENDRECV() *(cont.)*

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

type identifies the type of the message that you are sending.

sbuf refers to the buffer that contains the message to be sent.

slen is a positive integer that specifies the size (in bytes) of the message that you wish to send. Messages to or from a host process are limited to a maximum of 256K bytes. There is no limit on message length between nodes.

tonode an integer that defines the node(s) to which the message is to be sent.

topid for CX, is an integer that specifies the id of the process that is to receive the message. Valid *pids* include any integer value. Negative numbers are reserved for system programs. A node *pid* is assigned when you load a process and a host *pid* is assigned with **setpid()**. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on RX nodes).

typesel is an integer value that specifies the type(s) of the reply message. See the Discussion for more information.

rbuf refers to the buffer where the reply will be stored.

rlen is an integer value that specifies the size of the reply buffer in bytes.

ISENDRECV() *(cont.)*

ISENDRECV() *(cont.)*

Discussion

Use **isendrecv**() to send a message to a node or host process, and simultaneously post a receive for the reply. When a message arrives whose *type* matches the type(s) specified in the *typesel* parameter, the calling process receives the message, and stores it in *rbuf*. The calling process continues executing instructions while the send-receive operation is occurring.

When it is executed, **isendrecv**() returns a message id. Use **msgdone**() or **msgwait**() to free these message id numbers and to determine whether the **isendrecv**() operation has completed. You cannot use the **info...**() calls to get information about the reply message, because **isendrecv**() does not disturb the information.

The *type* parameter is a user-defined variable typically used to identify the kind of information contained in the message being sent. Types `0` to `999,999,999` are normal types that can be used by any send or receive call, including this one. Types `1,073,741,824` to `1,999,999,999` are special force types available to the user intended specifically for the "sendrecv" 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. If no receive is posted, as when the receiving process has been killed, a message with a force type is discarded. In general, bypassing the normal flow control mechanisms causes no problem because **isendrecv**() guarantees that a receive is posted for the message.

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.

The buffer for the send and receive may be any legal data *type*. It is recommended that data types match in send and receive operations.

If *tonode* is a positive integer, the message is sent to the node with that number. Nodes within a partition are numbered consecutively from 0, and a node number higher than the highest node in the cube causes an error to be returned. To send a message to the host, use the host's id, which is returned by **myhost**().

ISENDRECV() *(cont.)*

ISENDRECV() *(cont.)*

If *tonode* is a negative number up to a certain value, it causes a "global" send. If node is `-1`, the message is sent to all nodes. Other negative numbers cause the message to be sent to a subcube composed of a set of the nodes surrounding the sender. The dimension of this cube is calculated as (*d* - **nodedim**() `-1`), where *d* is the dimension of the subcube, and is less than or equal to **nodedim**().

For RX nodes, if a global send specifies its own process (**mypid**())it does not receive its own message. For CX nodes, however, if a process other than its own is specified, the sending node also receives the message. RX nodes can only send messages with *pid*s of 0. Since receive processes ignore *pid*, it is possible to receive a message with a *pid* of any value. However, RX nodes interpret *pid* values other than 0 as an error. To avoid this situation, you should set *pid* to 0 for all processes.

The *typesel* parameter describes the type(s) of the message(s) expected to be received. 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. You can use a force type, as described above.
- If *typesel* is `-1`, the first message to arrive for the process that initiated the receive operation will be recognized. After the first message has been received, you can use `-1` again to receive the next message.
- If *typesel* is any negative number other than `-1`, a set of message types will be recognized. Refer to Appendix B for information on building a *typesel* mask.

Errors

`isendrecv: No pid defined`

Use **setpid**() to define a host process id.

`isendrecv: Invalid type`

Use a non-negative number.

`isendrecv: Invalid length`

Use a non-negative number or a length less than or equal to maximum message length.

ISENDRECV() *(cont.)*

ISENDRECV() *(cont.)*

`isendrecv: Invalid node`

Use **numnodes()** to determine cube size or **myhost()** to determine host number.

`isendrecv: Buffer length exceeds allocation`

Make sure parameter length does not exceed buffer size.

`isendrecv: Invalid buffer pointer`

Specify a pointer which contains the address of a valid data buffer (C only).

`isendrecv: Not enough memory`

Make sure message buffer starts on a boundary that is a multiple of four or make more memory available to this process.

`isendrecv: Received message too long for buffer`

Make sure buffer is large enough to hold the message.

See Also

iprobe(), irecv(), isend(), csendrecv(), msgcancel(), msgdone(), msgwait()

ISEOF() ISEOF()

Tests CFS reads for end-of-file

Synopsis

INTEGER FUNCTION **ISEOF**(*unit*)

Parameter Declarations

INTEGER *unit*

Return Value

`1` if the file pointer is at the end of the file, and `0` if not.

Example

The following example uses **iseof()** to make sure that all the data in a file is processed. Data is read and processed until the file pointer is at the end of the file. The variable *UNIT1* contains the value you assign the *unit* parameter of the Fortran **open()** routine

```
            .
            .
10     CONTINUE
C     READ CONTENTS OF THE FILE TO THE BUFFER
       CALL CREAD(UNIT1, BUFFER, BUFLEN)
C     PROCESS DATA
            .
            .
            .
C     CHECK FOR EOF; IF NOT, READ MORE DATA FROM FILE
       IF (ISEOF(UNIT1) .EQ. 0) GOTO 10
C     IF AT END-OF-FILE, GO ON
```

Environment

Node

ISEOF() *(cont.)*

ISEOF() *(cont.)*

Languages

C, Fortran

Description of Parameters

unit is a positive integer assigned with the *unit* parameter of **open()**.

Discussion

Use **iseof()** to determine whether you have reached the end of the file you are reading. It returns a `1` if the file pointer for a file identified by the *unit* parameter is at the end of the file. If not, the call returns a `0`. The unit parameter must be the value you assigned to the *unit* parameter of **open()** when you opened the file. You can use **iseof()** with all reads and writes but is most useful with reads.

Errors

`iseof: Bad file number`

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

`iseof: Not implemented on the SRM`

Applies to CFS files only; cannot be applied to host files.

`iseof: Attempt to mix standard and CFS I/O calls`

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

`iseof: No such unit`

Unit must be a value no larger than 100.

`iseof: Fortran runtime error: Unit not open`

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

ISEOF() *(cont.)*

ISEOF() *(cont.)*

See Also

cread(), iread()
iPSC®/860 Fortran Language Reference Manual: **OPEN()**

IWRITE() IWRITE()

High-speed asynchronous write to a file.

Synopsis

INTEGER FUNCTION **IWRITE**(*unit, buf, len*)

Parameter Declarations

INTEGER *unit*
INTEGER *buf(*)*
INTEGER *len*

Return Value

Returns a non-negative id to be used in **iodone()** or **iowait()**.

Example

This example uses **iwrite()** to write the contents of *BUF* to the file designated by *UNIT1*. This is the value you assigned the *unit* parameter of **open()**. Then, when the buffer is required, **iowait()** blocks the procedure until the write is complete and clears the id for use by future reads or writes.

```
      CHARACTER*100 BUF
      INTEGER ID, UNIT1
      UNIT1 = 3
      OPEN(UNIT=UNIT1,FILE='/cfs/mydata',FORM='UNFORMATTED')
          .
          .
      ID = IWRITE(UNIT1, BUF, LEN(BUF))
C      DO PROCESSING THAT DOES NOT REQUIRE THE BUFFER
          .
          .
      IOWAIT(ID)
C      YOU MAY NOW RE-USE THE BUFFER.
```

IWRITE() *(cont.)*

IWRITE() *(cont.)*

Environment

Node

Languages

C, Fortran

Description of Parameters

unit is the unit you assign the file with **open()**.

buf refers to the buffer containing the data to be written. The buffer can be any valid data type.

len is an integer value that specifies the size of the buffer in bytes. Transfer size is limited only by the memory available for the buffer.

Discussion

For a given file, you can use either the iPSC system reads and writes (**cread()**, **iread()**, **cwrite()**, **iwrite()**), or the Fortran **read()** and **write()**. An attempt to access a file with a mixture of the two kinds causes an error. Files automatically created with **read()** or **write()** without using **open()** will not have the correct format for the high-speed iPSC system reads and writes, but can be accessed with **read()** and **write()** (These files will be assigned a name of the form `fnnn.unit`, where `nnn` is the node number and *unit* is the value of the unit parameter).

With **iwrite()** the process continues executing instructions while the write is being done. When you need to use the write buffer in processing, use the **iowait()** call to block further processing until the write is complete, and to free the id number for use by a future read or write (this is required because the number of IDs available is limited, and using them up causes an error). The **iodone()** call will also clear the id when it returns a `1`.

iwrite() modifies the file pointer immediately, so you can call **lseek()** or another read or write call right away without waiting for the write to complete.

IWRITE() *(cont.)*

IWRITE() *(cont.)*

Errors

`iwrite: Invalid argument`

Check arguments.

`iwrite: Mixed file operations`

In I/O mode 2 or 3, nodes are attempting different operations (reads and writes) to a shared file. Ensure that nodes are performing the same operation.

`iwrite: I/O error`

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

`iwrite: Not enough core`

Not enough memory on node for operation.

`iwrite: No space left on device`

Not enough space on device to which you are writing. Create more space in file system, or if you have restricted the disks to which you are writing with **restrictvol()**, re-execute **restrictvol()** to include more drives.

`iwrite: Too many requests`

Use **iowait()** for **iread()** or **iwrite()** requests that are outstanding.

See Also

cread(), cwrite(), iread(), iodone(), iowait(), setiomode()
iPSC®/860 Fortran Language Reference Manual: **OPEN()**

KILLCUBE() KILLCUBE()

Terminate and clear out process(es).

Synopsis

SUBROUTINE **KILLCUBE**(*node, pid*)

Parameter Declarations

INTEGER *node, pid*

Return Value

None

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

node is an integer that identifies a particular node on which you want to terminate processes and flush messages or `-1` to indicate all nodes. Nodes in a partition are numbered consecutively from 0. A host id is invalid.

pid for CX, is an integer value that specifies the process id that you want to terminate and flush. *Pid* must match a *pid* given for a previous load, or it may be `-1` to terminate and flush all processes and messages related to those processes. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on RX nodes).

KILLCUBE() *(cont.)* KILLCUBE() *(cont.)*

Discussion

Use **killcube()** to kill the specified node process(es) and flush messages related to those processes. It is included for convenience and is equivalent to the sequence:

```
CALL KILLCUBE(NODE, PID)
CALL WAITALL(NODE, PID)
CALL FLUSHMSG(-1, NODE, PID)
```

It is not an error to use **killcube()** to kill a nonexistent process.

Errors

`killcube: Invalid node`

Use **numnodes()** to determine cube size and **myhost()** to determine host number.

`killproc: Invalid pid`

Use a non-negative number. For RX nodes, *pid* must be 0.

See Also

load(), **killproc()**, **flushmsg()**, **waitall()**

KILLPROC() KILLPROC()

Terminate a process.

Synopsis

SUBROUTINE **KILLPROC**(*node, pid*)

Parameter Declarations

INTEGER *node, pid*

Return Value

None

Example

The following statements are examples of how to use the **killproc()** call on CX nodes.

```
C       KILL PROCESS 3 ON NODE 15.
C       FOR RX NODES, PID MUST ALWAYS BE 0.
        CALL KILLPROC(15, 3)

C       KILL PROCESS 10 ON ALL NODES.
        CALL KILLPROC(-1, 10)

C       KILL ALL PROCESSES ON ALL NODES.
        CALL KILLPROC(-1, -1)
```

Environment

Host, Node

Languages

C, Fortran

KILLPROC() *(cont.)*

KILLPROC() *(cont.)*

Description of Parameters

node is an integer value that identifies the node(s) on which you want to terminate a process. A host id is invalid.

pid for CX„ is an integer value that specifies the id of the process that you want to terminate. *Pid* must match a *pid* given for a previous load, or it may be `-1` to terminate all processes. Valid *pids* include any integer value, but negative numbers other than `-1` are reserved for system programs. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on RX nodes).

Discussion

Use **killproc()** to kill the node process specified by *pid* and *node*. It has no effect on messages already sent from or in transit to the process. It causes the selected process to terminate, but does not wait for a process to complete, or flush messages related to that process.

If *node* is a positive integer, the specified process(es) on that node is killed.

If *node* is a negative number up to a certain value, it causes a "global" kill. If *node* is `-1`, the specified process on all nodes in the cube is killed. Other negative numbers kill the process(es) in a set of the nodes surrounding the sender. The dimension of this cube is calculated as (*d* - **nodedim()** `-1`), where *d* is the dimension of the subcube, and is less than or equal to **nodedim()**.

Errors

`killproc: Invalid pid`

Use a non-negative number. For RX nodes, *pid* must be 0.

`killcube: Invalid node`

Use **numnodes()** to determine cube size and **myhost()** to determine host number.

See Also

load(), flushmsg(), killcube(), waitone(), waitall()

KILLSYSLOG() KILLSYSLOG()

Terminate *syslog* process. This is intended for use by host programs and is not available for node programs.

Synopsis

SUBROUTINE **KILLSYSLOG**()

Parameter Declarations

None

Return Value

None

Environment

Host

Languages

C, Fortran

Description of Parameters

None

KILLSYSLOG() *(cont.)*

KILLSYSLOG() *(cont.)*

Discussion

Use **killsyslog()** to kill the *syslog* process and reset the standard output to the terminal (*/dev/tty*). It can only be used to kill the *syslog* process started by **setsyslog()**; it cannot be used to kill the *syslog* process started by the **syslog** command.

The purpose of the *syslog* process is to make sure that output from host programs goes through the same file server used by the node programs. To change file servers to the current working environment, the *syslog* process must be killed and restarted by invoking **setsyslog()** to make sure the output goes to the correct file server.

Errors

```
syslog: No active syslog exists
```

To start the *syslog* process, use **setsyslog()**.

```
syslog: There is no attached cube
```

Use **getcube()** to get a cube.

See Also

setsyslog(), **newserver()**

LED() LED()

Turn the node board's green LED (light emitting diode) on or off. This is intended for use by node programs and is not available for host programs.

Synopsis

SUBROUTINE **LED**(*lstate*)

Parameter Declarations

INTEGER *lstate*

Return Value

None

Environment

Node

Languages

C, Fortran

Description of Parameters

lstate an integer value specifying on or off state of the node board's green LED. The possible settings are:

1 LED is turned on.

0 LED is turned off.

LED() *(cont.)*

LED() *(cont.)*

Discussion

Use **led()** to take control of the green LED from the node operating system, and turn it on or off, depending upon the value of *lstate*. Behavior is not defined if values other than `0` or `1` are used.

Errors

None

LOAD() LOAD()

Load a node process.

Synopsis

SUBROUTINE **LOAD**(*filename, node, pid*)

Parameter Declarations

CHARACTER *filename*(*)*
INTEGER *node, pid*

Return Value

None

Example

The following statement loads the file named "NODE" into all nodes in the current partition, and assigns it a process id of 0 on all nodes.

```
          .
          .
CALL LOAD('NODE', -1, 0)
          .
          .
```

Environment

Host, Node

Languages

C, Fortran

LOAD() *(cont.)*

LOAD() *(cont.)*

Description of Parameters

filename is a pointer to a character string which specifies the pathname of the program file that you want to load into the node. Valid pathnames are ASCII character strings.

node is an integer value that identifies the node you want to load a process into (use `-1` to select all nodes). Nodes within a partition are numbered consecutively from 0. A host id is invalid. A process can use **mynode()** to obtain its node id after it begins execution.

pid is an integer value that specifies the process id assigned to the loaded node process.

Discussion

Use **load()** to load the file designated by *filename* onto the specified *node(s)*, assign *pid* to the process(es), and begin execution. The host id is invalid because this routine does not load processes into the host. The *node* and *pid* specified in the load operation are used in send, receive, and kill operations. RX nodes allow only one process, and *pid* must be 0.

On CX nodes, valid *pids* include any integer value, but negative numbers are reserved for system programs. An error results if you specify a *pid* that is already in use by a previously loaded process on the same node. A *pid* may be reused as soon as a process completes. Up to 20 processes per CX node are allowed. A process can use **mypid()** to obtain its process id after it begins execution.

Errors

```
load: There is no attached cube
```

Use **getcube()** to allocate a cube and **attachcube()** to change cubes within your program.

```
load: Invalid pid
```

The *pid* must not be negative. For RX nodes, *pid* must be `0`.

```
load: Invalid node
```

Node number must be in range. Use **numnodes()** to determine cube size.

LOAD() *(cont.)*

LOAD() *(cont.)*

`load: Out of process slots`

Use fewer processes.

`load: Pid already in use`

Use a different *pid.*

`load: Not enough memory`

Reduce the size of the process.

`load: Invalid loader record`

Internal error, try again. System administrator may have to **bootcube.**

`load: Invalid loader message`

Internal error, try again. System administrator may have to **bootcube.**

`load: No such file or directory`

Correct the *filename.*

`load: Invalid object file`

Specify a loadable file.

`load: There are no cubes allocated`

Use **getcube()** to allocate a cube.

`load: No pid defined`

Use **setpid()** to define a host process id.

See Also

killcube(), mynode(), mypid()

LSEEK() LSEEK()

Move the read/write file pointer.

Synopsis

INTEGER FUNCTION **LSEEK**(*unit, offset, whence*)

Declarations of Parameters

INTEGER *unit*
INTEGER *offset*
INTEGER *whence*

Return Values

The resulting file pointer location, as measured in bytes from the beginning of the file is returned.

Example

The following example uses **lseek()** to move the file pointer back to the beginning of a file after a write and just before a read. The example first creates the file with **open()**. Then, after setting up the buffer, the example uses **cwrite()** to write the data in the buffer into the file. It then assumes that it is necessary to read the file, so **lseek()** locates the top of the file, and **cread()** then reads the information into the buffer you created:

```
            .
            .
C     CREATE A FILE ON CFS NAMED MYDATA
      OPEN(UNIT=3, FILE='/cfs/mydata', FORM = 'UNFORMATTED')
C     SET UP A BUFFER NAMED "BUFFER" WITH DATA
            .
            .
C     WRITE CONTENTS OF BUFFER TO FILE
       CALL CWRITE(3, BUFFER, BUFLEN)
C     DO OTHER PROCESSING
            .
            .
C     FIND THE FRONT OF THE FILE
       CALL LSEEK(3, 0, 0)
C     READ CONTENTS OF THE FILE TO THE BUFFER
       CALL CREAD(3, BUFFER, BUFLEN)
```

LSEEK() *(cont.)*

LSEEK() *(cont.)*

Environment

Node

Languages

Fortran

Description of Parameters

unit is the unit you assign the file with the **open()** call.

offset an integer value that affects the current file pointer depending upon the value of *whence*.

whence an integer value of `0, 1,` or `2` that determines how the offset value is used to set the pointer. These values have the following effects:

`0`	Sets the file pointer to *offset*.
`1`	Sets the file pointer to the current location of the file pointer plus *offset*.
`2`	Sets the file pointer to the current file size plus *offset*.

Discussion

Use **lseek()** to move the read/write file pointer in the file designated by the *unit* parameter, which should be the value you assigned the *unit* parameter of **open()**. The *offset* parameter is a value in bytes that depends upon the value of the *whence* parameter. If *whence* is `0`, the pointer is set to *offset* bytes from the beginning of the file. If *whence* is `1`, the pointer is set to the *offset* added to the current location of the file pointer. If *whence* is `2`, the pointer is set to the size of the file plus *offset* (the units are bytes.)

If successful, the location of the new file pointer is returned in units of bytes.

When you open a file in APPEND access, the file pointer is set to the end of the file before every write. If you issue **lseek()** before the write, the effect of the **lseek()** is nullified. APPEND access is a Fortran VMS extension.

LSEEK() *(cont.)*

LSEEK() *(cont.)*

Errors

`lseek: Bad file number`

Use the unit number assigned by **open()**.

`lseek: No such unit`

Unit must be a value no larger than 100.

`lseek: Mixed file operations`

In I/O mode 2 or 3, nodes are attempting different operations (reads and writes) to a shared file. Ensure that nodes are performing the same operation.

See Also

cread(), cwrite(), iread(), iwrite()
i860™ 64-Bit Microprocessor Fortran Language Reference Manual: **OPEN()**
UNIX System V Programmer's Reference Manual: **open(2)**

LSIZE() LSIZE()

Increase the size of a file.

Synopsis

INTEGER FUNCTION **LSIZE**(*unit, offset, whence*)

Parameter Declarations

INTEGER *unit*
INTEGER *offset*
INTEGER *whence*

Return Values

The file size in bytes is returned if the call is successful; otherwise, it returns a `-1`.

Example

The following statement of the **lsize()** call increases the size of the file designated by the unit variable *UNIT1* (the value you assigned the *UNIT* variable of the Fortran **open()** call) to one million bytes.

```
      INTEGER SIZE UNIT1
          .
          .
C     INCREASE FILE SIZE TO 1 MILLION BYTES
      SIZE = LSIZE(UNIT1, 1000000, 0)
```

Environment

Node

Languages

C, Fortran

LSIZE() *(cont.)*

LSIZE() *(cont.)*

Description of Parameters

unit is the unit you assign the file with the **open()** call.

offset a long integer value that affects the current size of the file depending upon the value of *whence*.

whence an integer value of `0`, `1`, or `2` that determines how the offset value is used to set the size of the file. These values have the following effects:

`0`	sets the file size to the greater of the current size or *offset*.
`1`	sets the file size to the greater of the current size or the current location of the file pointer plus *offset*.
`2`	sets the file size to the greater of the current size or the current size plus *offset*.

If the new size specified by *offset* and *whence* will be greater than the available disk space, **lsize()** silently allocates the available space and returns the actual new size.

Discussion

Use **lsize()** to increase the size of a file.

The primary use of **lsize()** is to ensure that enough space is allocated before you do any sensitive calculations. If *offset* is positive, **lsize()** ensures that the disk blocks are allocated to the file from the starting position to the ending position (as specified by *whence* and *offset*). If allocation has been restricted by **restrictvol()**, any new disk blocks allocated will come from the restricted set.

If the call is successful, the call returns the new size of the file in bytes. If the call is not successful, the call will fail and return a `-1`.

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

LSIZE() *(cont.)*

LSIZE() *(cont.)*

Errors

`lsize: Invalid size`

Cannot be used in extended file sizes. Use **esize()**.

`lsize: Bad file number`

Use the file descriptor returned by **open()**.

`lsize: No space left on device`

Not enough space on device to which you are writing. Create more space in file system, or if you have restricted the disks to which you are writing with **restrictvol()**, re-execute **restrictvol()** to include more drives.

`lsize: Not implemented on the SRM`

Available only for CFS files.

See Also

restrictvol(), **esize()**
i860™ 64-Bit Microprocessor Fortran Language Reference Manual: **OPEN()**

MASKTRAP() MASKTRAP()

Enable or disable a receive trap. Intended for use by node programs and is not available for host programs.

Synopsis

INTEGER FUNCTION **MASKTRAP**(*state*)

Parameter Declarations

INTEGER *state*

Return Value

An integer that is the previous value of *state*.

Example

The following example uses **masktrap()** to prevent critical processing from being interrupted by the execution of the handler procedure called at the completion of the **hrecv()**.

```
      CHARACTER BUF*1000
      EXTERNAL PROC
      INTER OLDVAL, NEWVAL

      CALL HRECV(101, BUF, LEN(BUF), PROC)
C      PROC WILL BE CALLED WHEN THE RECEIVE COMPLETES
      OLDVAL = MASKTRAP(1)
C      PROCESSING HERE WILL NOT BE INTERRUPTED
      NEWVAL = MASKTRAP(OLDVAL)
C      PROC CAN BE CALLED AGAIN
```

Environment

Node

Languages

C, Fortran

MASKTRAP() *(cont.)*

MASKTRAP() *(cont.)*

Description of Parameters

state Either `1` or `0`; `1` is used to block receive traps and `0` is used to enable receive traps.

Discussion

Use **masktrap()** to protect critical processing from being interrupted by the execution of the handler procedure called when an **hrecv()**, **hsend()**, or **hsendrecv()** completes. If these complete while traps are masked, the trap will be held until unmasked. Assigning the *state* parameter a value of `1` blocks the receive trap until **masktrap()** is called again, assigning state a value of `0`.

Errors

None

See Also

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

MCLOCK() MCLOCK()

Return elapsed time.

Synopsis

INTEGER FUNCTION **MCLOCK**()

Parameter Declarations

None

Return Value

An integer containing a relative time value.

Example

The following example illustrates how to use **mclock()** to determine elapsed time of a process.

```
      INTEGER STIME, ETIME, TTIME
          .
          .
C FIND STARTING TIME
      STIME = MCLOCK()
C EXECUTE PROCESS
          .
          .
C FIND ENDING TIME
      ETIME = MCLOCK()
C FIND ELAPSED TIME
      TTIME = ETIME - STIME
```

MCLOCK() *(cont.)*

MCLOCK() *(cont.)*

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

None

Discussion

Use **mclock()** as a simple mechanism to measure time intervals; it returns the value of a counter that reflects relative time in milliseconds. Programs that use **mclock()** should obtain an initial time value and interpret all other time values relative to this initial value.

When **mclock()** is called from a host program, the relative time (in milliseconds) it returns counts only the time that the UNIX system was actually used to execute your host program, any child processes the host program creates, and system operations related to the host program.

On the nodes, the value returned by **mclock()** represents the actual elapsed milliseconds since the node was initialized. The **bootcube** command, used by the system administrator, initializes the nodes.

It is recommended that you do not use **mclock()** to synchronize different aspects of process execution because every node maintains its own clock value and values on different nodes may vary after initialization.

On both the host and nodes, the **mclock()** value rolls over approximately every 25 days.

Errors

None

MSGCANCEL() MSGCANCEL()

Cancel an asynchronous send or receive operation.

Synopsis

SUBROUTINE **MSGCANCEL**(*id*)

Parameter Declarations

INTEGER *id*

Return Value

None

Example

The following example uses **msgcancel()** to cancel the receipt of a message and clear its id if an error occurs.

```
      CHARACTER*100 BUF
      INTEGER MSGID, TYPE
      LOGICAL ERROR
           .
           .
      MSGID = IRECV(TYPE, BUF, LEN(BUF))
           .
           .
C PERFORM TASKS WHILE WAITING FOR THE RECEIVE TO COMPLETE.
           .
           .
      IF (ERROR) CALL MSGCANCEL(MSGID)
```

MSGCANCEL() *(cont.)*

MSGCANCEL() *(cont.)*

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

id is the message id returned when you invoke an **isend()** or **irecv()** operation.

Discussion

Use **msgcancel()** to cancel an asynchronous send or receive operation. The *id* parameter identifies the operation to cancel. *Id* must correspond to the value returned by a previous **isend()** or **irecv()**. When **msgcancel()** returns, you do not know whether the send or receive operation completed, but you do know that the operation is no longer active, that the buffer may be reused. You also know that the message id is no longer valid, and so is available for use by further asynchronous send and receive operations, and cannot be used further in **msgdone()** or **msgwait()**.

Errors

```
msgcancel: Invalid message id
```

Use message id returned by **irecv()** or **isend()**.

```
msgcancel: No pid defined
```

Use **setpid()** to define a host process id.

See Also

irecv(), **isend()**

MSGDONE() MSGDONE()

Determine whether an asynchronous send or receive operation is complete.

Synopsis

INTEGER FUNCTION **MSGDONE**(*id*)

Parameter Declarations

INTEGER *id*

Return Values

0	indicates that the send or receive message is not yet complete. In this case, you can use the same id again until the send or receive operation completes.
1	indicates that the message is complete and that the associated message buffer is available for reuse, in the case of a send operation, or that the buffer contains valid data in the case of a receive operation.

Example

This example uses **msgdone()** to determine whether an **isend()** has completed sending the message. If so, it reuses the message buffer, otherwise it does not use it because the message is in the buffer and has not yet been sent.

```
      CHARACTER*100 BUF
      INTEGER MSGID, TYPE
              .
              .
      MSGID = ISEND(TYPE, BUF, LEN(BUF), 1, 0)
              .
              .
      IF (MSGDONE(MSGID) .EQ. 1) THEN
C      YOU MAY NOW RE-USE THE MESSAGE BUFFER.

      ELSE

C      THE MESSAGE BUFFER STILL NEEDS TO BE PROTECTED.

      END IF
```

MSGDONE() *(cont.)*

MSGDONE() *(cont.)*

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

id is the message id returned when you invoke an **isend()** or **irecv()** operation.

Discussion

Use **msgdone()** to determine whether an **isend()** or **irecv()** operation is complete. When **msgdone()** returns a `1`, the **isend()** or **irecv()** operation is complete. This means that the associated message buffer is available for reuse in the case of an **isend()**, or the buffer contains valid data in the case of an **irecv()**.

It also releases the id number used by the asynchronous message passing call to which it refers. You must use either **msgdone()** or **msgwait()** after an asynchronous message passing call to ensure that the id is cleared. You cannot call **msgdone()** or **msgwait()** with the same id parameter after **msgdone()** returns `1` because the send or receive operation is complete and the id is no longer valid. The *id* parameter identifies the operation you are checking status on. The *id* must correspond to the value returned by a previous **isend()** or **irecv()**.

When **msgdone()** returns `1` for a receive operation, you can use the **info...()** calls get more information about the message.

MSGDONE() *(cont.)*

MSGDONE() *(cont.)*

Errors

`msgdone: Invalid message id`

Use message id returned by **irecv()** or **isend()**.

`msgdone: No pid defined`

Use **setpid()** to define a host process *id*.

`msgdone: Received message too long for buffer`

Make sure the buffer is large enough to hold the message.

See Also

infocount(), **infonode()**, **infopid()**, **infotype()**, **irecv()**, **isend()**, **msgwait()**, **msgcancel()**

MSGWAIT() MSGWAIT()

Wait for completion of an asynchronous send or receive operation.

Synopsis

SUBROUTINE **MSGWAIT**(*id*)

Parameter Declarations

INTEGER *id*

Return Value

None

Example

This example uses **msgwait()** to stop further execution of the process until an **irecv()** posted earlier has been completed. It also clears the id for use by subsequent sends and receives.

```
      INTEGER RESULT
      INTEGER MSGID, TYPE
            .
            .
      MSGID = IRECV(TYPE, RESULT, 4)

C  PERFORM TASKS THAT DO NOT USE 'RESULT'.
            .
            .
C  STOP UNTIL THE MESSAGE IS RECEIVED.
      CALL MSGWAIT(MSGID)

C  NOW YOU CAN USE 'RESULT'.
```

MSGWAIT() *(cont.)*

MSGWAIT() *(cont.)*

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

id is the message id returned when you invoke an **isend()** or **irecv()**.

Discussion

Use **msgwait()** to block a process until an **isend()** or **irecv()** operation is complete, and when the operation is complete, it releases the *id* number used by the asynchronous message passing call to which it refers. Because a limited number of message IDs are available, you must use either **msgdone()** or **msgwait()** after an asynchronous message passing call. This ensures that you do not run out of message IDs, which would cause an error. The *id* parameter identifies the operation that you are waiting for. The *id* must correspond to the value returned by a previous **isend()** or **irecv()** call.

When the message is complete, **msgwait()** returns and the associated message buffer is available for reuse, in the case of a send operation, or the buffer contains valid data, in the case of a receive operation.

When **msgwait()** returns after waiting for an **irecv()** to complete, you can use **info...()** calls to get more information about the message. You cannot use the *id* parameter again because the send or receive operation is complete.

MSGWAIT() *(cont.)*

MSGWAIT() *(cont.)*

Errors

`msgwait: Invalid message id`

Use message id returned by **irecv()** or **isend()**.

`msgwait: No pid defined`

Use **setpid()** to define a host process *id*.

`msgwait: Received message too long`

Make sure the buffer is large enough for buffer to hold the message.

See Also

infocount(), infonode(), infopid(), infotype(), irecv(), isend(), msgcancel(), msgdone()

MYHOST() MYHOST()

Obtain the id of the host machine.

Synopsis

INTEGER FUNCTION **MYHOST**()

Parameter Declarations

None

Return Value

The host id is returned.

Example

The following example uses **myhost()** to obtain the host id for a **csend()** operation from a node to the host.

```
      CHARACTER*100 MSG
      INTEGER HOSTID
          .
          .
      HOSTID = MYHOST()

C SEND A TYPE 10 MESSAGE TO PROCESS 1 ON THE HOST.

      CALL CSEND(10, MSG, LEN(MSG), HOSTID, 1)
```

Environment

Host, Node

Languages

C, Fortran

MYHOST() *(cont.)*

MYHOST() *(cont.)*

Description of Parameters

None

Discussion

Use **myhost()** to return the host id of the caller's host machine for use in send and receive operations. For host processes, **myhost()** returns the same id as **mynode()** (node id of the calling process). In a **D4** cube, with cubes numbered 0-15, **myhost()** would return `16`.

Errors

```
myhost: No pid defined
```

Use **setpid()** to define a host process *id*.

See Also

csend(), isend(), mynode()

MYNODE() MYNODE()

Obtain the node id of the calling process.

Synopsis

INTEGER FUNCTION **mynode()**

Parameter Declarations

None

Return Value

The node id of the process that initiated the routine is returned.

Example

The following example uses **mynode()** to provide the node id to **killproc()**.

```
      INTEGER THISNODE
          .
          .
C OBTAIN THE LOGICAL NODE ID OF THIS NODE

      THISNODE = MYNODE()
C KILL PROCESS 0 ON THIS NODE
      CALL KILLPROC(THISNODE, 0)
```

Environment

Host, Node

Languages

C, Fortran

MYNODE() *(cont.)*

MYNODE() *(cont.)*

Description of Parameters

None

Discussion

Use **mynode()** to return the node id of the calling process. Nodes within a partition are numbered consecutively from 0. The host's id is the return value of **myhost()**. The node id is used in send and kill operations.

Errors

```
mynode: No pid defined
```

Use **setpid()** to define a host process id.

See Also

myhost(), **mypid()**, **setpid()**, **numnodes()**

MYPID() MYPID()

Obtain the NX/2 process id of the calling process.

Synopsis

INTEGER FUNCTION **MYPID()**

Parameter Declarations

None

Return Value

The process id is returned. For RX nodes, *pid* is always 0.

Example

This example uses **mypid()** to return the current process id to be used in a **csend()**.

```
      CHARACTER *100 BUF
      INTEGER TYPE
            .
C SEND MSG TO PROCESSES ON ALL NODES WITH SAME PID AS THIS
            .
      CALL CSEND(TYPE, BUF, LEN(BUF), -1, MYPID())
```

Environment

Host, Node

Languages

C, Fortran

MYPID() *(cont.)*

MYPID() *(cont.)*

Description of Parameters

None

Discussion

Use **mypid()** to return the cube process id of the calling process. On a node, this is the *pid* that was supplied when the process was loaded. On the host, this *pid* is determined by **setpid()**. (This cube process id should not be confused with the UNIX process id.) The *pids* are used in send and kill operations. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on RX nodes).

Errors

```
mypid: No pid defined
```

Use **setpid()** to define a host process id.

See Also

csend(), csendrecv(), flushmsg(), isend(), isendrecv(), load(), killcube(), killproc(), setpid()

NEWSERVER() NEWSERVER()

Start a new file server for the specified cube.

Synopsis

SUBROUTINE **NEWSERVER**(*cubename*)

Parameter Declarations

CHARACTER *cubename*(*)*

Return Values

None

Example

Before running a program named **host**, enter the **getcube** command at the system prompt. With no redirection, the node output is the standard output (*stdio*) of the **getcube** command. However, you intend to run your program twice, and you want the node output in two separate files. If you do not use **newserver()** within the program, the node output will go to *stdio* of the **getcube** command. If you do use it, the node output will go to the file indicated on the invocation line.

At the system prompt:

```
%getcube
%host > file1
%host > file2
```

Within the host program:

```
      CHARACTER*11 CUBE
      CUBE ='defaultname'
           .
           .
C MAKE THE OUTPUT GO TO THE FILE NAMED AT PROGRAM INVOCATION
      CALL NEWSERVER(CUBE)
```

NEWSERVER() *(cont.)*

NEWSERVER() *(cont.)*

Environment

Host

Languages

C, Fortran

Description of Parameters

cubename refers to the name of the cube, assigned with **getcube**. Valid names include any ASCII string, 15 characters or less. Names longer than 15 characters are truncated to 15 characters. If the parameter is null or points to a null string, the currently attached cube is used. If you specify a cube that has not been allocated with **getcube** or there is no current attached cube, an error message results and the process terminates. Use the **cubeinfo** command or call to find out the names of allocated cubes.

Discussion

When you allocate a cube with **getcube**, a file server to handle I/O for the nodes is automatically created. If you want to redirect the output of your node process to a different file or device at program invocation, you must use **newserver()** within your program to allow the output to go to the desired place. The **newserver()** call actually kills the original file server and starts a new one. When you issue **newserver()**, changes that you make to your environment are passed to the new file server.

For information on redirection, refer to the *UNIX System V User's Guide*.

NEWSERVER() *(cont.)*

NEWSERVER() *(cont.)*

Errors

`newserver: Cubename does not exist`

Use **cubeinfo** to determine cube names.

`newserver: Commser not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with the **bootcube** or **rebootcube** command.

`newserver: Lifeline not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with the **bootcube** or **rebootcube** command.

See Also

cubeinfo(), getcube(), syslog()

NODEDIM() NODEDIM()

Returns the dimension of the allocated cube.

Synopsis

INTEGER FUNCTION **NODEDIM**()

Parameter Declarations

None

Return Value

An integer from `0` to `7` representing the dimension of the cube.

Example

The following example uses **nodedim**() to return the size of the cube obtained by **getcube**().

```
      INTEGER ALPHADIM
          .
          .
      GETCUBE ('ALPHA', '32','', 0)
      ALPHADIM = NODEDIM()
C THE DIMENSION SHOULD BE 5
C 2**5 IS 32
```

Environment

Host, Node

Languages

C, Fortran

NODEDIM() *(cont.)*

NODEDIM() *(cont.)*

Description of Parameters

None

Discussion

Use **nodedim()** to return the dimension of the allocated cube. For example, the dimension of a 64-node cube is 6 ($2^6 = 64$). If the number of nodes allocated is 8, **nodedim()** returns 3. Use **numnodes()** to return the number of allocated nodes.

Errors

```
nodedim: No pid defined
```

Use **setpid()** to define a host process id.

See Also

getcube(), numnodes()

NUMNODES() NUMNODES()

Obtain the number of nodes in the cube.

Synopsis

INTEGER FUNCTION NUMNODES()

Parameter Declarations

None

Return Value

An integer in the range `1-128`, representing the number of nodes in a cube.

Example

The following example gets two cubes, and then uses **numnodes()** to find the highest-numbered node in the cube "BETA" and the total number of nodes in the cube "ALPHA."

```
      INTEGER LASTNODE, TOTALNODES
            .
            .
      CALL GETCUBE('ALPHA', 'd3m4', 'SAXON', 1)
      CALL GETCUBE('BETA', '32vx', 'VIKING', 1)
      CALL SETPID(1)
C     THE LAST NODE ON THE CUBE NAMED 'BETA' IS 31.
      LASTNODE = NUMNODES() - 1
C     THE TOTAL NUMBER OF NODES ON THE CUBE NAMED 'ALPHA' is 8.
C     2**3 = 8.
      CALL ATTACHCUBE('ALPHA')
      TOTALNODES = NUMNODES()
```

NUMNODES() *(cont.)*

NUMNODES() *(cont.)*

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

None

Discussion

Use **numnodes()** to return the number of nodes allocated to the cube. For instance, if you allocate a cube of size 16 with **getcube()**, **numnodes()** will return `16` (2^4). Use **getcube()** to assign a number of nodes to an application.

Errors

```
numnodes: No pid defined
```

Use **setpid()** to define a host process id.

See Also

getcube(), **nodedim()**

PLOGON(), PLOGOFF() PLOGON(),PLOGOFF()

Log process related information such as cube allocation, cube deallocation, process initialization, and process completion. Requires root privilege to run.

Synopsis

SUBROUTINE **PLOGON**(*append, noprocs, query, logfile*)
SUBROUTINE **PLOGOFF**()

Parameter Declarations

PLOGON()

INTEGER*2 *append, noprocs, query*
CHARACTER *logfile*(*)

PLOGOFF()

None.

Return Value

None

Example

This example turns on logging to the file */usr/joelc/mylog*. Booting the iPSC system, turning on/off logging, cube allocation, cube deallocation, process initialization, and process completion are logged.

```
      CHARACTER*16 FNAME
      FNAME = '/usr/joelc/mylog'
          .
          .
C LOGGING TURNED ON
      CALL PLOGON(0,0,0,fname)
          .
          .
```

PLOGON(), PLOGOFF() *(cont.)*

PLOGON(), PLOGOFF() *(cont.)*

```
C LOGGING TURNED OFF
      CALL PLOGOFF()
         .
         .
```

Environment

Host, Node

Languages

C, Fortran

Description of Parameters

append is an integer. When it is `1`, the logging information is appended to *logfile* (if *logfile* exists). When it is `0`, the logging information overwrites an existing *logfile*.

noprocs is an integer. When it is 1, no process loading or termination information is logged. When it is 0, process initialization, and process completion are logged as well as execution of **bootcube**, **bootcube()**, **plogon**, **plogon()**, **plogoff()**, **plogoff**, **getcube**, **getcube()**, **relcube**, and **relcube()**.

query is an integer that causes **plogon()** to to return the name of the current logfile in the parameter *logfile*. No other action is performed.

logfile is a string that identifies the pathname of the log file. If it is null, the default */usr/ipsc/log/process.log* is used.

Discussion

plogon() turns on the logging function, and **plogoff()** turns it off. If already logging to a file, **plogon()** continues logging to the same file or to a new one if a different file is specified.

The process executing **plogon()** or **plogoff()** must be running at root privilege. If the process does not have root privilege, it aborts when it encounters the call.

PLOGON(), PLOGOFF() *(cont.)*

PLOGON(), PLOGOFF() *(cont.)*

Errors

`plogon: Permission denied`

Only the superuser can run **plogon()**.

`plogoff: Permission denied`

Only the superuser can run **plogoff()**.

`plogoff: No active log file exists`

plogoff() cannot find a log function to terminate.

RELCUBE() RELCUBE()

Release a cube.

Synopsis

SUBROUTINE **RELCUBE** (*cubename*)

Parameter Declarations

CHARACTER *cubename*(*)*

Return Value

None

Example

The following example gets a cube with **getcube()**, and later uses **killcube()** to terminate any remaining processes, and finally uses **relcube()** to release the cube.

```
      CALL GETCUBE('BIG', '128', '', 0)
            .
            .
C  CLEAN UP AND RELEASE THE CUBE.

      CALL KILLCUBE(-1, -1)
      CALL RELCUBE('BIG')
```

Environment

Host

Languages

C, Fortran

RELCUBE() *(cont.)*

RELCUBE() *(cont.)*

Description of Parameters

cubename refers to a character string that is the name of the cube that you want to release. You name a cube when you invoke **getcube**. Valid names include any ASCII string, 15 characters or less. Names longer than 15 characters are truncated to 15 characters. If the parameter is null or points to a null string, the currently attached cube is used. If you specify a cube that has not been allocated with **getcube** or if there is no cube attached, an error message results and the process terminates. Use **cubeinfo()** to find out the names of allocated cubes.

Discussion

Use **relcube()** to release a cube that you allocated when you invoked **getcube()**. It also kills the file server and any processes attached to *cubename* and any node processes running on the cube.

Errors

`relcube: Cubename does not exist`

Use **cubeinfo** to determine cube names.

`relcube: Commser not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with **bootcube**.

`relcube: Lifeline not responding`

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with **bootcube**.

See Also

cubeinfo(), getcube()

RESTRICTVOL() RESTRICTVOL()

Restrict or return the disk volumes to which a file can be allocated.

Synopsis

INTEGER FUNCTION **RESTRICTVOL**(*unit, nvol, vollist*)

Parameter Declarations

INTEGER *unit*
INTEGER *nvol*
INTEGER *vollist* (*)

Return Value

If *nvol* is a positive integer or `0`, and the volume restriction is set successfully, the return value is `0`. If the restriction is not successful, a `-1` is returned.

If *nvol* is a negative number, the call returns the number of volumes currently available for allocation.

Example

The following example shows a call to **restrictvol()** with `-1` for the *unit* parameter and one volume to ensure that the file header is created on disk volume 1. The second call to **restrictvol()** restricts the file specified by the UNIT value 3 plus disk volumes 1 and 4. Later, following an **lsize()** call increasing the file size, the programmer removes the restriction. Then, following data writes to the file, the programmer uses **restrictvol()** to find out which volumes have actually been used (a maximum array size for *vollist* is used in the example).

RESTRICTVOL() *(cont.)*

RESTRICTVOL() *(cont.)*

```
      INTEGER VOL1, VOLLST(2)
      INTEGER NEWVOLLIST(128)
      INTEGER NUMVOLS, ERRVAL
      DATA VOL1/1/
      DATA VOLLST/1,4/
           .
           .
      ERRVAL = RESTRICTVOL(-1, 1, VOL1)
      OPEN(UNIT=3, FILE= 'MYDATA', FORM = 'UNFORMATTED')
      ERRVAL = RESTRICTVOL(3, 2, VOLLST)
C WHEN YOU WRITE DATA TO THIS FILE, ALLOCATION WILL BE
C RESTRICTED TO VOLUMES 1 AND 4
           .
           .
C REMOVE PREVIOUS VOLUME RESTRICTION AFTER AN LSIZE() CALL
      ERRVAL = RESTRICTVOL(UNIT,0,0)
C PROCESSING FOLLOWS
           .
           .
C RETURN THE NUMBER OF VOLUMES USED AND STORE THE LIST
C OF VOLUMES IN NEWVOLLIST
      NUMVOLS = RESTRICTVOL(UNIT, -1, NEWVOLLIST)
           .
           .
```

RESTRICTVOL() *(cont.)*

RESTRICTVOL() *(cont.)*

Environment

Node

Languages

C, Fortran

Description of Parameters

unit is a positive integer value which must be the value you assigned to the *unit* parameter of **open()** when you opened the file, or `-1` which indicates that the file header created by the next **open()**, **creat()**, or **dup()** call should be restricted.

nvol The value of this parameter determines the effect of the call:

- If *nvol* is a positive integer, **restrictvol()** expects to find the values for an array of that size in the *vollist* parameter, and restricts subsequent file allocation to the specified volume(s).
- If *nvol* is `0`, the file may be allocated to all volumes in the system (no restrictions). This is the default.
- If *nvol* is a negative number, the current volume list is put into the variable *vollist*, and the number of volumes to which the specified file is allocated is returned. *vollist* must be large enough to hold all information.

vollist An array of integers containing the volume numbers to which file allocation is to be restricted if *nvol* is positive. If *nvol* is negative, the volume numbers to which the specified file has been allocated are put into the array. If *nvol* is `0`, the *vollist* parameter is not used.

Discussion

Use **restrictvol()** either to restrict the volumes to which the blocks of a file can be allocated, or to return the volumes available for allocation. Volume restriction must be done on a file-by-file basis. The file is determined by the *unit* parameter, the value of which must be the value you assigned the *unit* parameter of **open()**.

RESTRICTVOL() *(cont.)* RESTRICTVOL() *(cont.)*

When there is no restriction, file blocks are allocated in a round-robin fashion to volumes in order by number. Volumes with no free space are skipped. CFS allocates file blocks to volumes when a file is opened, when it is written beyond the current end of the file or in a previously unallocated space, and when the **lsize()** call is used to extend the file size.

To restrict the number of volumes, the *nvol* parameter must be set to the number of volumes that you want to use for the file, and the *vollist* parameter must contain the volume number(s) that you want to use. If the restriction is successful, the call returns a `0`.

To return the volumes that a file is residing on, the *nvol* parameter must be any negative number, and the *vollist* parameter must be an empty array at least the size of the number of drives on your system, to avoid error. In this case, **restrictvol()** returns the number of volumes used, and puts the volume numbers into the *vollist* array.

Errors

`restrictvol: Invalid argument`

Check arguments.

`restrictvol: Bad file number`

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

`restrictvol: No such unit`

Unit must be a value no larger than 100.

`restrictvol: Fortran runtime error: Unit not open`

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

See Also

lsize()

SETIOMODE() SETIOMODE()

Sets the I/O mode as it performs a global synchronization operation.

Synopsis

SUBROUTINE **SETIOMODE**(*unit, mode*)

Parameter Declarations

INTEGER *unit*
INTEGER *mode*

Return Value

None

Example

The following example uses **setiomode()** to set the I/O mode to 2. In I/O mode 2, reads and writes are global operations that must be performed on all nodes at once, and are written to the file and read out again by node number. Therefore, from the point of the **setiomode()** call, all nodes must have the same sequence of reads and writes:

```
      INTEGER BUFLEN1
          .
C      CREATE A FILE ON CFS NAMED MYDATA
        OPEN(UNIT=3, FILE='/cfs/mydata', FORM = 'UNFORMATTED')
C      SET UP A BUFFER NAMED "BUFFER" WITH DATA
          .
C      SET THE I/O MODE OF THE FILE TO 2
      CALL SETIOMODE(3,2)
C      WRITE CONTENTS OF BUFFER TO FILE
      CALL CWRITE(3, BUFFER, BUFLEN)
C      DO OTHER PROCESSING
          .
C      FIND THE FRONT OF THE FILE
      CALL LSEEK(3, 0, 0)
C      READ CONTENTS OF THE FILE TO THE BUFFER
      CALL CREAD(3, BUFFER, BUFLEN)
```

SETIOMODE() *(cont.)*

SETIOMODE() *(cont.)*

Environment

Node

Languages

C, Fortran

Description of Parameters

unit is the unit you assign the file with **open()**.

mode an integer from 0 through 3 that sets the I/O mode of the designated file. Briefly, these modes are defined as follows:

0 **Individual file pointer.** Each process has its own file pointer. Nodes can read or write information anywhere in the file. This is the default mode for all shared files and treats processes on different nodes in the same way that UNIX treats different processes.

1 **Common file pointer.** There is one file pointer for all nodes. I/O requests are serviced or on a first-come, first-serve basis. Nodes can read and write at any time but reads and writes to the file are done one at a time using a common file pointer.

2 **Synchronized, common file pointer, variable length requests.** There is one file pointer for all nodes. I/O requests must be read by all nodes at once, and reads and writes are performed in node order. Records may be variable length.

3 **Synchronized, common file pointer, fixed length requests.** Same as mode 2 except that the amount of data read or written in each of the global requests must be the same.

SETIOMODE() *(cont.)*

SETIOMODE() *(cont.)*

Discussion

Use **setiomode()** to set the I/O mode of the file specified by the *unit* parameter of **open()**. At the same time, **setiomode()** performs a global synchronizing operation like that of the **gsync()** call. Every node must use the same *unit*, and the file pointer must be in the same position in the file for each node at the time the call is made.

When you open a shared file with the **open()** statement, it is assigned `STATUS='UNKNOWN'`. Files that are created automatically by writing to a *unit* for which no file has been opened will include the node number in the file name and are shared files. All shared files start in mode 0 (always true on CFS).

Use **iomode()** to return the current mode. There are four possible I/O modes (0-3), described in detail in the following paragraphs.

Mode 0: Individual File Pointer

Mode 0 is the mode in which all shared files are opened. Shared files are those that more than 1 node accesses in a process. In this mode, all nodes may read and write to any part of the file at any time. This is the fastest mode, except for the time required for seeks. If two nodes write to the same place in the file, the data written by the second node to write overwrites the data already there. Therefore, this mode is most useful in situations where all nodes are only reading a data file or where each node is responsible for data in a specific area of a file. This mode has the highest performance.

Mode 1: Common File Pointer

In this mode, reads and writes to the file can come from any node at any time, and requests are serviced on a first-come, first-serve basis. There is a common file pointer. Because requests from nodes can be performed in any order, the data in the file may not have a consistent order from run to run. This mode is most useful for log files. This mode has lower performance. *stdin*, *stdout*, and *stderr* are always in mode 1.

Mode 2: Synchronized, Common File Pointer, Variable Length Requests

Mode 2 treats reads and writes as global operations, in which all nodes must read or write at once. The amount of data read or written may, however, vary from node to node. For writes, data is written to the file in order by node number. Reads are also done in order by node number. This mode treats reads and writes as synchronizing to the extent that a second request by any one node is blocked until all other nodes have performed the first operation. The only valid use for the **lseek()** or **eseek()** call in this mode is for all nodes to seek to the same position in the file prior to a request. If nodes seek to different positions, an error is returned immediately. This mode offers lower performance.

SETIOMODE() *(cont.)*

SETIOMODE() *(cont.)*

Mode 3: Synchronized Ordered Mode, Fixed Length Requests

Mode 3, like mode 2, requires that all nodes must read or write at once. The difference is that for any given read or write, all nodes must do the same operation with the same buffer size. For writes, data is written to the file in order by node number. Reads are also done in order by node number. The only valid use for the **lseek()** or **eseek()** call in this mode is for all nodes to seek to the same position in the file prior to a request. If nodes seek to different positions or read or write different amounts of data, an error is returned eventually. This is a high performance mode.

Errors

`setiomode: Bad file number`

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

`setiomode: Not implemented on the SRM`

Can be executed by the node only; not available on the host.

`setiomode: Bad io mode number`

I/O mode must be set to `0`, `1`, `2`, or `3`.

`setiomode: File is not synchronized`

In I/O modes 2 or 3, all nodes must set the file pointer to the same location.

`setiomode: No such unit`

Unit must be a value no larger than 100.

`setiomode: Fortran runtime error: Unit not open`

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

See Also

cread(), **cwrite()**, **iomode()**, **iread()**, **iwrite()**

SETPID() SETPID()

Sets the process id of a host process.

Synopsis

SUBROUTINE **SETPID**(*pid*)

Parameter Declarations

INTEGER *pid*

Return Value

None

Example

The following example sets the host *pid* for the cube to which you are attached.

```
CALL GETCUBE('ALPHA', '16vx', ' ', 0)
CALL GETCUBE('BETA', '8m8', 'TRAINON', 1)

CALL ATTACHCUBE('ALPHA')
CALL SETPID(99)
          .
          .
CALL ATTACHCUBE('BETA')
CALL SETPID(0)
          .
          .
```

Environment

Host

Languages

C, Fortran

SETPID() *(cont.)*

SETPID() *(cont.)*

Description of Parameters

pid specifies the process id that you are assigning to a host process. Valid *pids* include any integer value, but negative numbers are reserved for system programs.

Discussion

Use **setpid()** to set the process id of the host process. You must call **setpid()** before using any of the message-passing calls such as send and receive operations.

You must allocate a cube before you call **setpid()**. If you change the current cube inside your user process by calling **getcube()** or **attachcube()**, you must call **setpid()**. If you attach to a cube you were previously attached to, **setpid()** must be called. The *pid* used previously can be reused in **setpid()**. Use **mypid()** to obtain the *pid* of a host process after it has been set. **setpid()** is not valid for a node process because the *pid* for a node process is set in **load()**, although using it on a node process does not cause an error.

Errors

```
setpid: Pid already in use
```

Select another *pid*.

```
setpid: Pid already set
```

Do not call **setpid()** again.

```
setpid: There is no attached cube
```

Use **cubeinfo** to find out which cubes you own.

```
setpid: Internal cube usage limit
```

Limit of 10 cubes may be allocated. Try again later when system is not as busy.

```
setpid: Commser not responding
```

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with the **bootcube** command.

SETPID() *(cont.)*

SETPID() *(cont.)*

```
setpid: Lifeline not responding
```

Internal System Resource Manager process error. Cube management processes are not responding. Try rebooting cube with the **bootcube** command.

See Also

getcube(), attachcube(), mypid()

SETSYSLOG() SETSYSLOG()

Start the *syslog* process.

Synopsis

SETSYSLOG(*stdfd*)

Parameter Declarations

INTEGER *stdfd*

Return Values

None

Example

The following example starts the *syslog* process to redirect host process output to *stdio* before loading a program.

```
CALL GETCUBE('ALPHA', '16', ' ', 0)
CALL SETPID(99)
CALL SETSYSLOG(1)
CALL LOAD('PROG1', -1, 0)
          .
          .
```

Environment

Host

Languages

C, Fortran

SETSYSLOG() *(cont.)*

SETSYSLOG() *(cont.)*

Description of Parameters

stdfd is an integer value that indicates where process output is to be sent:

0	redirect output to standard output of the current file server.
1	redirect output to standard error of the current file server.

Discussion

The *syslog* process is a system process used to send the output of a host process to either the standard output or standard error of the current file server of the nodes. Use **setsyslog()** to start the *syslog* process and redirect the standard output of the calling program to the standard input of the *syslog* process. For more information on using the **syslog** command, refer to the *iPSC®/2 and iPSC®/860 User's Guide.*

Errors

`setsyslog: There is no attached cube`

Use **cubeinfo** to find out which cubes you own.

`setsyslog: No pid defined`

Use **setpid()** to define a host process id.

`setsyslog: Active syslog exists`

syslog already in progress.

`setsyslog: Cannot start syslog process`

System will not allow *syslog* process.

See Also

killsyslog(), syslog command

WAITALL() WAITALL()

Wait for all the specified processes to complete.

Synopsis

SUBROUTINE **WAITALL**(*node, pid*)

Parameter Declaration

INTEGER *node, pid*

Return Values

None

Example

The following examples illustrate different ways to use **waitall()**.

```
C       WAIT FOR PROCESS 0 ON NODE 10 TO COMPLETE
C       FOR RX NODES, PID MUST BE 0
      CALL WAITALL(10, 0)

C       WAIT FOR ALL PROCESSES ON NODE 15 TO COMPLETE
      CALL WAITALL(15, -1)

C       WAIT FOR ALL PROCESSES ON ALL NODES TO COMPLETE
      CALL WAITALL(-1, -1)
```

Environment

Host, Node

Languages

C, Fortran

WAITALL() *(cont.)*

WAITALL() *(cont.)*

Description of Parameters

node is an integer value that identifies a particular node where a process resides or `-1` can be used to indicate all nodes. Nodes within a partition are numbered consecutively from 0. A host id is invalid.

pid for CX node, is an integer value that specifies the process id (or `-1` to specify all processes). Valid *pids* can be any integer value, but negative numbers other than `-1` are reserved for system programs. *Pid* is assigned when you load a process. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on the RX node). It is not an error to wait for nonexistent processes; **waitall()** simply returns immediately.

Discussion

Use **waitall()** to block further execution of the process until the selected process(es) complete on the selected node(s). Unlike **waitone()**, it will wait for all processes to complete if `-1` is specified.

The procedure does not return until all specified processes on all specified nodes are completed. The completion codes are discarded.

Errors

```
waitall: Invalid node.
```

See Also

waitone(), load(), killcube(), killproc()

WAITONE() WAITONE()

Wait for a specified process to complete. If all processes or nodes are specified, **waitone()** returns as soon as the first process completes.

Synopsis

SUBROUTINE **WAITONE**(*node, pid, cnode, cpid, ccode*)

Parameter Declarations

INTEGER *node, pid, cnode, cpid, ccode*

Return Value

None

Example

The following example uses **waitone()** to supply information to **write()** when the first node process completes.

```
      INTEGER CNODE, CPID, CCODE
          .
          .
      CALL WAITONE(-1, -1, CNODE, CPID, CCODE)
C     WAIT FOR THE FIRST PROCESS ON ANY NODE TO COMPLETE.
C     FOR RX NODES, PID MUST BE SET TO 0.
      WRITE(*,*) 'PROCESS', CPID, 'ON NODE', CNODE,
     1 'FINISHED WITH COMPLETION CODE', CCODE
```

Environment

Host, Node

Languages

C, Fortran

WAITONE() *(cont.)*

WAITONE() *(cont.)*

Description of Parameters

node is an integer that is either the number of a node where a process resides or is `-1` to indicate all nodes. Nodes are numbered consecutively from 0. A host id is invalid.

pid for CX nodes, is an integer value that specifies the process id. Valid *pids* include any integer value, but negative numbers other than `-1` are reserved for system programs. If `-1` is used to select all processes, **waitone()** returns as soon as the first process completes. *Pid* is assigned when you load a process. For the RX node, *pid* 0 is the only valid process ID (since only a single process can run on the RX node). An error results and the program terminates if a *pid* is specified that does not correspond to an existing process.

cnode, cpid, ccode

When a process with a matching *node* and *pid* completes, the *node, pid,* and completion code of the process are returned in *cnode, cpid,* and *ccode,* respectively. The information returned in *ccode* is equivalent to the status information returned by the UNIX **wait()** call. If the process terminated with a call to **exit()**, *ccode* will contain the least-significant eight bits of the exit code (the argument to **exit()** in the upper eight bits and zero in the lower eight bits). If the process falls off the end of main, the exit code is undefined. If the process is killed or terminated due to an exception, *ccode* will contain the signal number indicating the cause of termination.

On the Intel386, exceptions map to signal numbers in the following way:

SIGNAL #	Intel386™ EXCEPTION #
12 SIGSYS	NA Invalid system call
09 SIGKILL	NA Process killed
08 SIGFPE	0 Integer divide by zero
05 SIGTRAP	1 Single-step interrupt
09 SIGKILL	2 Non-maskable interrupt
05 SIGTRAP	3 Breakpoint interrupt

WAITONE() *(cont.)*

WAITONE() *(cont.)*

SIGNAL #	Intel386™ EXCEPTION #
11 SIGSEGV	4 Integer overflow exception
11 SIGSEGV	5 Array bounds exception
04 SIGILL	6 Invalid opcode
08 SIGFPE	7 Numeric processor not available
08 SIGFPE	9 Numeric processor overrun
08 SIGFPE	11 Segment not present fault
11 SIGSEV	12 Stack overflow
11 SIGSEV	13 General protection fault
11 SIGSEV	14 Memory fault
08 SIGFPE	NA Vector processor error
08 SIGFPE	16 Floating point exception

Discussion

Use **waitone()** to wait for a process to complete on any of the nodes and provide useful information on how the process terminated. If `-1` is used to select all processes, **waitone()** returns as soon as the first process completes. Similarly, if `-1` is used to select all nodes, **waitone()** returns as soon as the specified process on any node completes. This differs from the **waitall()** which does not return until all selected processes on selected nodes complete.

Errors

```
waitone: No pid defined
```

Use **setpid()** to define a host process id.

See Also

waitall(), **waitcube** command

iPSC® SYSTEM COMMANDS, SYSTEM CALLS, and ROUTINES A

Table A-1. SRM Cube Command Summary (1 of 2)

Command Invocation	Description
archcube [**-c** *cubename*]	Displays architecture type of the cube. (i860™ or Intel386™)
attachcube [**-c** *cubename*]	Make the default or specified *cubename* the currently attached cube.
bootcube [**-B** *file*] [**-b** *file*] [**-C**] [**-D***n* | **-D***m-n*] [**-d** *dim* | **-n** *nodes*] [**-f**] [**-K** *file*] [**-k** *file*] [**-L**] [**-l**] [**-o**] [**-Q** *file*] [**-R** *file*] [**-r** *file*] [**-S** *file*] [**-s** *file*] [**-T** *file*] [**-U** *file*]	Load NX/2 operating system onto nodes, and start iPSC daemons. (System Administrator only)
cubeinfo [**-a** | **-s** | **-n** | **-h** *srmname*]	Displays cube ownership information.
fsplit [*file*]	Splits a multi-routine Fortran file into individual files
getcube [**-c** *cubename*] [**-t** *cubetype*] [**-h** *srmname*]	Allocate a cube, and make it the currently attached cube.
killcube [**-c** *cubename*] [**-p** *pid*] [*node...*]	Kill node process(es).
load [**-c** *cubename*] [**-p** *pid*] [**-H**] [*node...*] *filename* [*arguments...*]	Loads a user process into the cube.
mkcfs	Construct a file system. (System Administrator only)

Table A-1. SRM Cube Command Summary (2 of 2)

Command Invocation	Description
newserver [**-c** *cubename*]	Start a new file server for the specified cube.
nsh [*arguments*]	Load C-shell onto node 0.
plogoff	Stop logging process-related information. (System Administrator only)
plogon [**-a**] [**-c**] [*pathname*]	Start logging process-related information. (System Administrator only)
rar [**-h** *srmname*] [*ar_options*] *filename...* **ras** [**-h** *srmname*] [*as_options*] *filename...* **rcc** [**-cpp**] [**-h** *srmname*] [*cc_options*] *filename...* **rf77** [**-cpp**] [**-h** *srmname*] [*f77_options*] *filename...* **rld** [**-h** *srmname*] [*ld_options*] *filename...*	Librarian, assembler, C compiler, Fortran compiler, and linker. These remote development utilities let the remote host build node-executable programs. (Available only on remote host.)
rebootcube	Load NX/2 operating system onto nodes, and start iPSC daemons. Same as a **bootcube** with defaults, but does not require root privilege.
relcube [**-c** *cubename* | **-a** | **-f** *cubeid* | **-F**]	Release one or more cubes.
showvol	Display available CFS disk drives.
startcube [**-c** *cubename*] [**-p** *pid*] [*node...*]	Start a process executing on the cube.
syslog [**-e**]	Send output of host process to file server handling I/O from nodes.
waitcube [**-c** *cubename*] [**-p** *pid*] [**-f**] [**-i**] [**-s**] [*node...*]	Wait for process(es) on node(s) to finish before proceeding.

Table A-2. Summary of System Calls for Cube Control (C Version) (1 of 2)

Call	Environment	Synopsis	Description
attachcube()	Host	**attachcube**(*cubename*) char **cubename*;	Attach to a cube and make it the current cube.
cubeinfo()	Host	long **cubeinfo**(*ct*, *numslots*, *global*) struct cubetable **ct*; long *numslots*, *global*;	Obtain information about allocated cubes.
flick()	Host, Node	**flick**()	On a node, relinquish the CPU to another process. On the host, is a no-op.
getcube()	Host	**getcube**(*cubename*, *cubetype*, *srmname*, *keep*) char **cubename*, **cubetype*, **srmname*; long *keep*;	Allocate a cube.
handler()	Node	**handler**(*etype*, *proc*) long *etype*; void(**proc*)();	Provide user-written exception handler for program failures.
killcube()	Host, Node	**killcube**(*node*, *pid*) long *node*, *pid*;	Terminate and clear node process(es).
killproc()	Host, Node	**killproc**(*node*, *pid*) long *node*, *pid*;	Terminate a node process.
killsyslog()	Host	**killsyslog**()	Terminate *syslog* process.
load()	Host, Node	**load**(*filename*, *node*, *pid*) char **filename*; long *node*, *pid*;	Load a node process.
myhost()	Host, Node	long **myhost**()	Obtain node ID of host machine.
mynode()	Host, Node	long **mynode**()	Obtain node ID of calling process.
mypid()	Host, Node	long **mypid**()	Obtain NX/2 process ID of calling process.

Table A-2. Summary of System Calls for Cube Control (C Version) (2 of 2)

Call	Environment	Synopsis	Description
newserver()	Host	**newserver**(*cubename*) char **cubename*;	Start new file server for specified cube.
nodedim()	Host, Node	long **nodedim()**	Obtain dimension of current cube.
numnodes()	Host, Node	long **numnodes()**	Obtain number of nodes in current cube.
plogoff()	Host	**plogoff()**	Stop logging process-related information.
plogon()	Host	**plogon**(*append, alloconly, logfile*) short *append, alloconly*; char **logfile*;	Start logging process-related information.
relcube()	Host	**relcube**(*cubename*) char **cubename*;	Release specified cube.
setpid()	Host	**setpid**(*pid*) long *pid*;	Set NX/2 process ID for host program.
setsyslog()	Host	**setsyslog**(*stdfd*) long *stdfd*;	Start *syslog* process.
waitall()	Host, Node	**waitall**(*node, pid*) long *node, pid*;	Wait for all specified processes to complete.
waitone()	Host, Node	**waitone**(*node, pid, cnode, cpid, ccode*) long *node, pid*; long **cnode, *cpid, *ccode*;	Wait for specified processes to complete.

Table A-3. Summary of Routines for Cube Control (Fortran Version) (1 of 2)

Routine	Environment	Synopsis	Description
ATTACHCUBE()	Host	SUBROUTINE **ATTACHCUBE**(*cubename*) CHARACTER *cubename**(*)	Attach to a cube and make it the current cube.
CUBEINFO()	Host	INTEGER FUNCTION **CUBEINFO**(*ct, numslots, global*) CHARACTER*16 *ct*(*slotsize, numslots*) INTEGER *numslots, global*	Obtain information about allocated cubes.
FLICK()	Host, Node	SUBROUTINE **FLICK()**	On a node: Relinquish the CPU to another process. On the host: A no-op.
GETCUBE()	Host	SUBROUTINE **GETCUBE**(*cubename, cubetype, srmname, keep*) CHARACTER *cubename**(*) CHARACTER *cubetype**(*) CHARACTER *srmname**(*) INTEGER *keep*	Allocate a cube.
HANDLER()	Node	SUBROUTINE **HANDLER**(*etype, proc*) INTEGER *etype* EXTERNAL *proc* *proc* must be written in C. It has one argument: the hardware exception code.	Provide user-written exception handler for program failures.
KILLCUBE()	Host, Node	SUBROUTINE **KILLCUBE**(*node, pid*) INTEGER *node, pid*	Terminate and clear node process(es).
KILLPROC()	Host, Node	SUBROUTINE **KILLPROC**(*node, pid*) INTEGER *node, pid*	Terminate a node process.
KILLSYSLOG()	Host	SUBROUTINE **KILLSYSLOG()**	Terminate *syslog* process.
LOAD()	Host, Node	SUBROUTINE **LOAD**(*filename, node, pid*) CHARACTER *filename**(*) INTEGER *node, pid*	Load a node process.

Table A-3. Summary of Routines for Cube Control (Fortran Version) (2 of 2)

Routine	Environment	Synopsis	Description
MYHOST()	Host, Node	INTEGER FUNCTION **MYHOST()**	Obtain node ID of host machine.
MYNODE()	Host, Node	INTEGER FUNCTION **MYNODE()**	Obtain node ID of calling process.
MYPID()	Host, Node	INTEGER FUNCTION **MYPID()**	Obtain NX/2 process ID of calling process.
NEWSERVER()	Host	SUBROUTINE **NEWSERVER**(*cubename*) CHARACTER *cubename**(*)	Start new file server for specified cube.
NODEDIM()	Host, Node	INTEGER FUNCTION **NODEDIM()**	Obtain dimension of current cube.
NUMNODES()	Host, Node	INTEGER FUNCTION **NUMNODES()**	Obtain number of nodes in current cube.
PLOGOFF()	Host	SUBROUTINE **PLOGOFF()**	Stop logging process-related information.
PLOGON()	Host	SUBROUTINE **PLOGON**(*append, alloconly, logfile*) INTEGER *append, alloconly* CHARACTER *logfile**(*)	Start logging process-related information.
RELCUBE()	Host	SUBROUTINE **RELCUBE**(*cubename*) CHARACTER *cubename**(*)	Release specified cube.
SETPID()	Host	SUBROUTINE **SETPID**(*pid*) INTEGER *pid*	Set NX/2 process ID for host program.
SETSYSLOG()	Host	SUBROUTINE **SETSYSLOG**(*stdfd*) INTEGER *stdfd*	Start *syslog* process.
WAITALL()	Host, Node	SUBROUTINE **WAITALL**(*node, pid*) INTEGER *node, pid*	Wait for all specified processes to complete.
WAITONE()	Host, Node	SUBROUTINE **WAITONE**(*node, pid, cnode, cpid, ccode*) INTEGER *node, pid* INTEGER *cnode, cpid, ccode*	Wait for specified processes to complete.

Table A-4. Summary of System Calls for Message Passing (C Version) (1 of 3)

Call	Environment	Synopsis	Description
cprobe()	Host, Node	**cprobe**(*typesel*) long *typesel*;	Wait for a message to arrive.
crecv()	Host, Node	**crecv**(*typesel*, *buf*, *len*) long *typesel*; char **buf*; long *len*;	Receive a message, and wait for completion.
csend()	Host, Node	**csend**(*type*, *buf*, *len*, *node*, *pid*) long *type*; char **buf*; long *len*, *node*, *pid*;	Send a message, and wait for completion.
csendrecv()	Host, Node	long **csendrecv**(*type*, *sbuf*, *slen*, *tonode*, *topid*, *typesel*, *rbuf*, *rlen*) long *type*; char **sbuf*; long *slen*, *tonode*, *topid*, *typesel*; char **rbuf*; long *rlen*;	Simultaneously, send a message, and post a receive for the reply. Wait for completion.
flushmsg()	Host, Node	**flushmsg**(*typesel*, *node*, *pid*) long *typesel*, *node*, *pid*;	Flush specified messages from the system.
hrecv()	Node	**hrecv**(*typesel*, *buf*, *len*, *proc*) long *typesel*; char **buf*; long *len*; void(**proc*)(); proc(*type*, *count*, *node*, *pid*) long *type*, *count*, *node*, *pid*;	Provide user-written exception handler for receive traps.
hsend()	Node	**hsend**(*type*, *buf*, *len*, *node*, *pid*, *proc*) long *type*; char **buf*; long *len*, *node*, *pid*; void(**proc*)(); proc(*type*, *count*, *node*, *pid*) long *type*, *count*, *node*, *pid*;	Send a message, and set up a handler procedure to be called when the reply arrives.

Table A-4. Summary of System Calls for Message Passing (C Version) (2 of 3)

Call	Environment	Synopsis	Description
hsendrecv()	Node	**hsendrecv**(*type*, *sbuf*, *slen*, *tonode*, *topid*, *typesel*, *rbuf*, *rlen*, *proc*) long *type*; char **sbuf*; long *slen*, *tonode*, *topid*, *typesel*; char **rbuf*; long *rlen*; void(**proc*)(); proc(*type*, *count*, *node*, *pid*) long *type*, *count*, *node*, *pid*;	Simultaneously, send a message, and post a receive for the reply. Also, set up a handler procedure to be called when the reply arrives.
infocount() **infonode()** **infopid()** **infotype()**	Host, Node	long **infocount()** long **infonode()** long **infopid()** long **infotype()**	Return information about a pending or received message.
iprobe()	Host, Node	long **iprobe**(*typesel*) long *typesel*;	Determine whether a message of a selected type is pending.
irecv()	Host, Node	long **irecv**(*typesel*, *buf*, *len*) long *typesel*; char **buf*; long *len*;	Receive a message.
isend()	Host, Node	long **isend**(*type*, *buf*, *len*, *node*, *pid*) long *type*; char **buf*; long *len*, *node*, *pid*;	Send a message.
isendrecv()	Host, Node	long **isendrecv**(*type*, *sbuf*, *slen*, *tonode*, *topid*, *typesel*, *rbuf*, *rlen*) long *type*; char **sbuf*; long *slen*, *tonode*, *topid*, *typesel*; char **rbuf*; long *rlen*;	Simultaneously, send a message, and post a receive for the reply. Do not wait for completion.
masktrap()	Node	long **masktrap**(*state*) long *state*;	Enable or disable receive trap.
msgcancel()	Host, Node	**msgcancel**(*id*) long *id*;	Cancel a send or receive operation.

Table A-4. Summary of System Calls for Message Passing (C Version) (3 of 3)

Call	Environment	Synopsis	Description
msgdone()	Host, Node	long **msgdone**(*id*) long *id*;	Determine whether a send or receive operation has completed.
msgwait()	Host, Node	**msgwait**(*id*) long *id*;	Wait for completion of a send or receive operation.

Table A-5. Summary of Routines for Message Passing (Fortran Version) (1 of 3)

Routine	Environment	Synopsis	Description
CPROBE()	Host, Node	SUBROUTINE **CPROBE**(*typesel*) INTEGER *typesel*	Wait for a message to arrive.
CRECV()	Host, Node	SUBROUTINE **CRECV**(*typesel, buf, len*) INTEGER *typesel* INTEGER *buf*(*) INTEGER *len*	Receive a message, and wait for completion.
CSEND()	Host, Node	SUBROUTINE **CSEND**(*type, buf, len,node, pid*) INTEGER *type* INTEGER *buf*(*) INTEGER *len, node, pid*	Send a message, and wait for completion.
CSENDRECV()	Host, Node	INTEGER FUNCTION **CSENDRECV**(*type, sbuf, slen, tonode, topid, typesel, rbuf, rlen*) INTEGER *type* INTEGER *sbuf*(*) INTEGER *slen, tonode, topid, typesel* INTEGER *rbuf*(*) INTEGER *rlen*	Simultaneously, send a message, and post a receive for the reply. Wait for completion.
FLUSHMSG()	Host, Node	SUBROUTINE **FLUSHMSG**(*typesel, node, pid*) INTEGER *typesel, node, pid*	Flush specified messages from the system.
HRECV()	Node	SUBROUTINE **HRECV**(*typesel, buf, len, proc*) INTEGER *typesel* INTEGER *buf*(*) INTEGER *len* EXTERNAL *proc* *proc* must be written in C: proc(*type, count, node, pid*) long *type, count, node, pid*;	Provide user-written exception handler for receive traps.

Table A-5. Summary of Routines for Message Passing (Fortran Version) (2 of 3)

Routine	Environment	Synopsis	Description
HSEND()	Node	SUBROUTINE **HSEND**(*type, buf, len, node, pid, proc*) INTEGER *type* INTEGER *buf*(*) INTEGER *len, node, pid* EXTERNAL *proc* *proc* must be written in C: proc(*type, count, node, pid*) long *type, count, node, pid*;	Send a message, and set up a handler procedure to be called when the reply arrives.
HSENDRECV()	Node	SUBROUTINE **HSENDRECV**(*type, sbuf, slen, tonode, topid, typesel, rbuf, rlen, proc*) INTEGER *type* INTEGER *sbuf*(*) INTEGER *slen, tonode, topid, typesel* INTEGER *rbuf*(*) INTEGER *rlen* EXTERNAL *proc* *proc* must be written in C: proc(*type, count, node, pid*) long *type, count, node, pid*;	Simultaneously, send a message, and post a receive for the reply. Also, set up a handler procedure to be called when the reply arrives.
INFOCOUNT() **INFONODE()** **INFOPID()** **INFOTYPE()**	Host, Node	INTEGER FUNCTION **INFOCOUNT()** INTEGER FUNCTION **INFONODE()** INTEGER FUNCTION **INFOPID()** INTEGER FUNCTION **INFOTYPE()**	Return information about a pending or received message.
IPROBE()	Host, Node	INTEGER FUNCTION **IPROBE**(*typesel*) INTEGER *typesel*	Determine whether a message of a selected type is pending.
IRECV()	Host, Node	INTEGER FUNCTION **IRECV**(*typesel, buf, len*) INTEGER *typesel* INTEGER *buf*(*) INTEGER *len*	Receive a message.
ISEND()	Host, Node	INTEGER FUNCTION **ISEND**(*type, buf, len, node, pid*) INTEGER *type* INTEGER *buf*(*) INTEGER *len, node, pid*	Send a message.

Table A-5. Summary of Routines for Message Passing (Fortran Version) (3 of 3)

Routine	Environment	Synopsis	Description
ISENDRECV()	Host, Node	INTEGER FUNCTION **ISENDRECV**(*type, sbuf, slen, tonode,* *topid, typesel, rbuf, rlen*) INTEGER *type* INTEGER *sbuf*(*) INTEGER *slen, tonode, topid, typesel* INTEGER *rbuf*(*) INTEGER *rlen*	Simultaneously, send a message, and post a receive for the reply. Do not wait for completion.
MASKTRAP()	Node	INTEGER FUNCTION **MASKTRAP**(*state*) INTEGER *state*	Enable or disable a receive trap.
MSGCANCEL()	Host, Node	SUBROUTINE **MSGCANCEL**(*id*) INTEGER *id*	Cancel a send or receive operation.
MSGDONE()	Host, Node	INTEGER FUNCTION **MSGDONE**(*id*) INTEGER *id*	Determine whether a send or receive operation has completed.
MSGWAIT()	Host, Node	SUBROUTINE **MSGWAIT**(*id*) INTEGER *id*	Wait for completion of a send or receive operation.

Table A-6. Global Operations (C Version) (1 of 3)

Call	Environment	Synopsis	Description
gcol()	Node	**gcol**(*x*, *xlen*, *y*, *ylen*, *ncnt*) char *x*[]; long *xlen*; char *y*[]; long *ylen*; long **ncnt*;	Global concatenation operation.
gcolx()	Node	**gcolx**(*x*, *xlens*, *y*) char *x*[]; long *xlens*; char *y*[];	Global concatenation operation for contributions of known length.
gdhigh()	Node	**gdhigh**(*x*, *n*, *work*) double *x*[]; long *n*; double *work*[];	Global vector double precision MAX operation.
gdlow()	Node	**gdlow**(*x*, *n*, *work*) double *x*[]; long *n*; double *work*[];	Global vector double precision MIN operation.
gdprod()	Node	**gdprod**(*x*, *n*, *work*) double *x*[]; long *n*; double *work*[];	Global vector double precision MULTIPLY operation.
gdsum()	Node	**gdsum**(*x*, *n*, *work*) double *x*[]; long *n*; double *work*[];	Global vector double precision SUM operation.
giand()	Node	**giand**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Global vector integer bitwise AND operation.
gihigh()	Node	**gihigh**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Global vector integer MAX operation.

Table A-6. Global Operations (C Version) (2 of 3)

Call	Environment	Synopsis	Description
gilow()	Node	**gilow**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Global vector integer MIN operation.
gior()	Node	**gior**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Global vector integer bitwise OR operation.
giprod()	Node	**giprod**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Global vector integer MULTIPLY operation.
gisum()	Node	**gisum**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Global vector integer SUM operation.
gixor()	Node	**gixor**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Global vector integer bitwise exclusive OR operation.
gland()	Node	**gland**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Global vector logical AND operation.
glor()	Node	**glor**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Global vector logical inclusive OR operation.
glxor()	Node	**glxor**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Global vector logical exclusive OR operation.

Table A-6. Global Operations (C Version) (3 of 3)

Call	Environment	Synopsis	Description
gopf()	Node	**gopf**(*x, xlen, work, f*) char *x*[]; long *xlen*; char *work*[]; long (**f*)();	Arbitrary commutative function.
gsendx()	Node	**gsendx**(*type, x, xlen, nodenums, nlen*) long *type*; char *x*[]; long *xlen*; long *nodenums*[]; long *nlen*;	Send a vector to a list of nodes.
gshigh()	Node	**gshigh**(*x, n, work*) float *x*[]; long *n*; float *work*[];	Global vector real MAX operation.
gslow()	Node	**gslow**(*x, n, work*) float *x*[]; long *n*; float *work*[];	Global vector real MIN operation.
gsprod()	Node	**gsprod**(*x, n, work*) float *x*[]; long *n*; float *work*[];	Global vector real MULTIPLY operation.
gssum()	Node	**gssum**(*x, n, work*) float *x*[]; long *n*; float *work*[];	Global vector real SUM operation.
gsync()	Node	**gsync**()	Global synchronization operation.

Table A-7. Global Operations (Fortran Version) (1 of 3)

Routine	Environment	Synopsis	Description
GCOL()	Node	SUBROUTINE **GCOL**(*x, xlen, y, ylen, ncnt*) INTEGER *x*(*) INTEGER *xlen* INTEGER *y*(*) INTEGER *ylen* INTEGER *ncnt*	Global concatenation operation.
GCOLX()	Node	SUBROUTINE **GCOLX**(*x, xlens, y*) INTEGER *x*(*) INTEGER *xlen* INTEGER *y*(*)	Global concatenation operation for contributions of known length.
GDHIGH()	Node	SUBROUTINE **GDHIGH**(*x, n, work*) DOUBLE PRECISION *x*(*) INTEGER *n* DOUBLE PRECISION *work*(*)	Global vector double precision MAX operation.
GDLOW()	Node	SUBROUTINE **GDLOW**(*x, n, work*) DOUBLE PRECISION *x*(*) INTEGER *n* DOUBLE PRECISION *work*(*)	Global vector double precision MIN operation.
GDPROD()	Node	SUBROUTINE **GDPROD**(*x, n, work*) DOUBLE PRECISION *x*(*) INTEGER *n* DOUBLE PRECISION *work*(*)	Global vector double precision MULTIPLY operation.
GDSUM()	Node	SUBROUTINE **GDSUM**(*x, n, work*) DOUBLE PRECISION *x*(*) INTEGER *n* DOUBLE PRECISION *work*(*)	Global vector double precision SUM operation.
GIAND()	Node	SUBROUTINE **GIAND**(*x, n, work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Global vector integer bitwise AND operation.
GIHIGH()	Node	SUBROUTINE **GIHIGH**(*x, n, work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Global vector integer MAX operation.

Table A-7. Global Operations (Fortran Version) (2 of 3)

Call	Environment	Synopsis	Description
GILOW()	Node	SUBROUTINE **GILOW**(*x*, *n*, *work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Global vector integer MIN operation.
GIOR()	Node	SUBROUTINE **GIOR**(*x*, *n*, *work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Global vector integer bitwise OR operation.
GIPROD()	Node	SUBROUTINE **GIPROD**(*x*, *n*, *work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Global vector integer MULTIPLY operation.
GISUM()	Node	SUBROUTINE **GISUM**(*x*, *n*, *work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Global vector integer SUM operation.
GIXOR()	Node	SUBROUTINE **GIXOR**(*x*, *n*, *work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Global vector integer bitwise exclusive OR operation.
GLAND()	Node	SUBROUTINE **GLAND**(*x*, *n*, *work*) LOGICAL *x*(*) INTEGER *n* LOGICAL *work*(*)	Global vector logical AND operation.
GLOR()	Node	SUBROUTINE **GLOR**(*x*, *n*, *work*) LOGICAL *x*(*) INTEGER *n* LOGICAL *work*(*)	Global vector logical inclusive OR operation.
GLXOR()	Node	SUBROUTINE **GLXOR**(*x*, *n*, *work*) LOGICAL *x*(*) INTEGER *n* LOGICAL *work*(*)	Global vector logical exclusive OR operation.

Table A-7. Global Operations (Fortran Version) (3 of 3)

Call	Environment	Synopsis	Description
GOPF()	Node	SUBROUTINE **GOPF**(*x, xlen, work, f*) INTEGER *x*(*) INTEGER *xlen* INTEGER *work*(*) EXTERNAL *f*	Arbitrary commutative function.
GSENDX()	Node	SUBROUTINE **GSENDX**(*type, x, xlen, nodenums, nlen*) INTEGER *type* INTEGER *x*(*) INTEGER *xlen* INTEGER *nodenums*(*) INTEGER *nlen*	Send a vector to a list of nodes.
GSHIGH()	Node	SUBROUTINE **GSHIGH**(*x, n, work*) REAL *x*(*) INTEGER *n* REAL *work*(*)	Global vector real MAX operation.
GSLOW()	Node	SUBROUTINE **GSLOW**(*x, n, work*) REAL *x*(*) INTEGER *n* REAL *work*(*)	Global vector real MIN operation.
GSPROD()	Node	SUBROUTINE **GSPROD**(*x, n, work*) REAL *x*(*) INTEGER *n* REAL *work*(*)	Global vector real MULTIPLY operation.
GSSUM()	Node	SUBROUTINE **GSSUM**(*x, n, work*) REAL *x*(*) INTEGER *n* REAL *work*(*)	Global vector real SUM operation.
GSYNC()	Node	SUBROUTINE **GSYNC**()	Global synchronization operation.

Table A-8. Byte-Swapping System Calls for Remote Host/Cube Communication (C Version)

Call	Environment	Synopsis	Description
createstruc()	Host	**createstruc(***ptr***)** char ***ptr**;	Initialize array used when byte-swapping structures.
CTOHC()	Host	**CTOHC(***sv, n***)** unsigned char ***sv**; long *n*;	Used when byte-swapping structures with chars, cube to host.
CTOHD()	Host	**CTOHD(***sv, n***)** double ***sv**; long *n*;	Byte-swap cube to host double.
CTOHF()	Host	**CTOHF(***sv, n***)** float ***sv**; long *n*;	Byte-swap cube to host float.
CTOHL()	Host	**CTOHL(***sv, n***)** long ***sv**; long *n*;	Byte-swap cube to host long.
CTOHS()	Host	**CTOHS(***sv, n***)** short ***sv**; long *n*;	Byte-swap cube to host short.
HTOCC()	Host	**HTOCC(***sv, n***)** unsigned char ***sv**; long *n*;	Used when byte-swapping structures with chars, cube to host.
HTOCD()	Host	**HTOCD(***sv, n***)** double ***sv**; long *n*;	Byte-swap host to cube double.
HTOCF()	Host	**HTOCF(***sv, n***)** float ***sv**; long *n*;	Byte-swap host to cube float.
HTOCL()	Host	**HTOCL(***sv, n***)** long ***sv**; long *n*;	Byte-swap host to cube long.
HTOCS()	Host	**HTOCS(***sv, n***)** short ***sv**; long *n*;	Byte-swap host to cube short.
relstruc()	Host	unsigned long **relstruc()**	Release array used when byte-swapping structures.

Table A-9. Byte-Swapping Routines for Remote Host/Cube Communication (Fortran Version)

Routine	Environment	Synopsis	Description
CTOHD()	Host	SUBROUTINE **CTOHD**(*sv*, *n*) DOUBLE PRECISION *sv*(*) INTEGER *n*	Byte-swap cube to host double precision.
CTOHF()	Host	SUBROUTINE **CTOHF**(*sv*, *n*) REAL *sv*(*) INTEGER *n*	Byte-swap cube to host real.
CTOHL()	Host	SUBROUTINE **CTOHL**(*sv*, *n*) INTEGER*4 *sv*(*) INTEGER *n*	Byte-swap cube to host integer.
CTOHS()	Host	SUBROUTINE **CTOHS**(*sv*, *n*) INTEGER*2 *sv*(*) INTEGER *n*	Byte-swap cube to host short integer.
HTOCD()	Host	SUBROUTINE **HTOCD**(*sv*, *n*) DOUBLE PRECISION *sv*(*) INTEGER *n*	Byte-swap host to cube double precision.
HTOCF()	Host	SUBROUTINE **HTOCF**(*sv*, *n*) REAL *sv*(*) INTEGER *n*	Byte-swap host to cube real.
HTOCL()	Host	SUBROUTINE **HTOCL**(*sv*, *n*) INTEGER*4 *sv*(*) INTEGER *n*	Byte-swap host to cube integer.
HTOCS()	Host	SUBROUTINE **HTOCS**(*sv*, *n*) INTEGER*2 *sv*(*) INTEGER *n*	Byte-swap host to cube short integer.

Table A-10. Miscellaneous System Calls (C Version)

Call	Environment	Synopsis	Description
dclock()	Node	double **dclock()**	Returns time in seconds since booting the cube.
ginv()	Host, Node	long **ginv**(*j*) long *j*;	Return the position of an element in the binary-reflected gray code sequence. Inverse of **gray()**.
gray()	Host, Node	long **gray**(*j*) long *j*;	Return the binary-reflected gray code for an integer.
hwclock()	Node	void **hwclock**(*hwtime*) esize_t **hwtime*;	Places the current value of the hardware counter in the 64-bit unsigned integer.
led()	Node	void **led**(*lstate*) long *lstate*;	Turn the node's green LED on or off.
mclock()	Host, Node	unsigned long **mclock()**	Return the time (in milliseconds).

Table A-11. Miscellaneous Routines (Fortran Version)

Routine	Environment	Synopsis	Description
DCLOCK()	Node	DOUBLE PRECISION **DCLOCK()**	Returns time in seconds since booting the cube.
GINV()	Host, Node	INTEGER FUNCTION **GINV**(*j*) INTEGER *j*	Return the position of an element in the binary-reflected gray code sequence. Inverse of **GRAY()**.
GRAY()	Host, Node	INTEGER FUNCTION **GRAY**(*j*) INTEGER *j*	Return the binary-reflected gray code for an integer.
HWCLOCK()	Node	SUBROUTINE **HWCLOCK**(*hwtime*) INTEGER *hwtime*(2)	Places the current value of the hardware counter in the 64-bit unsigned integer.
LED()	Node	SUBROUTINE **LED**(*lstate*) INTEGER *lstate*	Turn the node's green LED on or off.
MCLOCK()	Node	INTEGER FUNCTION **MCLOCK()**	Return elapsed time (in milliseconds).

Table A-12. Summary of System Calls for Concurrent File I/O (C Version) (1 of 2)

Call	Environment	Synopsis	Description
cread()	Node	**cread**(*fildes*, *buf*, *len*) long *fildes*; char **buf*; long *len*;	Read from a file, and wait for completion.
cwrite()	Node	**cwrite**(*fildes*, *buf*, *len*) long *fildes*; char **buf*; long *len*;	Write to a file, and wait for completion.
eseek()	Node	esize_t **eseek**(*fildes*, *offset*, *whence*) long *fildes*; esize_t *offset*; long *whence*;	Move file pointer in extended file.
esize()	Node	esize_t **esize**(*fildes*, *offset*, *whence*) long *fildes*; esize_t *offset*; long *whence*;	Increase size of extended file.
estat()	Node	long **estat**(*path*, *buf*) char **path*; struct estat **buf*;	Get status of extended file.
festat()	Node	long **festat**(*fildes*, *buf*) long *fildes*; struct estat **buf*;	Get status of extended file from file descriptor.
iodone()	Node	long **iodone**(*id*) long *id*;	Determine whether an asynchronous I/O read or write operation is complete.
iomode()	Node	long **iomode**(*fildes*) long *fildes*;	Return the current I/O mode for a file.
iowait()	Node	**iowait**(*id*) long *id*;	Wait for completion of an asynchronous I/O read or write operation.
iread()	Node	long **iread**(*fildes*, *buf*, *len*) long *fildes*; char **buf*; long *len*;	Asynchronous read from a file. (Do not wait for completion.)

Table A-12. Summary of System Calls for Concurrent File I/O (C Version) (2 of 2)

Call	Environment	Synopsis	Description
iseof()	Node	long **iseof**(*fildes*) long *fildes*;	Test for end-of-file.
iwrite()	Node	long **iwrite**(*fildes*, *buf*, *len*) long *fildes*; char **buf*; long *len*;	Asynchronous write to a file. (Do not wait for completion.)
lsize()	Node	long **lsize**(*fildes*, *offset*, *whence*) long *fildes*, *offset*, *whence*;	Increase size of a file.
restrictvol()	Node	long **restrictvol**(*fildes*, *nvol*, *vollist*) long *fildes*, *nvol*, *vollist*[];	Restrict or show the disk volumes to which a file can be allocated.
setiomode()	Node	**setiomode**(*fildes*, *mode*) long *fildes*, *mode*;	Set the I/O mode for a file.

Table A-13. Summary of Routines for Concurrent File I/O (Fortran Version) (1 of 2)

Routine	Environment	Synopsis	Description
CREAD()	Node	SUBROUTINE **CREAD**(*unit*, *buf*, *len*) INTEGER *unit* INTEGER *buf*(*) INTEGER *len*	Read from a file, and wait for completion.
CWRITE()	Node	SUBROUTINE **CWRITE**(*unit*, *buf*, *len*) INTEGER *unit* INTEGER *buf*(*) INTEGER *len*	Write to a file, and wait for completion.
ESEEK()	Node	SUBROUTINE **ESEEK**(*unit*, *offset*, *whence*, *newpos*) INTEGER *unit* INTEGER *offset*(2) INTEGER *whence* INTEGER *newpos*(2)	Move file pointer in large file.
ESIZE()	Node	SUBROUTINE **ESIZE**(*unit*, *offset*, *whence*, *newsize*) INTEGER *unit* INTEGER *offset*(2) INTEGER *whence* INTEGER *newsize*(2)	Increase size of large file.
FORCEFLUSH()	Node	SUBROUTINE **FORCEFLUSH()**	Flush all buffered I/O when an exception occurs.
FORFLUSH()	Node	SUBROUTINE **FORFLUSH**(*iunit*) INTEGER *iunit*	Complete all buffered I/O on the specified unit.
IODONE()	Node	INTEGER FUNCTION **IODONE**(*id*) INTEGER *id*	Determine whether an asynchronous I/O read or write operation is complete.
IOMODE()	Node	INTEGER FUNCTION **IOMODE**(*unit*) INTEGER *unit*	Return the current I/O mode for a file.
IOWAIT()	Node	SUBROUTINE **IOWAIT**(*id*) INTEGER *id*	Wait for completion of an asynchronous I/O read or write operation.

Table A-13. Summary of Routines for Concurrent File I/O (Fortran Version) (2 of 2)

Routine	Environment	Synopsis	Description
IREAD()	Node	INTEGER FUNCTION **IREAD**(*unit*, *buf*, *len*) INTEGER *unit* INTEGER *buf*(*) INTEGER *len*	Asynchronous read from a file. (Do not wait for completion.)
ISEOF()	Node	INTEGER FUNCTION **ISEOF**(*unit*) INTEGER *unit*	Test for end-of-file.
IWRITE()	Node	INTEGER FUNCTION **IWRITE**(*unit*, *buf*, *len*) INTEGER *unit* INTEGER *buf*(*) INTEGER *len*	Asynchronous write to a a file. (Do not wait for completion.)
LSEEK()	Node	INTEGER FUNCTION **LSEEK**(*unit*, *offset*, *whence*) INTEGER *unit* INTEGER *offset* INTEGER *whence*	Move the read/write file pointer.
LSIZE()	Node	INTEGER FUNCTION **LSIZE**(*unit*, *offset*, *whence*) INTEGER *unit* INTEGER *offset* INTEGER *whence*	Increase size of a file.
RESTRICTVOL()	Node	INTEGER FUNCTION **RESTRICTVOL**(*unit*, *nvol*, *vollist*) INTEGER *unit* INTEGER *nvol* INTEGER *vollist*(*)	Restrict or show the disk volumes to which a file can be allocated.
SETIOMODE()	Node	SUBROUTINE **SETIOMODE**(*unit*, *mode*) INTEGER *unit* INTEGER *mode*	Set the I/O mode for a file.

Table A-14. Summary of Mathematical System Calls (C Version)

Call	Environment	Synopsis	Description
eadd()	Node	esize_t **eadd**(*e1*, *e2*) esize_t *e1*; esize_t *e2*;	Add two extended numbers.
ecmp()	Node	long **ecmp**(*e1*, *e2*) esize_t *e1*; esize_t *e2*;	Compare two extended numbers.
ediv()	Node	long **ediv**(*e*, *n*) esize_t *e*; long *n*;	Divide extended number by integer.
emod()	Node	long **emod**(*e*, *n*) esize_t *e*; long *n*;	Display remainder of **ediv()**.
emul()	Node	esize_t **emul**(*e*, *n*) esize_t *e*; long *n*;	Multiply extended number by integer.
esub()	Node	esize_t **esub**(*e1*, *e2*) esize_t *e1*; esize_t *e2*;	Subtract two extended numbers.
etos()	Node	**etos**(*e*, *s*) esize_t *e*; char **s*;	Convert extended number to string.
stoe()	Node	esize_t **stoe**(*s*) char **s*;	Convert string to extended number.

Table A-15. Summary of Mathematical Routines (Fortran Version)

Routine	Environment	Synopsis	Description
EADD()	Node	SUBROUTINE **EADD**(*e1*, *e2*, *eresult*) INTEGER *e1*(2) INTEGER *e2*(2) INTEGER *eresult*(2)	Add two extended numbers.
ECMP()	Node	INTEGER FUNCTION **ECMP**(*e1*, *e2*) INTEGER *e1*(2) INTEGER *e2*(2)	Compare two extended numbers.
EDIV()	Node	SUBROUTINE **EDIV**(*e*, *n*, *result*) INTEGER *e*(2) INTEGER *n* INTEGER *result*	Divide extended number by integer.
EMOD()	Node	SUBROUTINE **EMOD**(*e*, *n*, *result*) INTEGER *e*(2) INTEGER *n* INTEGER *result*	Display remainder of **EDIV()**.
EMUL()	Node	SUBROUTINE **EMUL**(*e*, *n*, *eresult*) INTEGER *e*(2) INTEGER *n* INTEGER *eresult*(2)	Multiply extended number by integer.
ESUB()	Node	SUBROUTINE **ESUB**(*e1*, *e2*, *eresult*) INTEGER *e1*(2) INTEGER *e2*(2) INTEGER *eresult*(2)	Subtract two extended numbers.
ETOS()	Node	SUBROUTINE **ETOS**(*e*, *s*) INTEGER *e*(2) CHARACTER *s*(*)	Convert extended number to string.
FPSETMASK()	Node	SUBROUTINE **FPSETMASK**(*mask*) INTEGER *mask*	Set 387™ exception mask.
STOE()	Node	SUBROUTINE **STOE**(*s*, *e*) CHARACTER *s*(*) INTEGER *e*(2)	Convert string to extended number.

Table A-16. C-Shell Built-Ins and UNIX Utilities That Run Under the Node's C-Shell

C-Shell Built-Ins		UNIX Utilities	
alias	login	awk	ln
break	logout	cat	ls[1]
breaksw	newgrp	chgrp	lsize
case	nice	chmod	mkdir
cd	nohup	chown	mv
chdir	onintr	cmp	pr
continue	rehash	cp	ps
default	repeat	csh	pwd
echo	set	dd	rm
else	sentenv	df	rmdir
end	shift	diff	sed
endif	source	diffh	star
endsw	switch	du	stream
exec	time	egrep	sync
exit	umask	env	tail
foreach	unalias	fgrep	tar
glob	unhash	file	touch
goto	unset	grep	wc
hashstat	unsetenv	kill	
history	wait		
if	while		

1. Does not show . and ..

Table A-17. UNIX-Compatible System Calls (C Version)

access	execvp	lseek	sigpause
alarm	exit	mkdir	sigrelse
brk	fork	nice[2]	sigset
chdir	fstat	open	stat
chmod	fstatfs	pause	stafs
chown	getegid	pipe	sync
close	geteuid	read	sysi86[3]
creat	getgid	rmdir	time
dup	getpid	sbrk	times
excel	getuid	setgid	umask
execle	gtty	setuid	unlink
execlp	ioctl[1]	sighold	utime
execv	kill	sigignore	wait
execve	link	signal	write

1. Supports only the following requests on terminal devices:

   TCGETA
   TCSETA
   TCSETAW
   TCSETAF
   TCSBRK
   RCXONC
   TCFLSH

2. No operation.

3. Supports only the SI86FPHW and the SI86MEM commands.

Table A-18. UNIX Compatible I/O Library Calls (C Only Version)

a641	fpgetsticky	getw	printf	strdup
abort	fprintf	gmtime	putchar	strlen
abs	fpsetmask	gsignal	putenv	strncat
asctime	fpsetround	hcreate	putpwent	strncmp
atof	fpsetsticky	hdestroy	puts	strncpy
atoi	fputc	hsearch	putw	strpbrk
atol	fputs	isatty	qsort	strrchr
bsearch	fread	isnand	rand	strspn
calloc	free	jrand48	readdir	strtod
clearerr	freopen	l3tol	realloc	strtok
closedir	frexp	l64a	rewind	strtol
crypt	fscanf	lcong48	rewinddir	swab
ctermid	fseek	ldexp	scanf	tdelete
ctime	ftell	lfind	seed48	telldir
cuserid	ftok	localtime	seekdir	tempnam
drand48	ftw	longjmp	setbuf	tfind
ecvt	fwrite	lrand48	setgrent	tmpfile
encrypt	gcvt	lsearch	setjmp	tmpnam
endgrent	getchar	ltol3	setkey	tolower
endpwent	getcwd	malloc	setpwent	toupper
erand48	getenv	memccpy	setvbuf	tsearch
fclose	getgrent	memchr	sleep	ttyname
fcvt	getgrgid	memcmp	sprintf	ttyslot
fdopen	getgrnam	memcpy	srand	twalk
fflush	getlogin	memset	srand48	tzset
fgetc	getopt	mktemp	sscanf	ungetc
fgetgrent	getpass	modf	ssignal	vfprintf
fgetpwent	getpw	mrand48	strcat	vprintf
fgets	getpwent	nlist	strchr	vsprintf
fopen	getpwnam	nrand48	strcmp	
fpgetmask	getpwuid	opendir	strcpy	
fpgetround	gets	perror	strcspn	

Table A-19. Summary of SRM UNIX Extensions

Command Invocation	Description
cptape [*-i* | *-o* | *-t*] [*-f devicename*] *filename...*	Provide interface to **cpio** command.
head [*-n*] [*filename...*]	Display first few lines of specified file(s).
less [**-dstwcCeEmMqQuU**] [**-h** *N*] [**-b[fp]** *N*] [**-x** *N*] [**-[z]** *N*] [**-P[mM]** *string*] [**-[lL]** *logfile*] [+ *cmd*] [*filename...*]	Similar to **more**, but with additional features.
star [-]c[wfb] *starfile blocksize filename...* **star [-]r[wb]** *starfile blocksize* [*filename...*] **star [-]t[f]** *starfile* **star [-]u[wb]** *starfile blocksize* [*filename...*] **star [-]x[lmopwf]** *starfile* [*filename...*]	Create new tape archive. Add to existing tape archive. List contents of tape archive. Update existing tape archive. Extract files from tape archive.
stream [*KBytes_of_buffer*] [*source*] [*destination*]	Transfer large amounts of data to or from tape.

Table A-20. Node's C-Shell Cube Command Summary

Command Invocation	Description
archcube [**-c** *cubename*]	Displays architecture type of the current cube (i860 or Intel386).
cbackup [**-s** *mediasize*] [**-a**] [*volume...*] [*filename*]	Back up CFS disk drive(s).
crestore [*volume...*] [*filename*]	Restore backed up CFS disk image.
killcube -p *pid* [*node...*]	Kill node process(es).
load -p *pid* [*node...*] *filename* [*arguments...*]	Loads a user process into the cube.
mkdev *volume devicename*	Allocates a file that represents a tape managed by CFS.
mt [**-f** *tapename*] *command* [*count*]	A magnetic tape manipulating program.
showvol	Display information about available drives.
tapemode [*switches*] *devicename...*	Changes operating mode for tape drives.
waitcube -p *pid* [**-f**] [**-i**] [*node...*]	Wait for process(es) on node(s) to finish before proceeding.

Table A-21. Unique Node TCP/IP System Calls

Command Invocation	Description
getiphosts()	Obtain a list of the names of available socket nodes.
setiphost(*iphost*)	Specify a particular socket node as the iphost. The system dynamically chooses a default iphost based on load balancing. This call gives you the ability to override the system's decision.

iPSC® SYSTEM TYPES and TYPESEL MASKS B

TYPES

The *type* parameter used in message passing calls is a user-defined variable typically used to identify the kind of information contained in the message being sent. 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 send-receive 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.

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` (See the section "Typesel Mask List"). For example, if you want to receive message types `1`, `2`, `5`, and `12`, add the following hex numbers:

```
0x2, 0x4, 0x20, 0x1000 + 0x80000000 = 0x80001026
```

then enter

```
crecv (0x80001026, buf, len);
```

Or, if you want to receive any message except type `0` use:

```
crecv (0xFFFFFFFE, buf, len);
```

Typesel Mask List

The following list gives the hexadecimal number associated with bits 0-31:

Type	Hex Number
0	0x00000001
1	0x00000002
2	0x00000004
3	0x00000008
4	0x00000010
5	0x00000020
6	0x00000040
7	0x00000080
8	0x00000100
9	0x00000200
10	0x00000400
11	0x00000800
12	0x00001000
13	0x00002000
14	0x00004000
15	0x00008000
16	0x00010000
17	0x00020000
18	0x00040000
19	0x00080000
20	0x00100000
21	0x00200000
22	0x00400000
23	0x00800000
24	0x01000000
25	0x02000000
26	0x04000000
27	0x08000000
28	0x10000000
29	0x20000000
Other types	0x40000000

ERROR HANDLING IN C C

There are two versions of the C system routines. The standard version terminates a process when an error occurs and sends a message to standard error describing the error. The error return version does not terminate a process when an error occurs but returns a non-negative value if successful and a `-1` if not. The error return version of the C system routines is identified by an underscore as the first character of the name. When the call returns a `-1`, or a failure, the global error variable *errno* is set and the UNIX *perror* routine may be used to print an error message on standard error. For example,

```
extern long errno;
main( )
{
          .
          .
long status;
          .
          .
    status = _getcube("alpha", "8", "myhost", 0);
    if (status < 0) {        /* Errno is set to error. See errno.h for error list. */
                                                    /* Perform any action you wish */
        perror( "getcube");                         /*Print error message, if desired*/
    }
          .
          .
}
```

The underscore version of the C system routines allows you to write your program so that it takes a certain action when an error occurs rather than terminate. For example, **crecv()** would terminate when an error occurred and send a message to the standard output describing the error. The **_crecv()** call would not terminate with an error, but return a `-1` indicating an error had occurred. In addition, **_crecv()** would set *errno* to an error code indicating what the error was. Then *perror* could be used to print an error message. A list of possible iPSC-specific errors follows. A complete list of system errors is located in */usr/include/sys/errno.h*. The iPSC error list is appended to the bottom of this file.

Table C-1. iPSC®/2 and iPSC®/860 Error Messages (1 of 5)

Code Name	Message	Corrective Action
EQDRVERR	`Internal driver error`	
EQPBUF	`Invalid buffer pointer`	Specify a pointer which contains the address of a valid data buffer.
EQBLEN	`Buffer length exceeds allocation`	Make sure length does not exceed buffer size.
EQLEN	`Invalid length`	Use a non-negative number or a length less than or equal to maximum message length.
EQTIME	`Time limit exceeded`	
EQMSGLONG	`Received message too long for buffer`	Make sure buffer is large enough to hold the message.
EQPID	`Invalid pid`	Pid must not be negative.
EQNODE	`Invalid node`	Use **numnodes()** to determine cube size and **myhost()** to determine host number.
EQTYPE	`Invalid type`	Use a non-negative number.
EQMID	`Invalid message id`	Use message id returned by **irecv()** or **isend()**.
EQHND	`Invalid handler type`	Select one of the types listed in the handler description.
EQNOPROC	`Out of process slots`	Use fewer processes
EQUSEPID	`PID already in use`	Select another pid.
EQNOACT	`No active process`	Use the pid of a loaded process.
EQBADFIL	`Invalid object file`	Specify a loadable file.
EQPARAM	`Invalid parameter`	

Table C-1. iPSC®/2 and iPSC®/860 Error Messages (2 of 5)

Code Name	Message	Corrective Action
EQPFIL	`Invalid file name pointer`	
EQPCNODE	`Invalid cnode pointer`	
EQPCPID	`Invalid cpid pointer`	
EQPCCODE	`Invalid ccode pointer`	
EQPRIV	`Privileged operation`	
EQMEM	`Not enough memory`	Simplify application program.
EQINVREC	`Invalid loader record`	Internal error, try again. You may have to **bootcube.**
EQMSG	`Invalid loader message`	Internal error, try again. You may have to **bootcube.**
EQNOMID	`Too many requests`	Use **msgwait()** for outstanding **isend()** or **irecv()** requests.
EQSET	`Pid already set`	Do not call **setpid()** again.
EQNOSET	`No pid defined`	Use **setpid()** to define process id.
EQCUBETABFULL	`Internal cube usage limit`	Comm server has run out of slots for subcube allocation. Try again later when cube is not so busy.
EQCUBEEXISTS	`Cubename already exists`	Use a different name.
EQCUBENOTEXIST	`Cubename does not exist`	Use **cubeinfo** to find out cube names.
EQCUBENOTATTCH	`There is no attached cube`	Use **cubeinfo** to find out which cubes youown.
EQUSOCK	`Comm server out of UNIX` sockets	

Table C-1. iPSC®/2 and iPSC®/860 Error Messages (3 of 5)

Code Name	Message	Corrective Action
EQNETSOCK	`Comm server out of net sockets`	
EQTTY	`Comm server out of ttys`	Limit of 10 cubes may be allocated. Try again later when the system is not as busy.
EQNOCUBES	`There are no cubes allocated`	Use **getcube** to allocate a cube.
EQCUBENAMELEN	`Cubename longer than 15 characters.`	
EQHOSTNAMELEN	`Hostname longer than 15 characters`	
EQTYPENAMELEN	`Cubetype longer than 15 characters`	
EQBADGLOBAL	`Global value invalid`	
EQRCSBUSY	`Remote comm server busy, try again later.`	
EQINCMPREAD	`Did not read header specified bytes`	
EQNOCUBE	`Cubetype not found`	Could not find a cube of the type you requested.
EQLIFEBUSY	`Internal cube usage limit`	
EQNOCOMMSER	`Commser not responding`	Internal System Resource Manager processerror. Cube management process are not responding. Try rebooting cube with **bootcube.**
EQUSM	`Invalid diagnostic channel usm id`	

Table C-1. iPSC®/2 and iPSC®/860 Error Messages (4 of 5)

Code Name	Message	Corrective Action
EQDIM	`Invalid dimension`	
EQMODE	`Invalid diagnostic channel mode`	
EQSTATUS	`Invalid diagnostic channel status`	
EQNOLL	`Lifeline not responding`	Internal System Resource Manager processerror. Cube management process are notresponding. Try rebooting cube with **bootcube**.
EQBADNODE	`Node not responding`	
EQUSEVX	`Vector extension is in use by another process`	
EQNOSRM	`No SRM that matched your request was found`	Remote development machine error only. Could not find an SRM with a cube of type requested.
EQMSGSHORT	`Received message too short for buffer`	
EQNOSYSLOG	`No active syslog exists`	
EQSYSLOG	`Active syslog exists`	
EQLOGFAIL	`Cannot start syslog process`	
EQBIGSYS	`System too large for loading`	
EQRRTTY	`Cannot read from a remote terminal`	Check connections.
EQFSERVFAIL	`Cannot start fserver process`	Reboot cube.
ECFPS	`Seek to different file pointers`	See **setiomode()**.

Table C-1. iPSC®/2 and iPSC®/860 Error Messages (5 of 5)

Code Name	Message	Corrective Action
ENFPS	`Different file pointers`	See **setiomode()**.
EMIXIO	`Mixed file operations`	In I/O mode 2 or 3, ensure that nodes are performing the same operation.
EIMODE	`Bad io mode number`	Use 0, 1, 2 or 3.
ESETIO	`File is not synchronized`	In I/O modes 2 and 3, all nodes must read or write synchronously.
ESRMIO	`Not supported on the SRM`	CFS file operation only.
ENOCFS	`No CFS available`	
EQCOMMBUF	`Commser out of buffers`	
EQESIZE	`Invalid size`	
EQNOVX	`Vector extension not present`	
ERDEOF	`Attempt to read past end-of-fil`	