August, 1988
Order Number: 311017-003

iPSC®/2 C PROGRAMMER'S

REFERENCE MANUAL

intel Corporation

Copyright © 1988 by Intel Scientific Computers, Beaverton, Oregon All rights reserved. No part of this work may be reproduced or copied in any form or by any means...graphic, electronic, or mechanical including photocopying, taping, or information storage and retrieval systems...without the express written consent of Intel Corporation. The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update or to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's software license, or as defined in ASPR 7-104.9(a)(9).

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products:

Above	iLBX	iSBC	OTP	UPI
BITBUS	Im	iSBX	PC BUBBLE	VLSiCEL
COMMputer	iMDDX	iSDM	Plug-A-Bubble	4-SITE
CREDIT	iMMX	iSXM	PROMPT	
Data Pipeline	Insite	KEPROM	Promware	
FASTPATH	int$_e$l	Library Manager	QueX	
GENIUS	int$_e$lBOS	MAP-NET	QUEST	
I^2ICE	Intelevision	MCS	Programming	
i	int$_e$ligent Identifier	Megachassis	Quick-Pulse	
im	int$_e$ligent Programming	MICROMAINFRAME	Ripplemode	
ICE	Intellec	MULTIBUS	RMX/80	
iCEL	Intellink	MULTICHANNEL	RUPI	
iCS	iOSP	MULTIMODULE	Seamless	
iDBP	iPDS	ONCE	SLD	
iDIS	iRMX	OpenNET	SugarCube	

EXOS is a trademark or equipment designator of Excelan, Inc.
UNIX is a trademark of AT&T
VAST-2 is a registered trademark of Pacific-Sierra Research Corp.
iPSC is a registered trademark of Intel Corporation

REV.	REVISION HISTORY	DATE
-001	Original issue	12/87
-002	Revision	03/88
-003	Revision	08/88

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.

CHAPTER 1 - INTRODUCTION

CHAPTER 2 - iPSC/2 COMMANDS

CHAPTER 3 - iPSC/2 C ROUTINES

CHAPTER 3 - iPSC/2 C ROUTINES (cont.)

TABLES

CHAPTER 1

INTRODUCTION

PURPOSE

The purpose of this manual is to provide information for all the C routines and commands for the iPSC/2. The commands and routines are presented in alphabetical order by chapter.

SCOPE

This manual assumes you have a basic understanding of how these commands and routines are used. If you do not, you should read the iPSC/2 User's Guide.

Also, there are on-line example programs that may help familiarize you with iPSC/2 software usage. You can find these in the "*/usr/ipsc/examples/c*" directory.

This manual does not cover UNIX tools; nor does it provide information about any of the languages available on the iPSC/2 system. Refer to the list of applicable documents for the titles of manuals covering these topics.

CONTENTS

This manual contains the chapters and appendices listed in Table 1-1.

Table 1-1
Manual Contents

Chap	Title	Contents
1	Introduction	Describes this manual and other applicable documents.
2	iPSC/2 Commands	These are UNIX-type commands that control program execution on the cube. They provide such functions as: obtaining access to the cube, loading node processes, killing processes, and waiting for processes to complete.
3	iPSC/2 C Routines	These are program calls for the two C system interface libraries: the host library for linking with host programs and the node library for linking with node programs. These routines allow you to access and release cubes, load processes, pass messages between node processes, and many other functions.
A	Error Handling	Provides a complete list of error messages and a description of error handling for the iPSC/2 C system routines.
B	Typesel Mask	Gives the procedure for building a *typesel* mask which is used to select messages.
C	C Communication Utilities	These are routines which provide higher level constructs for interprocess communication.

APPLICABLE DOCUMENTS

The iPSC/2 manual set is composed of UNIX and iPSC/2 manuals as well as some additional manuals applicable to programming on the iPSC/2. Refer to Table 1-2.

Table 1-2
Applicable Documents

Title	Description
iPSC/2 User's Guide	Gives an overview of the iPSC/2 hardware and software, describes the iPSC/2 programming environment, and provides an example.
UNIX System V User's Guide	Provides a general description of UNIX.
UNIX System V User's Reference Manual	Describes all the UNIX system user commands.
UNIX System V Programmer's Guide	Describes the UNIX system programming environment and provides detailed descriptions of 14 programming tools.
UNIX System V Programmer's Reference Manual	Contains descriptions of commands, system calls, subroutines, libraries, file formats, macro packages, and character set tables.
The C Programming Language Manual	Provides a general overview to using the C language. Includes tutorial and reference sections.
Greenhills C Language Reference Manual	Describes the C compiler for the 80386. It explains the additions to the basic C language that are supported. It also explains register and memory allocation strategies, program optimization, and porting programs.

CHAPTER 2

iPSC/2 COMMANDS

INTRODUCTION

This chapter documents all the iPSC/2 commands. These commands allow you to perform such tasks 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.

In addition, almost all of these tasks can be initiated from user "host" programs via the iPSC/2 routines (refer to Chapter 3).

The iPSC/2 commands can be found in the */usr/bin* library.

Table 2-1
Command Summary

Command Invocation	Description
attachcube [-c cubename]	Make the default or specified cubename the current attached cube.
cubeinfo [-a] [-s] [-n]	Return cube ownership information.
getcube [-c cubename] [-t cubetype] [-h srmname]	Allocate a cube and make it the current attached cube.
killcube [-c cubename] [-p pid] [node...]	Kill node processes.
load [-c cubename] [-p pid] [-H] [node..] filename [arguments . . .]	Load a user process into the cube.
newserver [-c cubename]	Start a new file server for the specified cube.
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*	Remote iPSC/2 development utilities which are: the C compiler, FORTRAN compiler, linker, assembler, and librarian.
relcube [-c cubename] [-a]	Release cube(s).
startcube [-c cubename] [-p pid] [node...]	Start a process executing on the cube.
syslog [-e]	Send the output of a host process to the file server handling I/O from the nodes.
waitcube [-c cubename] [-p pid] [-f] [node...]	Wait for process(es) on node(s) to complete.

ATTACHCUBE(1) iPSC/2 ATTACHCUBE(1)

Make the default or specified cubename the current attached cube.

Invocation Sequence

attachcube [-c cubename]

Description

-c cubename Allows you to attach to a specific, named cube. You name a cube when you invoke *getcube*. Valid names are any ASCII character string, 15 characters or less. *Cubenames* cannot start with a dash ('-') or a space. If this argument is not used, the default cubename, "defaultname", is used. 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 *cubename* to which you are attached. If you need to find out the names of the cubes that you have allocated, use the *cubeinfo* command.

You can attach to only one cube at a time. If multiple names are specified, an error is returned.

Attachcube is used when you have several cube partitions and you need to attach to one other than the one you are currently attached to. In the course of a session, you may allocate several cubes using the *getcube* command. Let's say that these three commands were invoked:

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

The current attached cube is the last cube allocated (*beta*). To change the current cube back to the default *cubename*, type *attachcube*. Compare this command to the *attachcube* routine. The command changes the shell's current cube. The routine changes only a host process's current cube; the shell is unaffected.

Usage Examples

Attach to the default cubename of "defaultname"	`attachcube`
Attach to a cube named "alpha"	`attachcube -c alpha`

Errors

attachcube: cubename does not exist	Use *cubeinfo* to determine cube names.
attachcube: internal cube usage limit	Limit of 10 cubes may be allocated. Try again later when system is not so busy.
attachcube: commser not responding	Internal system resource manager process error. Cube management processes are not responding. Try rebooting cube with *bootcube*.
attachcube: lifeline not responding	Internal system resource manager process error. Cube management processes are not responding. Try rebooting cube with *bootcube*.

See Also

cubeinfo, getcube, relcube

CUBEINFO(1) iPSC/2 CUBEINFO(1)

Returns cube ownership information.

Invocation Sequence

cubeinfo [-a] [-s] [-n]

Description

-a	Allows you to get information about all the cubes that you own on the system from which you invoked the command.
-s	Gets information about all the cubes on the system from which the command was executed. If executed on an SRM, this command will show how the cube that is connected to that SRM is allocated. If executed on a remote development machine, this command will show all cubes that have been allocated from that workstation.
-n	Gets information about how cubes are allocated on all SRMs . It is equivalent to executing "cubeinfo -s" on each SRM on the network. This parameter is not allowed on the SRM.

All of the arguments are optional. *Cubeinfo* with no arguments will return information about the current attached cube.

Below are three examples of output using different *cubeinfo* options:

cubeinfo (no switches) from an SRM named "yosemite":

CUBENAME	USER	SRM	HOST	TYPE	TTYS
cube1	don	yosemite	yosemite	4	console

cubeinfo -a from a remote development machine named "shasta":

CUBENAME	USER	SRM	HOST	TYPE	TTYS
defaultname	don	klingon	shasta	4	ttyT1
alphabeta	don	klingon	shasta	8	

cubeinfo -s from a remote development machine named "shasta":

CUBENAME	USER	SRM	HOST	TYPE	TTYS
defaultname	don	klingon	shasta	4	REMOTE
alphabeta	don	klingon	shasta	8	REMOTE
testcube	jan	argon	shasta	1	REMOTE
defaultname	root	salmon	shasta	2m8	REMOTE

The columns are defined 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

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.

Usage Examples

Find out the name of your current cube	**cubeinfo**
Find out the names of all your cubes on this system	**cubeinfo -a**
Find out how cubes on the system you are on are allocated	**cubeinfo -s**
Find out about cube allocation on all SRMs	**cubeinfo -n**

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: internal cube usage limit	Limit of 10 cubes may be allocated. Try again later when system is not as busy.
cubeinfo: commser not responding	Internal system resource manager process error. Cube management processes are not responding. Try rebooting cube with *bootcube*.
cubeinfo: lifeline not responding	Internal system resource manager process error. Cube management processes are not responding. Try rebooting cube with *bootcube*.

See Also

getcube, relcube

GETCUBE(1) iPSC/2 GETCUBE(1)

Allocate a cube and make it the current cube.

Invocation Sequence

getcube [-c cubename] [-t cubetype] [-h srmname] [-t diag]

Description

-c cubename

Allows you to allocate multiple cubes. You assign names to cubes when you invoke getcube so that the system will know which cube a program will be connected to. Valid names are any ASCII character string, 15 characters or less. *Cubenames* cannot start with a dash ('-') or a space. One of the cubes that you get can be named the default name of "defaultname". Other cubes must be assigned unique names (see Usage Examples). If you specify a name that you (or anyone using your login name) is already using, an error is returned.

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.

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.

-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).

Size can be either:

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).

If you do not specify size, you will get the largest available cube. (Use *cubeinfo* to find out the actual size and type allocated.) The number of nodes allocatd will always be a power of two. If the size you specify is not a power of two, it will be rounded up.

The choices for type are as follows:

nothing	gets cube of any type.
vx	vector
sx	nodes with SX installed
m1	nodes with 1M byte of memory each
m4	nodes with 4M bytes of memory each
m8	nodes with 8M bytes of memory each
m16	nodes with 16M bytes of memory each
f	nodes with 387 installed

So, *-t 2* specifies a 2-node cube, *-t 16vx* specifies a 16-node vector cube, and *-t d3m4* specifies an 8-node 4M byte per node cube.

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

-h srmname Allows you to specify a particular system resource manager from multiple system resource managers on the network. Specify the network name of the SRM. This switch is only useful on a remote development machine and is ignored on an SRM. From an SRM, you can only allocate cubes on your local SRM.

Getcube allocates a new cube, names it for future reference, and makes it the current cube. If you want to run more than one application, allocate additional cubes, assigning each one a unique name. You are automatically attached to the last cube allocated. If you want to attach to another cube, use *attachcube*.

Getcube also starts a file server to handle I/O operations performed by the nodes in the cube. If you do a *getcube* and then change shells, current working directories, or environmental variables, use *newserver* to pass these changes to the file server. If you do not invoke *newserver*, changes you make to your host environment will not be passed to the file server. For example, if you change the file server's current directory (you do a *cd* after a *getcube*) and don't invoke

If you want to redirect the input or output of the file servers, redirect *getcube*. For example, invoke *getcube > outfile*.

All of the parameters are optional and will have defaults assigned if you omit them. *Getcube* with no arguments will reserve a cube for you with the default name, "defaultname". The number of nodes reserved for this cube will be the largest integral power of 2 that is possible given the exisiting set of unallocated nodes.

Getcube reports when it is successful. If you use the defaults, you can invoke *cubeinfo* to find out the size of the cube that was actually allocated.

All the cubes that you own are automatically released when you log out unless you specify otherwise by invoking *getcube* with *nohup;* for example *nohup getcube*.

Usage Examples

Allocate first available cube of any type and size with the default name of "defaultname"	`getcube`
Allocate a 4-node, vector cube named "alpha"	`getcube -c alpha -t 4v`
Allocate first available cube of any type and size with the default name on a system resource manager named "hal"	`getcube -h hal`

Errors

getcube: internal cube usage limit	Limit of 10 cubes may be allocated. Try again later when cube system is not as busy.
getcube: commser not responding	Internal system resource manager process error. Cube management processes are not responding. Try rebooting cube with *bootcube*.
getcube: lifeline not responding	Internal system resource manager process error. Cube management processes are not responding. Try rebooting cube with *bootcube*.
getcube: cubename already exists	Use a different name.
getcube: cubetype not found	Could not find a cube of the type requested.
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.

See Also

attachcube, cubeinfo, newserver, relcube

KILLCUBE(1) iPSC/2 KILLCUBE(1)

Kill node processes.

Invocation Sequence

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

Description

-c cubename	The name of the cube that you wish to kill a process on. If no name is given, the current attached cube is used. If the cube 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.
node...	You can kill any process on any node(s). If no node is specified, all processes, or just those specified with *-p*, on all nodes are killed.

Killcube kills the specified processes and flushes messages related to those processes. It is not an error to use *killcube* to kill a nonexistent process.

Usage Examples

Kill all processes on all nodes in the cube named "alpha"	`killcube -c alpha`
Kill process 1 on the first three nodes in the shell's current cube	`killcube -p 1 0 1 2`

LOAD(1) iPSC/2 LOAD(1)

Loads a user process into the cube.

Invocation Sequence

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

Description

-c cubename	The name of the cube that you wish to load. If no name is given, the current attached cube is used. If the cube does not exist, an error is returned.
-p pid	The process ID assigned to this process. If -p is not used, the pid will be zero. If you are loading multiple processes on a single node, you must assign each a unique pid. Valid pids include any integer value, zero and greater.
-H	Do not start the process immediately. The process will not start until *startcube* is executed.
node...	Specify the number of a node into which the process is to be loaded. You can enter multiple node numbers, separated by a space. If no node is specified, all nodes are loaded.
filename	The pathname of the process that you wish to load. The first characeter of *filename* must not be a numeric digit.
arguments...	Arguments for the process.

When you load a process with the *load* command, a message is displayed giving the size of the process in bytes and the amount of memory still available.

To load more than one process, invoke the *load* command for each process separately. Assign each a unique "pid". You can vary the values of the other arguments for each load. For example, use the "-H" on one load and not on another.

Usage Examples

Load process "proc3" into all the nodes in the current cube assigning it process id 3	`load -p 3 /usr/tom/proc3`
Load process "node_pgm" into nodes 3 and 4 of a cube named "alpha", and give it arguments "input_file" and "1000".	`load -c alpha 3 4 node_pgm input_file 1000`

Errors

load: invalid pid	*Pid* must not be negative.
load: invalid node	Node number must be in range of cube size.
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 different cube.
load: invalid loader record	Internal error, try again. You may have to use *bootcube*.
load: invalid loader message	Internal error, try again. You may have to use *bootcube*.
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 *attachcube* to change cubes within your program.

See Also

killcube

NEWSERVER(1) iPSC/2 NEWSERVER(1)

Start a new file server for the specified cube.

Invocation Sequence

newserver [-c cubename]

Description

-c cubename Allows you to start a new file server for a specific named cube. You give cubes names when you invoke *getcube*. Valid names are any ASCII string, 15 characters or less. *Cubenames* cannot start with a dash ('-') or a space. If this argument is not used, the current attached cube is used. If the specified cube does not exist, an error is returned. If you have no cube attached, an error is returned.

If you need to find out the names of the cubes that you have allocated, use the *cubeinfo* command.

You can specify only one cube at a time. If multiple names are specified, an error is returned.

A file server, to handle I/O for the nodes, is automatically created when you allocate a cube with *getcube*. If after doing *getcube* you change shells, current working directories, or environmental variables, those changes do not get passed on to the file server unless you invoke *newserver*. *Newserver* actually kills the original file server and starts a new one. If you do not invoke *newserver*, changes that you make to your environment will not be passed to the file server. One undesired result of not invoking *newserver* might be that your program cannot find files, because the file server's current directory was not changed when you did a *cd* between programs. If you want to redirect the standard input, standard output, or standard error of the file servers, redirect *newserver*. For example, invoke *newserver > outfile*. For information on redirection, refer to the UNIX System V User's Guide.

Usage Examples

Start a new file server for the current attached cube	`newserver`
Start a new file server for a cube named "alpha"	`newserver -c alpha`

Errors

newserver: cubename does not exist	Use *cubeinfo* to find out cube names.
newserver: internal cube usage limit	Limit of 10 cubes may be allocated. Try again later when system is not as busy.
newserver: commser not responding	Internal system resource manager process error. Cube management processes are not responding. Try rebooting cube with *bootcube*.
newserver: lifeline not responding	Internal system resource manager process error. Cube management processes are not responding. Try rebooting cube with *bootcube*.

See Also

getcube, syslog

RCC, RF77, RLD, RAS, RAR(1) iPSC/2

Remote iPSC/2 development utilities: the C compiler, FORTRAN compiler, linker, assembler, and librarian. These utilities are run from a remote development machine.

Invocation Sequence

rcc [[-cpp] [-h srmname] [cc_options]] filename(s)
rf77 [[-cpp] [-h srmname] [f77_options]] filename(s)
rld [[-h srmname] [ld_options]] filename(s)
ras [[-h srmname] [as_options]] filename(s)
rar [[-h srmname] [ar_options]] filename(s)

Description

-cpp — Run the remote host C preprocessor before compiling the files on the 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(s) — The necessary input file(s) for the specified remote utility.

Cube programs must be compiled, assembled, and linked on an iPSC/2 SRM. Therefore, programs developed on a remote development machine must be processed with these utility programs before being run on an SRM or cube.

The *rcc*, *rf77*, *rld*, *ras*, and *rar* programs allow you to generate executables for the iPSC/2 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* flle. 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.

RCC, RF77, RLD, RAS, RAR(1) (cont.)

The resulting files are:

.o.ipsc	iPSC object files. iPSC is appended to differentiate these files from object files produced on the remote development machine. These files are produced by *ras*, *rcc -c*, and *rf77 -c*.
.s.ipsc	iPSC Assembly Language source files, produced by *rcc -S* and *rf77 -S*.
.i.ipsc	Preprocessed C source file, produced by running the C preprocessor on the SRM using *rcc -P*.
.ipsc	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 are node executable, they must be "loaded". In both cases, trying to run them on the remote development machine will produce unpredictable results.
.a.ipsc	Archives made by *rar*. You can only link to these files on the SRM.

Usage Examples

Preprocesses a program named *ctest* on the remote development machine; remote C compiles *ctest* on the first SRM available	**rcc -cpp -c ctest.c** - produces file *ctest.o.ipsc*
Remote FORTRAN compile and link a program named *ftest*	**rf77 -o ftest ftest.f** - produces file *ftest.ipsc*
Remote link a program named *ctest*	**rld -o ctest -lnode ctest.o.ipsc** - produces file *ctest.ipsc*
Remote assemble a program named *astest*	**ras -o astest astest.s** - produces file *astest.o.ipsc*
Remote archive two object files	**rar -cr local_archive local_obj1.o.ipsc local_obj2.o.ipsc** - produces file *local_archive.aipsc*

Errors

<command>: 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.

RELCUBE(1) iPSC/2 RELCUBE(1)

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

Invocation Sequence

relcube [-c cubename] [-a]

Description

-c cubename	Allows you to release a specific named cube. You give cubes names when you invoke *getcube*. Valid names are any ASCII string, 15 characters or less. *Cubenames* cannot start with a dash ('-') or a space. If this argument is not used, the current attached cube is released. If the specified cube does not exist, an error is returned. If you have not allocated a cube, an error is returned. If you need to find out the names of the cubes that you have allocated, use the *cubeinfo* command.
-a	Allows you to release all of the cubes that you own on the system from which you invoked *relcube*.

All of the parameters are optional and will have defaults assigned if you omit them. All the cubes that you own are automatically released when you log out unless you invoke *getcube* with *nohup*.

Usage Examples

Release the current cube	`relcube`
Release the cube named "alpha"	`relcube -c alpha`
Release all cubes owned by you	`relcube -a`

Errors

relcube: cubename does not exist	Use *cubeinfo* to find out cube names.
relcube: internal cube usage limit	Limit of 10 cubes may be allocated. Try again later when system is not as busy.
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

getcube

STARTCUBE(1) iPSC/2 STARTCUBE(1)

Start a process executing on the cube.

Invocation Sequence

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

Description

-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. If the cube does not exist, an error is returned.
-p pid	The pid of the process that you wish to start. If no pid is given, all processes are started.
node...	You can start any process on any node(s). If no node is specified, processes on all nodes are started.

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

Usage Examples

Start all processes loaded with the *-H* argument	`startcube`
Start process 3 on nodes 5 and 6 in cube named "alpha"	`startcube -c alpha -p 3 5 6`

See Also

killcube, load

SYSLOG(1) iPSC/2 SYSLOG(1)

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

Invocation Sequence

syslog [-e]

Description

-e Send data to standard error of file server.

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*. *Syslog* 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 *print** statements from both the node and host processes go to the same device/file.

Syslog 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.

Usage Examples

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

```
host | syslog
```

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

```
host | syslog -e
```

WAITCUBE(1) iPSC/2 WAITCUBE(1)

Wait for process(es) on node(s) to complete.

Invocation Sequence

waitcube [-c cubename] [-p pid] [-f] [node...]

Description

-c cubename	The name of the cube that contains the process you wish to wait on. If no name is given, the attached cube is used. If the cube does not exist, an error is returned.
-p pid	The pid of the process to be waited for. If no pid is given, *waitcube* waits on all processes.
-f	If -f is used, *waitcube* returns when the first process that meets the requirements specified by the other switches completes. The default is to wait for all specified processes.
node...	You can wait for any process on any node(s). If no node is specified, *waitcube* waits for all nodes.

This command is useful when 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 on a nonexistent process.

Usage Examples

Wait for all processes on all nodes of attached cube to complete	`waitcube`
Wait for process 5 on cube named "alpha" to complete	`waitcube -c alpha -p 5`
Wait for the first process on node 6 to complete	`waitcube -f 6`

CHAPTER 3

iPSC/2 C ROUTINES

INTRODUCTION

This chapter documents each of the iPSC/2 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 A for more information on the error return version and a list of possible error messages with descriptions.

The routines are listed alphabetically. Table 3-1 gives a summary of the routines with a synopsis, brief description, and whether the routine is available for host, node, or both programs. The routines are documented as follows:

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

Table 3-1 (Page 1 of 4)
C Routine Summary

Routine	Host/Node	Synopsis	Description
attachcube	host	attachcube(cubename) char *cubename;	Attach to a cube and make it the current cube.
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.
ctohd htocd	host	ctohd(sv,n) htocd(sv,n)	Converting node byte order to remote development machine byte order or remote development machine to node byte order).
ctohf htocf		ctohf(sv,n) htocf(sv,n)	
ctohl htocl		ctohl(sv,n) htocl(sv,n)	
ctohs htocs		ctohs(sv,n) htocs(sv,n) unsigned long *sv short n	
cubeinfo	host	long cubeinfo(ct, numslots, global) struct cubetable *ct ; long numslots; long global;	Obtain information about allocated cubes.
flick	host/node	flick()	Relinquish CPU to another process.
flushmsg	host/node	flushmsg(typesel, node, pid) long typesel, node, pid;	Flush specified messages from the system.

Table 3-1 (Page 2 of 4)
C Routine Summary

Routine	Host/Node	Synopsis	Description
getcube	host	**getcube (cubename, cubetype, srmname, keep)** **char *cubename;** **char *cubetype;** **char *srmname;** **long keep;**	Allocate a cube.
ginv	host/node	**long ginv (j)** **long j;**	Returns the inverse of *gray*.
gray	host/node	**long gray (j)** **long j;**	Returns the binary-reflected Gray code for an integer.
handler	node	**handler(type, proc)** **long type;** **void (*proc)( );**	Provide user-written exception handler for program failures.
hrecv	node	**hrecv(typesel, buf,len, proc)** **long typesel;** **char *buf;** **long len;** **void (*proc)( );**	Provide user-written exception handler for receive traps.
infocount **infonode** **infopid** **infotype**	host/node	**long infocount( )** **long infonode( )** **long infopid( )** **long infotype( )**	Returns 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.
killcube	host/node	**killcube(node, pid)** **long node, pid;**	Terminate and clear out a process(es).

Table 3-1 (Page 3 of 4)
C Routine Summary

Routine	Host/Node	Synopsis	Description
killproc	host/node	killproc(node, pid) long node, pid;	Terminate a process.
killsyslog	host	killsyslog( )	Terminate *syslog* process.
led	node	led(lstate) long lstate;	Turn the node board's green LED on or off.
load	host/node	load(filename, node, pid) char *filename; long node, pid;	Load a node process.
masktrap	node	long masktrap(mask) long mask;	Enable or disable a mask trap.
mclock	host/node	unsigned long mclock( )	Return the time.
msgcancel	host/node	msgcancel(id) long id;	Cancel a send or receive operation.
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.
myhost	host/node	long myhost( )	Obtain node id of the host machine.
mynode	host/node	long mynode( )	Obtain the node id of the calling process.
mypid	host/node	long mypid( )	Obtain process id of the calling process.
newserver	host	newserver(cubename) char *cubename;	Start a new file server for the specified cube.

Table 3-1 (Page 4 of 4)
C Routine Summary

Routine	Host/Node	Synopsis	Description
nodedim	host/node	**long nodedim ( )**	Returns the dimension of the allocated cube.
numnodes	host/node	**long numnodes( )**	Obtain the number of nodes in the cube.
relcube	host	**relcube(cubename)** **char *cubename;**	Release a cube.
setpid	host	**setpid(pid)** **long pid;**	Sets process id of a host process.
setsyslog	host	**setsyslog(stdfd)** **long stdfd;**	Start the *syslog* program.
waitall	host/node	**waitall(node, pid)** **long node, pid;**	Wait for all the specified processes to complete.
waitone	host/node	**waitone(node, pid, cnode, cpid, ccode)** **long node, pid;** **long *cnode, *cpid, *ccode;**	Wait for a specified process to complete.

ATTACHCUBE(3) iPSC/2 ATTACHCUBE(3)

Attach to a cube and make it the current cube. This call is intended for use by host programs and is not available for node programs.

Synopsis

```
attachcube( cubename )
char *cubename;
```

Description

cubename is a pointer to a character string which specifies the name of the cube you want to attach to. You name a cube when you invoke *getcube*. Valid names include any ASCII string, 15 characters or less. Names longer than 15 characters will be truncated to 15 characters. *Cubenames* cannot start with a dash ('-') or a space. If the parameter is null or points to a null string, the default *cubename*, "defaultname", is used. If you specify a cube that has not been allocated with *getcube*, an error message results and the process will terminate. Use *cubeinfo* to find out the names of allocated cubes.

The *attachcube* call is used when you want to run applications in several cube partitions. It allows you to attach to a cube that you have previously allocated. You can only attach to one cube at a time. Use *getcube* to allocate a cube initially and *attachcube* to change cubes within your program.

The *attachcube* call changes a process's current cube. 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*.

Example

```
char *cube1, *cube2;
   .
   .
cube1 = "alpha";
cube2 = "beta";

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

Return Values

None.

Errors

attachcube: cubename does not exist	Use *cubeinfo* to determine cube names.
attachcube: internal cube usage limit	Limit of 10 cubes may be allocated. Try again later when system is not so busy.
attachcube: commser not responding	Internal system resource manager process error. Cube management processes are not responding. Try rebooting cube with *bootcube*.
attachcube: lifeline not responding	Internal system resource manager process error. Cube management processes are not responding. Try rebooting cube with *bootcube*.

See Also

cubeinfo, getcube, relcube

CPROBE(3) iPSC/2 CPROBE(3)

Wait for a message to arrive. *Cprobe* blocks the calling process until the message is available for receipt.

Synopsis

```
cprobe( typesel )
long typesel;
```

Description

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. *Typesel* (*type* selector) allows you to select a specific message *type* or a set of message *types* based on a 32-bit mask.

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.

Cprobe causes a process to wait until a message of the selected *type(s)* is available for receipt. You specify the message *type* with the send operation. The *cprobe* call waits for a message whose *type* matches the *type(s)* specified by the calling process. 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.

Cprobe is synchronous, causing the calling process to be blocked until the desired message is available for receipt. Use *iprobe* 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.

Example

```
#define INIT_TYPE 10

long len, msg_id;
char msgbuf[80];
   .
   .
cprobe(INIT_TYPE);                    /* Wait for a message with type = 10 to arrive */
if(infocount () <= sizeof (msgbuf))
   crecv(INIT_TYPE, msgbuf,sizeof(msgbuf)); /* Receive message if it fits in buffer*/
```

Return Values

None.

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.

See Also

infocount, infonode, infopid, infotype, iprobe

CRECV(3) iPSC/2 CRECV(3)

Receive a message and wait for completion. *Crecv* blocks the calling process until the message is received.

Synopsis

```
crecv( typesel, buf, len )
long typesel;
char *buf;
long len;
```

Description

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. *Typesel* (*type* selector) allows you to select a specific message *type* or a set of message *types* based on a **32**-bit mask.

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 information on building a *typesel* mask.

buf is a pointer to the buffer where the received message will be stored. Valid entries point to any legal C 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) 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.

Crecv initiates the receipt of a message. The *crecv* call waits 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.

The *crecv* procedure is synchronous, 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.

Example

```
#define BUFLEN1 256

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

Return Values

None.

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 length does not exceed buffer size.
crecv: invalid buffer pointer	Specify a pointer which contains the address of a valid data buffer.
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:invalid type	Use a non-negative number.

See Also

csend, infocount, infonode, infopid, infotype, irecv, isend

CSEND(3) iPSC/2 CSEND(3)

Send a message and wait for completion. *Csend* blocks the calling process until the message is sent.

Synopsis

```
csend( type, buf, len, node, pid )
long type;
char *buf;
long len, node, pid;
```

Description

type — is an integer value that identifies the message that you are sending. Any non-negative number is valid. *Type* is a user-defined variable typically used to identify the kind of information contained in the message. Types 2,000,000,000 and up are used by the system and should be avoided.

buf — is a pointer to the buffer which contains the message to be sent. Valid entries point to any legal C 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 (in bytes) that you wish 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.

node — is an integer value that identifies a particular node or **-1** to indicate that the message should be sent to all nodes. Nodes are numbered so that they range from **0** to (*numnodes* - **1**) where *numnodes* is the number of nodes in the partition. If you want to send a message to the host, use *myhost* to determine the host's id.

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*.

CSEND(3) (cont.) iPSC/2 CSEND(3) (cont.)

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

Csend is synchronous, causing the calling process to be blocked until the send operation is complete. Use *isend* when you want to initiate a send request and want the process to continue.

Example

```
char *msg;
long len;
   .
   .
sprintf(msg, %s, "Hello node 0");
len = strlen(msg) + 1;      /* Length of the string including the null terminator */
csend(5, msg, len, 0, 1);   /* Send type 5 message to process 1 on node 0 */
   .
sprintf(msg, %s, "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 */
```

Return Value

None.

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: buffer length exceeds allocation	Make sure length does not exceed buffer size.
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 4 or make more memory available to this process.

See Also

cprobe, crecv, iprobe, irecv, isend

CTOHD, CTOHF, CTOHL, CTOHS(3) HTOCD, HTOCF, HTOCL, HTOCS(3)

iPSC/2

Converting node byte order to remote workstation byte order or remote workstation to node byte order.

Synopsis

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

```
unsigned long *sv;
short n;
```

Description

sv — starting address of swap variable(s). Individual variables of any type, elements of structures, or arrays can be used as long as the address is used. 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.

The remote development machine may have a different internal byte order representation of integer and floating point values than the nodes. Utilities are provided for each remote development machine to convert node byte order to the development machine byte order. Utilities are also provided to convert remote development machine byte order to node byte order. When sending a message *from* the remote development machine to the nodes, the byte order that the node expects to receive is node byte order. When messages are sent from the nodes to a remote development machine, the byte order is node byte order.

The byte swapping utilities are provided for source code compatability in both SRM and remote development machine libraries. On the SRM, they do nothing and are only useful for the remote development host processes.

Using the byte swapping utilities, the host program running on the remote development machine must convert to node byte order prior to sending a message and also convert messages received from the node prior to using the message.

Example

```
main() {
        .
        .
long datain[100];
long dataout[100];
        .
        .
   HTOCL(datain,100)
   csend(NODE_TYPE, datain, sizeof(datain), -1, NODE_PID);
        .
        .
   crecv(NODE_TYPE, dataout, sizeof(dataout));
   CTOHL(dataout,100)
        .
        .
}
```

Errors

None

CUBEINFO(3) iPSC/2 CUBEINFO(3)

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

Synopsis

```
long cubeinfo(ct, numslots, global )
struct cubetable *ct;
long numslots;
long global;
```

Description

ct is a pointer that specifies a buffer table where you want the cube information to be returned.

numslots is the number of structure elements in *cubetable*, which is where the cube information is returned.

global is an integer value that indicates which cube(s) on the system you want information about. The possible settings are:

- **0** returns information about the current attached cube.
- **1** returns information about all the cubes that you own and that have been allocated on the host that invoked the call.
- **2** returns information about all the cubes on the system from which the command was executed. If executed on an SRM, this command will show how the cube connected to that SRM is allocated. If executed on a remote development machine, this command will show all cubes that have been allocated from that workstation.
- **3** returns information about how cubes are allocated on all SRMs. It is equivalent to using global = 2 on each SRM on the network. This value is not allowed on the SRM.

Cubeinfo allows you to obtain ownership information about allocated cubes. *Cubeinfo* gathers information and fills slots in the *cubetable* buffer for all of the selected cubes, or for *numslots*, whichever is less. If there are fewer selected cubes than *numslots*, the remaining slots will be cleared to all zeros.

Example

The format of the *cubetable* structure is given below. This structure is defined in */usr/include/cube.h*.

```
#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 resource manager where cube is attached */
char hostname[NAMELEN];          /* Name of host where getcube was invoked. Name    */
                                 /*   is either the remote system name or srmname   */
char tty[NAMELEN];               /* tty where getcube was invoked                   */
char cubetype [NAMELEN];         /* Type of this subcube                            */
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 4 ttys may be attached to the same cube.  */
                                 /* Strings for attached ttys, are of the format    */
                                 /* '\0' for no tty and '/dev/tty' for valid ttys.  */
};
```

Return Values

The number of cubes found. This may or may not equal *numslots*.

Errors

cubeinfo: there is no attached cube	Use *attachcube* to attach your process to a cube.
cubeinfo: there are no cubes allocated	No information was available because no cubes have been allocated.
cubeinfo: commser not responding	Internal system resource manager process error. Cube management processes are not responding. Try rebooting cube with *bootcube*.
cubeinfo: lifeline not responding	Internal system resource manager process error. Cube management processes are not responding. Try rebooting cube with *bootcube*.
cubeinfo: global value invalid	Use a value of 0, 1, 2, or 3 for *global*.

See Also

attachcube, getcube, relcube

FLICK(3) iPSC/2 FLICK(3)

Relinquish CPU to another process.

Synopsis

```
flick()
```

Description

Flick allows a process to defer execution to another process on the same node. The scheduler runs processes on the 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 CPU 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. *Flick* simply allows other node processes to execute. *Csend, crecv, cprobe,* or *msgwait* operations may also defer execution of a calling process until they complete their task.

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.

You should use the *flick* routine when the current process has no useful work to do. For example, 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 process sending the message may be running on the same node, but be suspended waiting for the CPU.

Flick is a delay mechanism, not a synchronization mechanism. The scheduler does not know what conditions a process is waiting for. The following example shows the correct use of *flick*.

Example

```
id1 = irecv( 101, buf1, sizeof(buf1) );
id2 = irecv( 102, buf2, sizeof(buf2) );
/* Do some other processing here. */
/* Now wait for either receive to complete. */
for (;;) {
   if (msgdone (id1) ) {
      /* Process message 1. */
      break;
   } else if (msgdone (id2) ) {
      /* Process message 2. */
      break;
   } else {
      flick ();
   }
}
```

Return Values

None.

FLUSHMSG(3) iPSC/2 FLUSHMSG(3)

Flush specified messages from the system.

Synopsis

```
flushmsg( typesel, node, pid )
long typesel, node, pid;
```

Description

typesel is an integer value that specifies the *type(s)* of message that you want to flush. You assign a *type* to a message when you initiate a send operation. *Typesel* (*type* selector) allows you to select a specific message *type* or a set of message *types* based on a **32**-bit mask.

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 an integer value that identifies a particular node or **-1** to indicate all nodes and the host. Nodes are numbered so that they range from **0** to (*numnodes* - **1**) where *numnodes* is the number of nodes in the partition. If you want to flush messages just from the host, use *myhost* to determine the host's id.

pid is an integer value that specifies the 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*.

FLUSHMSG(3) (cont.) iPSC/2 FLUSHMSG(3) (cont.)

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.

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.

Flushmsg should be used in conjunction with *msgcancel* or *killproc* to ensure that all specified messages are flushed.

Example

```
char *msg1, *msg2, *msg3;
long node, pid, msg_id;
   .
   .
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);  /* Asynchronous send */

flushmsg(-1, node, pid); /* Flush messages of all types waiting to be received by */
                         /* process 0 on node 1. Note that since msg3 may not  */
                         /* have been sent yet, it may not be flushed.    */
msgcancel(msg_id);       /* Cancel msg3 */
flushmsg (-1, node, pid);   /* Now msg3 will be flushed */
```

Return Values

None.

Errors

flushmsg: invalid node — Use *numnodes* to determine cube size and *myhost* to determine host number.

See Also

load, waitall, waitone

GETCUBE(3) iPSC/2 GETCUBE(3)

Allocate a cube. This call is intended for use by host programs and is not available for node programs.

Synopsis

```
getcube (cubename, cubetype, srmname, keep);
char *cubename;
char *cubetype;
char *srmname;
long keep;
```

Description

cubename is a pointer to a character string which 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. *Cubenames* cannot start with a dash ('-') or a space. If the parameter is null or points to a null string, the default *cubename*, "defaultname", is used. If you specify a name that you (or anyone using your login name) are already using, an error is returned. Use *cubeinfo* to find out the names of allocated cubes. Use *attachcube* to access a different allocated cube.

cubetype is a pointer that specifies the size and type (in a contiguous string) of the cube that you want to allocate. Specify size first followed by type in a continuous string (no spaces). The *cubetype* string must be 15 characters or less. Strings longer than 15 characters will be truncated to 15 characters. If the parameter is null or points to a null string you will get the largest available cube. Use *numnodes* to find out the number of nodes allocated or *cubeinfo* to find out the type of cube allocated.

The choices for size are:

n where *n* is the literal number of nodes (for example: 8) or,

d*n* where **d** indicates dimension and *n* is the size of the dimension (for example: d3)

If you do not specify size, you will get the largest available cube. (Use *cubeinfo* to find out the actual size and type allocated.) The number of nodes allocated will always be a power of two. If the size you specify is not a power of two if will be rounded up.

The choices for type are:

nothing	gets a cube of any type.
vx	vector
sx	nodes with SX installed
m1	nodes with at least 1M byte of memory
m4	nodes with at least 4M bytes of memory
m8	nodes with at least 8M bytes of memory
m16	nodes with at least 16M bytes of memory
f	nodes with 387 installed

So, *2* specifies a 2-node cube, *16vx* specifies a 16-node vector cube, and *d3m4* specifies an 8-node 4M byte memory cube. To get a hybrid cube, specify the different sizes and types separated by "/" in one string. For example, *4vx/4m8* gets four vector nodes and four nodes with 8M bytes of memory.

srmname is a pointer to a character string which specifies the name of the host connected to the cube that you want to allocate. If you are on a system resource manager, *srmname* defaults to your system resource manager *srmname*. 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. Valid names include any network host name of a system resource manager with cubes attached.

keep is a flag that indicates the lifetime of the partition. 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 does a *relcube*.

Getcube allows you to allocate a new cube. *Getcube* assigns a name, type, host, and lifetime of the partition to a 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.

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.

If you call *getcube* and then change shells, current working directories, or environmental variables, use *newserver* to change file servers to the current working environment. If you want to redirect the input or output of the file servers, redirect the output of the program that calls *getcube*. For example, invoke your program with **progname > outfile.**

All the cubes that you own are automatically released when you log out unless you invoke your program with **nohup.** For example, invoke your program with **nohup progname**.

If any of the parameters are not specified, default values are used. *Getcube* with no arguments will reserve a cube for you with the default name, "defaultname". The number of nodes reserved for this cube will be the largest integral power of 2 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. If you use the defaults, you can use *cubeinfo* to find out the size of the cube that was actually allocated.

Example

```
getcube ("alpha",  "16vx", NULL, 0);
getcube ("beta",  "4m8", "trainon", 1);      /* Current cube is "beta" */
   .
   .
attachcube ("alpha");                        /* Current cube is "alpha" */
   .
   .
```

Return Values

None.

Errors

getcube: internal cube usage limit	Limit of 10 cubes may be allocated. Try again later when system is not as busy.
getcube: commser not responding	Internal system resource manager process error. Cube management processes are not responding. Try rebooting cube with *bootcube*.
getcube: lifeline not responding	Internal system resource manager process error. Cube management processes are not responding. Try rebooting cube with *bootcube*.
getcube: cubename already exists	Use a different name.
getcube: cubetype not found	Could not find a cube of the type requested.
getcube: no srm that matched your request was found	Remote host error only. Could not find an SRM with a cube of the type requested.

See Also

attachcube, cubeinfo, newserver, numnodes, relcube

GINV(3) iPSC/2 GINV(3)

Returns the inverse function of *gray*. It returns the position of an element in the binary-reflected Gray code sequence.

Synopsis

```
long ginv (j)
long j;
```

Description

j is a positive long integer value.

Ginv takes a Gray code element (positive long integer) as input and returns the position of the element in the binary-reflected Gray code sequence.

Example

Given the binary-reflected Gray code sequence,

000,	001,	011,	010,	110,	111,	101,	100	→	Gray code sequence in binary
0	1	3	2	**6**	7	5	4	→	Gray code sequence in decimal
0	1	2	3	**4**	5	6	7	→	position of each element in sequence

```
ginv (6);
```

returns a 4.

Return Values

Inverse of *gray*.

Errors

None.

See Also

gray

GRAY(3) iPSC/2 GRAY(3)

Returns the binary-reflected Gray code for an integer.

Synopsis

```
long gray (j)
long j;
```

Description

j is a positive long integer value.

Gray codes are 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 in position *j* in the binary-reflected Gray code sequence.

Example

Given the binary-reflected Gray code sequence,

000,	001,	011,	010,	110,	111,	101,	100	→	Gray code sequence in binary
0	1	3	2	6	7	5	4	→	Gray code sequence in decimal
0	1	2	3	4	5	6	7	→	position of each element in sequence

```
gray (4);
```

returns a 6.

Return Values

Binary-reflected Gray code for an integer.

Errors

None.

See Also

ginv

HANDLER(3) iPSC/2 HANDLER(3)

Provide user-written exception handler for program failures. This call is intended for use by node programs and is not available for host programs.

Synopsis

```
handler( type, proc)
long type;
void (*proc)( );
```

Description

type is an integer value that identifies the exception handler type. Exception handlers may be provided for the types listed below.

proc is the procedure executed when the specified exception occurs. This procedure is user-written and must be a C procedure with one argument. When the procedure is called, the value of the argument depends on the exception type selected. For exception handler types **11-14**, the argument will be the hardware exception code explained in Chapter 9 of the 80386 Programmer's Reference Manual.

The *handler* routine allows a node process to assign a user-written exception handler to the occurrence of specific exceptions. Exception handlers are effective only for exceptions caused by the process that calls *handler*.

The default action, if no handler is specified, is to print a message and kill the process. Exception types 0-31 indicate program failure.

Hardware exception types:

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 1-byte breakpoint instruction (INT 3) is executed.

Hardware exception types cont:

4 Overflow
Occurs when the *into* instruction is executed if the overflow bit of the *flags* register is set.

5 Bounds Check
Occurs when the *bound* instruction is executed if 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 if 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 if 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 non-present 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/2, an exception 16 is always a floating point error.

Return Values

None.

Errors

handler:invalid handler type	Specify one of types listed.
handler:invalid length	Correct PROC parameter.

See Also

386 Programmer's Reference Manual

HRECV(3) iPSC/2 HRECV(3)

Provide user-written exception handler for receive traps . This call is intended for use by node programs and is not available for host programs.

Synopsis

```
hrecv(typesel, buf, len, proc)
long typesel;
char * buf;
long len;
void (*proc) ( );
```

Description

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. *Typesel* (*type* selector) allows you to select a specific message *type* or a set of message *types* based on a 32-bit mask.

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 is a pointer to the buffer where the received message will be stored. Valid entries point to any legal C data type. 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 value 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 completes. This procedure is user-written and must be a C procedure with four arguments.

The *hrecv* routine posts a request to receive a message into the buffer and returns immediately. It also sets up a handler procedure which will be called when the receive operation is complete. The handler is called with parameters containing information about the message, so that the info calls are not needed within the handler.

The receive trap handler is called with four arguments:

```
proc(type, count, node, pid)
long type, count, node, pid;
```

These are the values which would be returned by the *info* calls.

It is possible to post several *hrecv* requests simultaneously, with separate handlers or with a common handler. In addition, receive completion interrupts can be termporarily blocked using the *masktrap* call to protect critical sections. If a message comes in which is too long to fit in the buffer, no error is given but the receive completes and the message is lost. You can detect this situation by checking the count argument in the handler.

Example

```
hrecv(typesel, buf, len, proc);  /* Initiate receive and set up handler */
oldmask = masktrap(1);           /* Temporarily disable receive trap */
   .                             /* Do critical processing here */
   .
masktrap(oldmask);               /* Re-enable receive trap */
```

Return Values

None.

Errors

hrecv:buffer length exceeds allocation	Make sure length does nt 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	Simplify the application program.
hrecv:too many requests	Use *msgwait* for *isend* or *irecv* requests that are outstanding.
hrecv:invalid buffer pointer	Specify a pointer which contains the address of a valid data buffer.

See Also

infocount, infopid, infonode, infotype, masktrap

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

Returns information about a pending or received message.

Synopsis

long infocount( )

long infonode( )

long infopid( )

long infotype( )

Description

Info calls return information about a pending or received message. They return information about the message detected when *cprobe* completes or *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**.

The *info* calls must immediately follow the related message-passing call. Intervening statements, such as "printf" could invalidate the values returned by the *info* calls.

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: INFOCOUNT, INFONODE, INFOPID, INFOTYPE(3)(cont.)

Example

```
#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 */
   .
   .
msg_id = irecv(10, buf, sizeof(buf));   /* Post a receive for a type 10 message */
   .
   .
if(msgdone(msg_id)) {                /* After receive is complete, find out who sent it */
      from_node = infonode ();
      from_pid = infopid ();
}
```

Return Values

The indicated information is returned. The return value is undefined unless the call is made after one of the calls listed above which indicate that a message is ready.

INFO: INFOCOUNT, INFONODE, INFOPID, INFOTYPE(3)(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, iprobe, msgdone, msgwait

IPROBE(3) iPSC/2 IPROBE(3)

Determine whether a message of a selected type is pending.

Synopsis

```
long iprobe( typesel )
long typesel;
```

Description

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. *Typesel* (*type* selector) allows you to select a specific message *type* or a set of message *types* based on a **32**-bit mask.

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 instructions on how to build a *typesel* mask.

Iprobe allows a process to determine whether a message of a selected *type* is available for receipt. You specify the message *type* with the send operation. The *iprobe* call 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.

The *iprobe* procedure is asynchronous 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.

Example

```
      .
      .
for (;;) {
   if (iprobe(1)) {
   /* Process type 1 message . */
      break;
   }
   else if (iprobe(2)) {
   /* Process type 2 message. */
      break;
   }
   else {
      flick();
   }
}
```

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

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

IRECV(3) iPSC/2 IRECV(3)

Receive a message.

Synopsis

```
long irecv( typesel, buf, len )
long typesel;
char *buf;
long len;
```

Description

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. *Typesel* (*type* selector) allows you to select a specific message *type* or a set of message *types* based on a **32**-bit mask.

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 is a pointer to the buffer where the received message will be stored. Valid entries point to any legal C data type. Data types should match in send and receive operations. *Buf* is not available for use until *msgwait* returns or *msgdone* indicates that the receive has completed.

len is an integer value 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.

Irecv initiates the receipt of a message from a process. The *irecv* call searches for a message whose *type* matches the *type*(s) specified in the *typesel* parameter. A message id is generated which is used by *msgdone* and *msgwait* to determine the status of a message. When the message is received it is stored in *buf*. 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.

The *irecv* procedure is asynchronous, allowing the calling process to continue even if the desired message is not yet available. *Irecv* 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.

Example

```
int buf[50];
long msg_id, type;
   .
   .
msg_id = irecv(type, buf, sizeof(buf));   /* Initiate receipt of a message */

/* Perform tasks which are not dependent on the message being received.
   .
   .
msgwait(msg_id);      /* Now wait to guarantee message receipt. */

/* Now you can use the contents of "buf" and/or use the info routines to find out more */
/* about the message received. */
```

Return Values

Returns a non-negative message id which is to be used in *msgcancel*, *msgdone*, or *msgwait*.

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 length does not exceed buffer size.
irecv: invalid buffer pointer	Specify a pointer which contains the address of a valid data buffer.

See Also

csend, infocount, infonode, infopid, infotype, isend, msgcancel, msgdone, msgwait

ISEND(3) iPSC/2 ISEND(3)

Send a message.

Synopsis

```
long isend( type, buf, len, node, pid )
long type;
char *buf;
long len, node, pid;
```

Description

type is an integer value that identifies the message that you want to send. Any non-negative integer is valid. The message *type* is used by the receiving process to identify what kind of information is contained in the message. Types 2,000,000,000 and up are used by the system and should be avoided.

buf is a pointer to the buffer which contains the message that you want to send. Valid entries point to any legal C data type. Data types should match in send and receive operations. You should not modify *buf* until *msgwait* returns or *msgdone* indicates that *isend* has completed. Otherwise, corrupted data might be sent.

len is an integer value 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 a particular node or **-1** to indicate the message should be sent to all nodes. Nodes are numbered so that they range from **0** to (*numnodes* - **1**) where *numnodes* is the number of nodes in the partition. If you want to send a message to the host, use *myhost* to determine the host's id.

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*.

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. Use *msgdone*, or *msgwait* to determine whether the *isend* operation has completed.

Isend 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.

Example

```
char msg[132];
long msg_id, len, node, pid, type;
   .
   .
len = sizeof(msg);
node = 16;
pid = 10;
type = 1;
msg_id = isend(type, msg, len, node, pid); /* Send type 1 message to process 10 */
   .                                       /* on node 16 */
   .
/*Perform tasks that do not change the contents of msg */
   .
   .
msgwait(msg_id);

/* Now you can re-use the message buffer. */
```

Return Values

Returns a non-negative message id which is used by *msgcancel*, *msgdone*, or *msgwait*.

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: buffer length exceeds allocation	Make sure length does not exceed buffer size.
isend: invalid buffer pointer	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* for *isend* or *irecv* requests that are outstanding.

See Also

crecv, csend, irecv, msgcancel, msgdone, msgwait

KILLCUBE(3) iPSC/2 KILLCUBE(3)

Terminate and clear out a process(es).

Synopsis

```
killcube( node, pid )
long node, pid;
```

Description

node is an integer value that identifies a particular node on which you want to terminate processes and flush messages or **-1** to indicate all nodes. Nodes are numbered so that they range from **0** to (*numnodes* **-1**) where *numnodes* is the number of nodes in the partition. A host id is invalid.

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.

Killcube kills the specified node process(es) and flushes messages related to those processes. It is not an error to use *killcube* to kill a nonexistent process. It is included for convenience and is equivalent to the sequence:

```
killproc( node, pid );
waitall( node, pid );
flushmsg(-1, node, pid );
```

Return Values

None.

Errors

None.

See Also

load, killproc, flushmsg, waitall

KILLPROC(3) iPSC/2 KILLPROC(3)

Terminate a process.

Synopsis

```
killproc( node, pid )
long node, pid;
```

Description

node is an integer value that identifies a particular node on which you want to terminate a process or **-1** to indicate all nodes. Nodes are numbered so that they range from **0** to (*numnodes* **-1**) where *numnodes* is the number of nodes in the partition. A host id is invalid.

pid is an integer value that specifies the process id 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.

Killproc kills the node process specified by *pid* and *node*. *Killproc* has no effect on messages already sent 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.

Example

```
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. */
```

Return Values

None.

Errors

killproc:invalid pid Use a non-negative number.

See Also

load, flushmsg, killcube, waitone, waitall

KILLSYSLOG(3) iPSC/2 KILLSYSLOG(3)

Terminate *syslog* process. This call is intended for use by host programs and is not available for node programs.

Synopsis

```
killsyslog( )
```

Description

Killsyslog kills the *syslog* process and resets the standard output to the terminal, /dev/tty. *killsyslog* can only be used to kill the syslog process started by *setsyslog*. *Killsyslog* cannot be used to kill the syslog process started by the *syslog* command.

The purpose of *syslog* 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, *syslog* must be killed and restarted by invoking *setsyslog* to make sure the output goes to the correct file server.

Return Values

None.

Errors

None.

See Also

setsyslog

LED(3) iPSC/2 LED(3)

Turn the node board's green LED (light emitting diode) on or off. This call is intended for use by node programs and is not available for host programs.

Synopsis

```
led( lstate )
long lstate;
```

Description

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 allows a process to turn the node board's green LED on or off depending on the value of *lstate*. Behavior is not defined if values other than **0** or **1** are used.

Return Values

None.

Errors

None.

LOAD(3) iPSC/2 LOAD(3)

Load a node process.

Synopsis

```
load( filename, node, pid )
char *filename;
long node, pid;
```

Description

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 any ASCII character string.

node is an integer value that identifies the node you want to load a process into (use **-1** to select all nodes). Nodes are numbered so that they range from **0** to (*numnodes* - **1**) where *numnodes* is the number of nodes in the partition. 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. 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 node are allowed. A process can use *mypid* to obtain its process id after it begins execution.

Filename is loaded into the specified *node*(s), *pid* is assigned to the process(es), and the process(es) 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.

Return Values

None.

Errors

load: there is no attached cube	Use *getcube* to allocate a cube and *attachcube* to change cubes within you program.
load: invalid pid	*Pid* must not be negative.
load: invalid node	Node number must be in range.
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. You may have to *bootcube*.
load: invalid loader message	Internal error, try again. You 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

MASKTRAP(3) iPSC/2 MASKTRAP(3)

Enable or disable a receive trap. This call is intended for use by node programs and is not available for host programs.

Synopsis

```
long masktrap(mask)
long mask;
```

Description

mask	a set of bits which correspond to receive traps. **1** is used to block receive traps and **0** is used to enable receive traps.

Masktrap is used to temporarily mask the receive traps for a process.

Example

```
hrecv(typesel, buf, len, proc);  /* Initiate receive and set up handler */
oldmask = masktrap(1);           /* Temporarily disable receive trap */
   .                             /* Do critical processing here */
   .
masktrap(oldmask);               /* Re-enable receive trap */
```

Return Values

The previous value of the mask.

Errors

None.

See Also

hrecv

MCLOCK(3) iPSC/2 MCLOCK(3)

Return the time.

Synopsis

unsigned long mclock()

Description

The *mclock* routine provides a simple mechanism to measure time intervals. *Mclock* 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.

The time value returned by *mclock* on the host reflects only the number of milliseconds that the UNIX system has 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 initializes the nodes. For more information on *bootcube*, refer to the iPSC/2 System Administrator's Guide.)

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.

Return Values

An unsigned long integer containing the time value.

Errors

None.

MSGCANCEL(3) iPSC/2 MSGCANCEL(3)

Cancel an asynchronous send or receive operation.

Synopsis

```
msgcancel( id )
long id;
```

Description

id	is the message id returned when you invoke an *isend* or *irecv* operation.

Msgcancel allows a process 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* call. 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 and that the buffer may be reused. The message id is no longer valid and cannot be used in *msgdone* or *msgwait*.

Example

```
char *buf;
long msg_id, type;
   .
   .
msg_id = irecv(type, buf, sizeof(buf));
   .
   .                          /* Perform tasks while waiting for receive to complete. */
if (error)                    /* error condition */
   msgcancel(msg_id);
```

Return Values

None.

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(3) iPSC/2 MSGDONE(3)

Determine whether an asynchronous send or receive operation has completed.

Synopsis

```
long msgdone( id )
long id;
```

Description

id is the message id returned when you invoke an *isend* or *irecv* operation.

Msgdone allows a process to determine whether an *isend* or *irecv* operation has completed. The *id* parameter identifies the operation you are checking status on. *Id* must correspond to the value returned by a previous *isend* or *irecv* call.

When *msgdone* returns **1** for a receive operation, you can use the *info* calls to get more information about the message.

Example

```
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. */
}
```

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.

0 indicates that the send or receive message is not yet complete and has no effect on the *info* calls. In this case, you can use the same *id* again until the send or receive operation completes.

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(3) iPSC/2 MSGWAIT(3)

Wait for completion of an asynchronous send or receive operation.

Synopsis

```
msgwait( id )
long id;
```

Description

id	is the message id returned when you invoke an *isend* or *irecv* operation.

Msgwait allows a node process to wait until an *isend* or *irecv* operation is complete. The *id* parameter identifies the operation that you are waiting for. *Id* must correspond to the value returned by a previous *isend* or *irecv* call.

Msgwait returns when the message is complete and the associated message buffer is available for reuse, in the case of a send operation, or when 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 the *info* calls to get more information about the message. You cannot use the id parameter again because the send or receive operation is complete.

Example

```
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'. */
```

Return Values

None.

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

See Also

infocount, infonode, infopid, infotype, irecv, isend

MYHOST(3) iPSC/2 MYHOST(3)

Obtain the node id of the host machine.

Synopsis

long myhost( )

Description

Myhost returns the node id of the caller's host machine for use in send and receive calls. For host processes, *myhost* returns the same id as *mynode* (node id of the calling process).

Example

```
char *msg;
long host_id;
   .
   .
host_id = myhost();
csend(10, msg, sizeof(msg), host_id, 1);  /* Send message to host process. */
```

Return Values

The host id is returned.

Errors

myhost: no pid defined	Use *setpid* to define a host process id.

See Also

csend, getcube, isend, mynode

MYNODE(3) iPSC/2 MYNODE(3)

Obtain the node id of the calling process.

Synopsis

long mynode( )

Description

Mynode returns the node id of the calling process. Nodes are numbered so that they range from **0** to (*numnodes* - **1**) where *numnodes* is the number of nodes in the partition. The host's node id is the same id returned by *myhost*. The node id is used in send and kill operations.

Example

```
long this_node;
   .
   .
this_node = mynode();      /*Obtain my logical node id */
killproc(this_node, -1); /* Kill all of the processes on this node, including myself. */
```

Return Values

The node id of the process that initiated the routine is returned.

Errors

mynode: no pid defined	Use *setpid* to define a host process id.

See Also

csend, flushmsg, getcube, isend, load, killcube, killproc, myhost

MYPID(3) iPSC/2 MYPID(3)

Obtain process id of the calling process.

Synopsis

long mypid( )

Description

Mypid returns the 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 the *setpid* call. (This process id should not be confused with the UNIX process id.) *Pids* are used in send and kill operations.

Example

```
char *buf;
long type;
   .
   .
csend(type, buf, sizeof(buf), -1, mypid());  /* Send message to the processes */
                                             /* with the same pid as mine on */
                                             /* all nodes. */
```

Return Values

The process id is returned.

Errors

mypid: no pid defined	Use *setpid* to define a host process id.

See Also

csend, flushmsg, isend, load, killcube, killproc, setpid

NEWSERVER(3) iPSC/2 NEWSERVER(3)

Start a new file server for the specified cube. This call is intended for use by host programs and is not available for node programs.

Synopsis

```
newserver( cubename )
char *cubename;
```

Description

cubename is a pointer to a character string which specifies the name of the cube. 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. *Cubenames* cannot start with a dash ('-') or a space. 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 *cubeinfo* to find out the names of allocated cubes.

A file server, to handle I/O for the nodes, is automatically created on the host machine when you allocate a cube with *getcube*. If, after calling *getcube*, you change shells, current working directories, or environmental variables, those changes do not get passed on to the file server unless you invoke *newserver*. *Newserver* starts a new file server and kills the old file server.

Example

```
getcube("alpha", "8", NULL, 0);
chdir("/usr/tmp");
newserver("alpha"); /* Start a new file server that will use files in '/usr/tmp' for I/O. */
```

Return Values

None.

Errors

newserver: cubename does not exist	Use *cubeinfo* to determine cube names.
newserver: internal cube usage limit	Limit of 10 cubes may be allocated. Try again later when system is not as busy.
newserver: commser not responding	Internal system resource manager process error. Cube management processes are not responding. Try rebooting cube with *bootcube*.
newserver: lifeline not responding	Internal system resource manager process error. Cube management processes are not responding. Try rebooting cube with *bootcube*.

See Also

cubeinfo, getcube

NODEDIM(3) iPSC/2 NODEDIM(3)

Returns the dimension of the allocated cube.

Synopsis

long nodedim ( )

Description

Nodedim returns the dimension of the allocated cube. You cannot use *nodedim* to find out how many nodes an application is running on unless the number of nodes allocated is a power of 2. For example, if the number of nodes allocated is equal to 7, *nodedim* returns 3.

Example

```
long alphadim;
      .
      .
getcube ("alpha", "32", NULL, 0);
alphadim = nodedim ( );                /* Dimension of alpha is 5 */
```

Return Values

A value from 0 to 7 for applications running in 1 to 128 nodes. For example, *nodedim* returns 6 for a 64-node cube (or a 2^6 - node cube).

Errors

nodedim:no pid defined Use *setpid* to define a host process id.

See Also

getcube, numnodes

NUMNODES(3) iPSC/2 NUMNODES(3)

Obtain the number of nodes in the cube.

Synopsis

long numnodes( )

Description

Numnodes returns the number of nodes assigned to the application. For instance, if you allocate a cube of size D4 with *getcube*, *numnodes* will return **16**. Use *getcube* to assign a number of nodes to an application.

Example

```
long last_node, total_nodes;
   .
   .
getcube("alpha", "8m4", "saxon", 1);
getcube("beta", "32vx", "viking", 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. */
```

Return Values

The value returned by *numnodes* is in the range **1-128**.

Errors

numnodes: no pid defined	Use *setpid* to define a host process id.

See Also

getcube

RELCUBE(3) iPSC/2 RELCUBE(3)

Release a cube. This call is intended for use by host programs and is not available for node programs.

Synopsis

```
relcube( cubename )
char *cubename;
```

Description

cubename is a pointer to a character string which specifies 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. *Cubenames* cannot start with a dash ('-') or a space. 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.

Use *relcube* to release a cube that you allocated when you invoked the *getcube* command. It also kills the file server and any processes attached to *cubename* and any node process running on the cube.

Example

```
getcube("big", "128", NULL, 0);
   .
   .                    /* Clean up and release the cube. */

killcube(-1, -1);

relcube("big");
```

Return Values

None.

Errors

relcube: cubename does not exist	Use *cubeinfo* to determine cube names.
relcube: internal cube usage limit	Limit of 10 cubes may be allocated. Try again later when system is not so busy.
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

SETPID(3) iPSC/2 SETPID(3)

Sets the process id of a host process. This call is intended for use by host programs and is not available for node programs.

Synopsis

```
setpid( pid )
long pid;
```

Description

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.

Setpid sets 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 with *getcube* 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, the *pid* used previously is now valid to use again. 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 the *load* call.

Example

```
getcube("alpha", "16vx", NULL, 0);
getcube("beta", "4m8", "trainon", 1);
setpid(99);
   .
   .
attachcube("alpha");
setpid(0);
   .
   .
attachcube("beta");
setpid(99);
   .
   .
```

Return Values

None.

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 *bootcube*.
setpid: lifeline not responding	Internal system resource manager process error. Cube management processes are not responding. Try rebooting cube with *bootcube*.

See Also

csend, isend, mypid

SETSYSLOG(3) iPSC/2 SETSYSLOG(3)

Start the *syslog* program. This call is intended for use by host programs and is not available for node programs.

Synopsis

```
setsyslog( stdfd )
long stdfd;
```

Description

stdfd is an integer value that indicates where program 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

Syslog is a system program used to send the output of a host program to either the standard output or standard error of the current file server of the nodes. *Setsyslog* starts the *syslog* program and redirects the standard output of the calling program to the standard input of the *syslog* program. For more information on using *syslog*, refer to the iPSC/2 User's Guide.

Example

```
getcube("alpha", "8", NULL, 0);
setpid(99);
setsyslog (1);
load("prog1", -1, 1);
   .
   .
```

Return Values

None.

Errors

setsyslog: there is no attached cube Use *cubeinfo* to find out which cubes you own.

See Also

killsyslog, *syslog* command (Chapter 2)

WAITALL(3) iPSC/2 WAITALL(3)

Wait for all the specified processes to complete.

Synopsis

```
waitall( node, pid )
long node, pid;
```

Description

node is an integer value that identifies a particular node where a process resides or **-1** can be used to indicate all nodes. Nodes are numbered so that they range from **0** to (*numnodes* **-1**) where *numnodes* is the number of nodes in the partition. A host id is invalid.

pid is an integer value that specifies the process id or **-1** can be used to specify all processes. Valid pids include any integer value, but negative numbers other than **-1** are reserved for system programs. *Pid* is assigned when you *load* a process. It is not an error to wait for non-existent processes; *waitall* simply returns immediately.

Waitall waits for the selected process(es) to complete on the selected node(s). Unlike *waitone*, *waitall* 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.

Example

```
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. */
```

Return Values

None.

Errors

None.

See Also

waitone

WAITONE(3) iPSC/2 WAITONE(3)

Wait for a specified process to complete.

Synopsis

```
waitone( node, pid, cnode, cpid, ccode)
long node, pid, *cnode, *cpid, *ccode;
```

Description

node is an integer value that identifies a particular node where a process resides or **-1** to indicate all nodes. Nodes are numbered so that they range from **0** to (*numnodes* **-1**) where *numnodes* is the number of nodes in the partition. A host id is invalid.

pid 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. An error results 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* depends on how the process terminated. If the process terminated with a call to *exit* , *ccode* will contain the exit code (the argument to *exit* .) 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 one of the following codes. When a process is terminated due to an exception, its completion code is the exception type with **256** subtracted from it. Some of these exceptions (NMI interrupt, Double fault, and Invalid TSS) are not possible unless there is a system error.

Completion Codes:

-257	Process was killed
-256	Integer divide error exception
-255	Single-step interrupt
-254	NMI interrupt
-253	Breakpoint interrupt
-252	Integer overflow exception
-251	Array bounds exception
-250	Invalid opcode exception

Completion Codes (cont.)

-249	NPX unavailable exception
-248	Double fault
-247	Floating point address exception
-246	Invalid TSS exception
-245	Segment not present fault
-244	Stack overflow exception
-243	General protection fault
-242	Page fault
-241	Not used
-240	Floating point exception

Waitone waits for a process to complete on any of the nodes and provides 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 first node completes.

Example

```
long *cnode, *cpid, *ccode;
   .
   .
waitone(-1, -1, cnode, cpid, ccode);

printf("Process %d on node %d finished with error code %d\n", cpid,
       cnode, ccode);
```

Return Values

None.

Errors

None.

See Also

waitall, waitcube command (Chapter 2)

APPENDIX A

ERROR HANDLING

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 /usr/include/cube.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 errors follows. This list includes iPSC/2-specific errors and is located in the */usr/include/cube.h* include file. A list of system errors can be found in the */usr/include/sys/errno.h* include file.

Table A-1 (Page 1 of 4)
iPSC/2 Error Messages

Code	Name	Message	Corrective Action
110	EQDRVERR	Internal driver error	
111	EQPBUF	Invalid buffer pointer	Specify a pointer which contains the address of a valid data buffer.
112	EQBLEN	Buffer length exceeds allocation	Make sure length does not exceed buffer size.
113	EQLEN	Invalid length	Use a non-negative number or a length less than or equal to maximum message length.
114	EQTIME	Time limit exceeded	
115	EQMSGLONG	Received message too long for buffer	Make sure buffer is large enough to hold the message.
116	EQPID	Invalid pid	*Pid* must not be negative.
117	EQNODE	Invalid node	Use *numnodes* to determine cube size and *myhost* to determine host number.
118	EQTYPE	Invalid type	Use a non-negative number.
119	EQMID	Invalid message id	Use message id returned by *irecv* or *isend*.
120	EQHND	Invalid handler type	Select one of the types listed in the *handler* description.
121	EQNOPROC	Out of process slots	Use fewer processes
122	EQUSEPID	PID already in use	Select another pid.
123	EQNOACT	No active process	Use the *pid* of a loaded process.
124	EQBADFIL	Invalid object file	Specify a loadable file.
125	EQPARAM	Invalid parameter	
126	EQPFIL	Invalid file name pointer	
127	EQPCNODE	Invalid cnode pointer	

Table A-1 (Page 2 of 4)
iPSC/2 Error Messages

Code	Name	Message	Corrective Action
128	EQPCPID	Invalid cpid pointer	
129	EQPCCODE	Invalid ccode pointer	
130	EQPRIV	Privileged operation	
131	EQMEM	Not enough memory	Simplify application program.
132	EQINVREC	Invalid loader record	Internal error, try again. You may have to *bootcube*.
133	EQMSG	Invalid loader message	Internal error, try again. You may have to *bootcube*.
134	EQNOMID	Too many requests	Use *msgwait* for *isend* or *irecv* requests that are outstanding.
135	EQSET	Pid already set	Do not call *setpid* again.
136	EQNOSET	No pid defined	Use *setpid* to define process id.
137	EQCUBETABFULL	Internal cube usage limit	Comm server has run out of slots for subcube allocation. Try again later when cube is not so busy.
138	EQCUBEEXISTS	Cubename already exists	Use a different name.
139	EQCUBENOTEXIST	Cubename does not exist	Use *cubeinfo* to find out cube names.
140	EQCUBENOTATTCH	There is no attached cube	Use *cubeinfo* to find out which cubes you own.
141	EQUSOCK	Comm server out of UNIX sockets	
142	EQNETSOCK	Comm server out of net sockets	
143	EQTTY	Comm server out of ttys	Limit of 10 cubes may be allocated. Try again later when the system is not as busy.
144	EQNOCUBES	There are no cubes allocated	Use *getcube* to allocate a cube.

Table A-1 (Page 3 of 4)
iPSC/2 Error Messages

Code	Name	Message	Corrective Action
145	EQCUBENAMELEN	Cubename longer than 14 characters.	
146	EQHOSTNAMELEN	Hostname longer than 14 characters	
147	EQTYPENAMELEN	Cubetype longer than 5 characters	
148	EQBADGLOBAL	Global value invalid	
149	EQRCSBUSY	Remote comm server busy, try again later.	
150	EQINCMPREAD	Did not read header specified bytes	
151	EQNOCUBE	Cubetype not found	Could not find a cube of the type you requested.
152	EQLIFEBUSY	Internal cube usage limit	
153	EQNOCOMMSER	Commser not responding	Internal system resource manager rpocess error. Cube management process are not responding. Try rebooting cube with *bootcube*.
154	EQUSM	Invalid diagnostic channel usm id	
155	EQDIM	Invalid dimension	
156	EQMODE	Invalid diagnostic channel mode	
157	EQSTATUS	Invalid diagnostic channel status	
158	EQNOLL	Lifeline not responding	Internal system resource manager rpocess error. Cube management process are not responding. Try rebooting cube with *bootcube*.

Table A-1 (Page 4 of 4)
iPSC/2 Error Messages

Code	Name	Message	Corrective Action
159	EQBADNODE	Node not responding	
160	EQUSEVX	Vector extension is in use by another process	
161	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.
162	EQMSGSHORT	Received message too short for buffer	

APPENDIX B

TYPESEL MASK

The *typesel* parameter 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. *Typesel* (*type* selector) allows you to select a specific message *type* or a set of message *types* based on a **32**-bit mask. *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 *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 Table B-1). 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);

The following table gives the hexadecimal number associated with bits 0-31:

Table B-1
Hexadecimal Number Representation

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
All other types	0x40000000

APPENDIX C
C COMMUNICATION UTILITIES

The communication primitives in the basic iPSC/2 operating system provide for sending a message from one single process to another. We have found it useful to expand these facilities to provide some higher level constructs for interprocess communication. Included are:

Global Commutative Operations

These operations accumulate data from the entire allocated cube. All the participants (nodes) call the same routine (with different data values) for a specific operation and the final result is distributed to all participants.

Ten operations are provided: addition, multiplication, maximum, minimum, bitwise inclusive-OR, bitwise AND, bitwise exclusive-OR, logical-OR, logical-AND, logical exclusive-OR. The operation is applied component-wise to each vector element.

For addition, multiplication, maximum and minimum, three different data types are supported. They are integer, real, and double precision data types. The integer data type applies to the bitwise operations. The logical data type applies to the logical operations.

For convenience, the second letter of each routine name indicates the data type and the word following it is the type of operator. For example, 'gisum' means global integer summation (or addition). For both C and FORTRAN, the naming conventions are the same.

The parameter types are:

I	integer
S	real
D	double precision
L	logical (true or false)

The operators are:

SUM	add the components
PROD	multiply the components
HIGH	maximum element
LOW	minimum element
OR	bitwise/logical inclusive-OR
AND	bitwise/logical AND
XOR	bitwise/logical exclusive-OR

Global Concatenation

The 'gcol' routine collects (concatenates) a contribution from each node in the allocated cube in node number order. At the end, each node has the same collected vector.

Global Synchronization

Any node calling 'gsync' will wait until all nodes in the allocated cube have called 'gsync'.

Global Function

'Gopf' allows you to supply a commutative function to be used in place of a particular operation in a global commutative operation.

Global send to specified nodes

In some applications, a node may have a message which must be sent to a specified subset of the other nodes. The 'gsendx' routine is provided for this purpose. It sends data to each node from a given node list.

The iPSC/2 send and receive calls use both a node number and a process id to specify the destination of a message. This allows multiple processes to execute on a node. However, the utilities described here assume only one participating process per node. For simplicity, it is assumed that each participating process has the same PID. For this reason, the PID is not included in the calls. It is obtained internally by a call to MYPID().

It is also assumed that the number of nodes in the allocated cube is a power of 2.

Refer to Chapter 3 of this manual for a detailed description of the iPSC/2 routines.

RESERVED MESSAGE TYPES

Message types 2000000001 and above are reserved for the system. Using a reserved message type can produce unpredictable results.

COMPILATION AND LINKING

The communication routines are included in the 'libnode.a' library. They are accessed from the 'cc' command with the -node option. They are for node programs only.

EXAMPLE PROBLEM

Suppose a vector of length **N** is distributed among the nodes of the cube. Let **X(M)** be a piece of the vector on this node. **M** need not have the same value on all nodes. Assume that this vector must be normalized and then the entire vector must be available to all nodes. The following code fragment will accomplish these tasks.

```
        int I, MYNODE, 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
 *  Assume there exists a subroutine called dsqrt
 */
          NORM = dsqrt(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);
```

The array **Y** in each node contains all of the vector **X**. The order of **X's** in **Y** is by increasing node number.

Note that the naive 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(3) iPSC/2 GCOL(3)

Global concatenation operation.

Synopsis

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

Description

x pointer to the input buffer to be used in the operation.

xlen the length of input buffer x (in bytes).

y pointer to the output buffer to be used in the operation. y contains the desired result.

ylen the length of output buffer y (in bytes).

ncnt pointer to the number of bytes returned in y

Gcol globally collects (concatenates) a contribution from each node and insures that contributions are received in order. The x and y parameters can be any data type but should be the same data type. The result is returned in y to every node. All participating processes must have the same PID.

Errors

gcol:received message too long for buffer	Make sure y 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.

Global vector double precision MAX operation.

Synopsis

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

Description

x pointer to the input vector to be used in the operation. When the operation completes, it contains the final result.

n the length of the vector x.

work pointer to an array which is used to receive the other contributions. The length of work must be at least n.

Gdhigh calculates the maximum value of each double precision component of x across all nodes . The result is returned in x to every node. All participating processes must have the same PID.

Errors

gdhigh:received message too long for buffer	Make sure vector length is the same for all nodes and is exactly the length of the result.
gdhigh:received message too short for buffer	Make sure vector length is the same for all nodes and is exactly the length of the result.
gdhigh:invalid buffer pointer	Specify a pointer which contains the address of a valid vector.
gdhigh:invalid length	Use a non-negative number for n.
gdhigh:buffer length exceeds allocation	Make sure vector is large enough.

Global vector double precision MIN operation.

Synopsis

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

Description

x pointer to the input vector to be used in the operation. When the operation completes, it contains the final result.

n the length of the vector x.

work pointer to an array which is used to receive the other contributions. The length of work must be at least n.

Gdlow calculates the minimum value of each double precision component of x across all nodes . The result is returned in x to every node. All participating processes must have the same PID.

Errors

gdlow:received message too long for buffer	Make sure vector length is the same for all nodes.
gdlow:received message too short for buffer	Make sure vector length is the same for all nodes.
gdlow:invalid buffer pointer	Specify a pointer which contains the address of a valid vector.
gdlow:invalid length	Use a non-negative number for n.
gdlow:buffer length exceeds allocation	Make sure vector is large enough.

GDPROD(3) iPSC/2 GDPROD(3)

Global vector double precision MULTIPLY operation.

Synopsis

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

Description

x pointer to the input vector to be used in the operation. When the operation completes, it contains the final result.

n the length of the vector x.

work pointer to an array which is used to receive the other contributions. The length of work must be at least n.

Gdprod calculates the product of each double precision component of x across all nodes . The result is returned in x to every node. All participating processes must have the same PID.

Errors

gdprod:received message too long for buffer	Make sure vector length is the same for all nodes.
gdprod:received message too short for buffer	Make sure vector length is the same for all nodes.
gdprod:invalid buffer pointer	Specify a pointer which contains the address of a valid vector.
gdprod:invalid length	Use a non-negative number for n.
gdprod:buffer length exceeds allocation	Make sure vector is large enough.

GDSUM(3) iPSC/2 GDSUM(3)

Global vector double precision SUM operation.

Synopsis

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

Description

x pointer to the input vector to be used in the operation. When the operation completes, it contains the final result.

n the length of the vector x.

work pointer to an array which is used to receive the other contributions. The length of work must be at least n.

Gdsum calculates the sum of each double precision component of x across all nodes . The result is returned in x to every node. All participating processes must have the same PID.

Errors

gdsum:received message too long for buffer	Make sure vector length is the same for all nodes.
gdsum:received message too short for buffer	Make sure vector length is the same for all nodes.
gdsum:invalid buffer pointer	Specify a pointer which contains the address of a valid vector.
gdsum:invalid length	Use a non-negative number for n.
gdsum:buffer length exceeds allocation	Make sure vector is large enough.

GIAND(3) iPSC/2 GIAND(3)

Global vector integer bitwise AND operation.

Synopsis

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

Description

x pointer to the input vector to be used in the operation. When the operation completes, it contains the final result.

n the length of the vector x.

work pointer to an array which is used to receive the other contributions. The length of work must be at least n.

Giand calculates the bitwise AND of each integer component of x across all nodes . The result is returned in x to every node. All participating processes must have the same PID.

Errors

giand:received message too long for buffer	Make sure vector length is the same for all nodes.
giand:received message too short for buffer	Make sure vector length is the same for all nodes.
giand:invalid buffer pointer	Specify a pointer which contains the address of a valid vector.
giand:invalid length	Use a non-negative number for n.
giand:buffer length exceeds allocation	Make sure vector is large enough.

GIHIGH(3) iPSC/2 GIHIGH(3)

Global vector integer MAX operation.

Synopsis

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

Description

x pointer to the input vector to be used in the operation. When the operation completes, it contains the final result.

n the length of the vector x.

work pointer to an array which is used to receive the other contributions. The length of work must be at least n.

Gihigh calculates the maximum value of each integer component of x across all nodes . The result is returned in x to every node. All participating processes must have the same PID.

Errors

gihigh:received message too long for buffer	Make sure vector length is the same for all nodes.
gihigh:received message too short for buffer	Make sure vector length is the same for all nodes.
gihigh:invalid buffer pointer	Specify a pointer which contains the address of a valid vector.
gihigh:invalid length	Use a non-negative number for n.
gihigh:buffer length exceeds allocation	Make sure vector is large enough.

GILOW(3) iPSC/2 GILOW(3)

Global vector integer MIN operation.

Synopsis

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

Description

x pointer to the input vector to be used in the operation. When the operation completes, it contains the final result.

n the length of the vector x.

work pointer to an array which is used to receive the other contributions. The length of work must be at least n.

Gilow calculates the minimum value of each integer component of x across all nodes . The result is returned in x to every node. All participating processes must have the same PID.

Errors

gilow:received message too long for buffer	Make sure vector length is the same for all nodes.
gilow:received message too short for buffer	Make sure vector length is the same for all nodes.
gilow:invalid buffer pointer	Specify a pointer which contains the address of a valid vector.
gilow:invalid length	Use a non-negative number for n.
gilow:buffer length exceeds allocation	Make sure vector is large enough.

GIOR(3) iPSC/2 GIOR(3)

Global vector integer bitwise OR operation.

Synopsis

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

Description

x pointer to the input vector to be used in the operation. When the operation completes, it contains the final result.

n the length of the vector x.

work pointer to an array which is used to receive the other contributions. The length of work must be at least n.

Gior calculates the bitwise OR of each integer component of x across all nodes . The result is returned in x to every node. All participating processes must have the same PID.

Errors

gior:received message too long for buffer	Make sure vector length is the same for all nodes.
gior:received message too short for buffer	Make sure vector length is the same for all nodes.
gior:invalid buffer pointer	Specify a pointer which contains the address of a valid vector.
gior:invalid length	Use a non-negative number for n.
gior:buffer length exceeds allocation	Make sure vector is large enough.

GIPROD(3) iPSC/2 GIPROD(3)

Global vector integer MULTIPLY operation.

Synopsis

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

Description

x pointer to the input vector to be used in the operation. When the operation completes, it contains the final result.

n the length of the vector x.

work pointer to an array which is used to receive the other contributions. The length of work must be at least n.

Giprod calculates the product of each integer component of x across all nodes . The result is returned in x to every node. All participating processes must have the same PID.

Errors

giprod:received message too long for buffer	Make sure vector length is the same for all nodes.
giprod:received message too short for buffer	Make sure vector length is the same for all nodes.
giprod:invalid buffer pointer	Specify a pointer which contains the address of a valid vector.
giprod:invalid length	Use a non-negative number for n.
giprod:buffer length exceeds allocation	Make sure vector is large enough.

GISUM(3) iPSC/2 GISUM(3)

Global vector integer SUM operation.

Synopsis

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

Description

x pointer to the input vector to be used in the operation. When the operation completes, it contains the final result.

n the length of the vector x.

work pointer to an array which is used to receive the other contributions. The length of work must be at least n.

Gisum calculates the sum of each integer component of x across all nodes . The result is returned in x to every node. All participating processes must have the same PID.

Errors

gisum:received message too long for buffer	Make sure vector length is the same for all nodes.
gisum:received message too short for buffer	Make sure vector length is the same for all nodes.
gisum:invalid buffer pointer	Specify a pointer which contains the address of a valid vector.
gisum:invalid length	Use a non-negative number for n.
gisum:buffer length exceeds allocation	Make sure vector is large enough.

GIXOR(3) iPSC/2 GIXOR(3)

Global vector integer bitwise exclusive OR operation.

Synopsis

```
gixor(x, n, work)
long x[];
long n;
long work[];
```

Description

x pointer to the input vector to be used in the operation. When the operation completes, it contains the final result.

n the length of the vector x.

work pointer to an array which is used to receive the other contributions. The length of work must be at least n.

Gixor calculates the bitwise exclusive OR of each integer component of x across all nodes . The result is returned in x to every node. All participating processes must have the same PID.

Errors

gixor:received message too long for buffer	Make sure vector length is the same for all nodes.
gixor:received message too short for buffer	Make sure vector length is the same for all nodes.
gixor:invalid buffer pointer	Specify a pointer which contains the address of a valid vector.
gixor:invalid length	Use a non-negative number for n.
gixor:buffer length exceeds allocation	Make sure vector is large enough.

GLAND(3) iPSC/2 GLAND(3)

Global vector logical AND operation.

Synopsis

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

Description

x pointer to the input vector to be used in the operation. When the operation completes, it contains the final result.

n the length of the vector x.

work pointer to an array which is used to receive the other contributions. The length of work must be at least n.

Gland calculates the bitwise AND of each logical component of x across all nodes . The result is returned in x to every node. All participating processes must have the same PID.

Errors

gland:received message too long for buffer	Make sure vector length is the same for all nodes.
gland:received message too short for buffer	Make sure vector length is the same for all nodes.
gland:invalid buffer pointer	Specify a pointer which contains the address of a valid vector.
gland:invalid length	Use a non-negative number for n.
gland:buffer length exceeds allocation	Make sure vector is large enough.

Global vector logical inclusive OR operation.

Synopsis

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

Description

x pointer to the input vector to be used in the operation. When the operation completes, it contains the final result.

n the length of the vector x.

work pointer to an array which is used to receive the other contributions. The length of work must be at least n.

Glor calculates the bitwise OR of each logical component of x across all nodes . The result is returned in x to every node. All participating processes must have the same PID.

Errors

glor:received message too long for buffer	Make sure vector length is the same for all nodes.
glor:received message too short for buffer	Make sure vector length is the same for all nodes.
glor:invalid buffer pointer	Specify a pointer which contains the address of a valid vector.
glor:invalid length	Use a non-negative number for n.
glor:buffer length exceeds allocation	Make sure vector is large enough.

GLXOR(3) iPSC/2 GLXOR(3)

Global vector logical exclusive OR operation.

Synopsis

```
glxor(x, n, work)
long x[];
long n;
long work[];
```

Description

x pointer to the input vector to be used in the operation. When the operation completes, it contains the final result.

n the length of the vector x.

work pointer to an array which is used to receive the other contributions. The length of work must be at least n.

Glxor calculates the bitwise exclusive OR of each logical component of x across all nodes . The result is returned in x to every node. All participating processes must have the same PID.

Errors

glxor:received message too long for buffer	Make sure vector length is the same for all nodes.
glxor:received message too short for buffer	Make sure vector length is the same for all nodes.
glxor:invalid buffer pointer	Specify a pointer which contains the address of a valid vector.
glxor:invalid length	Use a non-negative number for n.
glxor:buffer length exceeds allocation	Make sure vector is large enough.

Arbitrary commutative function.

Synopsis

```
gopf(x, n, work, f)
long x[];
long n;
long work[];
long (*f)();
```

Description

x pointer to the input vector to be used in the operation. When the operation completes, it contains the final result. x can be any type.

n the length of the vector x.

work pointer to an array which is used to receive the other contributions. The length of work must be at least n.

f the function to be called. It is called as f(x, n, work).
f(x, n, work) sets x to whatever function of x and work is desired.

Gopf applies f() to combine the vectors x from all nodes. The result is returned in x to all nodes. The function f() must be a commutative function of two vectors, returning its result in the first vector. All participating processes must have the same PID. An example follows.

Errors

None.

EXAMPLE

If a matrix is distributed by rows among the processors and Gaussian elimination with column pivoting is implemented, it is necessary to find out which element of a distributed vector is the largest (in absolute value). The following code fragment implements this calculation:

```
extern max_node();
struct PIVOT_NODE {
        float max;
        long  node;
};
struct PIVOT_NODE mine, work;
int i, max_loc;
/*
 * Assume n has been initialized
 */
float x[n];

mine.max = abs(x[0]);
mine.node = mynode();
max_loc = 0;
/*
 * Find the maximum and save the index of the maximum
 */
for (i=1; i < n; i++){
        if(mine.max < abs(x[i])){
                mine.max = abs(x[i]);
                max_loc = i;
        }
}
gopf(&mine, n, &work, max_node);
if (mine.node == mynode())
        /* I have the maximum */
...
return;
```

Let max_node be a function which finds the maximum among rows of a matrix and its associate owner(node) of the row.

```
max_node(mine, n, work)
        int n;
        struct PIVOT_NODE mine;
        struct PIVOT_NODE work;
{
        if((mine.max < work.max) ||
                ((mine.max==work.max) && (mine.node > work.node))) {
                mine.max = work.max;
                mine.node = work.node;
        }
        return;
}
```

GSENDX(3) iPSC/2 GSEND(3)

Send a vector to a list of nodes.

Synopsis

```
gsendx(type, x, len, nodenums, nlen)
long type;
long x[];
long len;
long nodenums[];
long nlen;
```

Description

type	message type. Must be the same for all participating processes. There must be no other messages of this type in the system.
x	pointer to the input vector to be sent.
len	the number of bytes of data in vector.
nodenums	pointer to the list of node numbers data is to be sent to.
nlen	the number of node numbers in the list.

This subroutine sends a vector to a set of nodes specified by a node list. All the nodes which are to receive the vector must call crecv, irecv, or hrecv. All participating processes must have the same PID.

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.

GSHIGH(3) iPSC/2 GSHIGH(3)

Global vector real MAX operation.

Synopsis

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

Description

x pointer to the input vector to be used in the operation. When the operation completes, it contains the final result.

n the length of the vector x.

work pointer to an array which is used to receive the other contributions. The length of work must be at least n.

Gshigh calculates the maximum value of each real component of x across all nodes . The result is returned in x to every node. All participating processes must have the same PID.

Errors

gshigh:received message too long for buffer	Make sure vector length is the same for all nodes.
gshigh:received message too short for buffer	Make sure vector length is the same for all nodes.
gshigh:invalid buffer pointer	Specify a pointer which contains the address of a valid vector.
gshigh:invalid length	Use a non-negative number for n.
gshigh:buffer length exceeds allocation	Make sure vector is large enough.

Global vector real MIN operation.

Synopsis

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

Description

x pointer to the input vector to be used in the operation. When the operation completes, it contains the final result.

n the length of the vector x.

work pointer to an array which is used to receive the other contributions. The length of work must be at least n.

Gslow calculates the minimum value of each real component of x across all nodes . The result is returned in x to every node. All participating processes must have the same PID.

Errors

gslow:received message too long for buffer	Make sure vector length is the same for all nodes.
gslow:received message too short for buffer	Make sure vector length is the same for all nodes.
gslow:invalid buffer pointer	Specify a pointer which contains the address of a valid vector.
gslow:invalid length	Use a non-negative number for n.
gslow:buffer length exceeds allocation	Make sure vector is large enough.

GSPROD(3) iPSC/2 GSPROD(3)

Global vector real multiply operation.

Synopsis

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

Description

x pointer to the input vector to be used in the operation. When the operation completes, it contains the final result.

n the length of the vector x.

work pointer to an array which is used to receive the other contributions. The length of work must be at least n.

Gsprod calculates the product of each real component of x across all nodes . The result is returned in x to every node. All participating processes must have the same PID.

Errors

gsprod:received message too long for buffer	Make sure vector length is the same for all nodes.
gsprod:received message too short for buffer	Make sure vector length is the same for all nodes.
gsprod:invalid buffer pointer	Specify a pointer which contains the address of a valid vector.
gsprod:invalid length	Use a non-negative number for n.
gsprod:buffer length exceeds allocation	Make sure vector is large enough.

GSSUM(3) iPSC/2 GSSUM(3)

Global vector real SUM operation.

Synopsis

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

Description

x pointer to the input vector to be used in the operation. When the operation completes, it contains the final result.

n the length of the vector x.

work pointer to an array which is used to receive the other contributions. The length of work must be at least n.

Gssum calculates the sum of each real component of x across all nodes. The result is returned in x to every node. All participating processes must have the same PID.

Errors

gssum:received message too long for buffer	Make sure vector length is the same for all nodes.
gssum:received message too short for buffer	Make sure vector length is the same for all nodes.
gssum:invalid buffer pointer	Specify a pointer which contains the address of a valid vector.
gssum:invalid length	Use a non-negative number for n.
gssum:buffer length exceeds allocation	Make sure vector is large enough.

Global synchronization operation.

Synopsis

gsync()

Description

Gsync synchronizes node processes by waiting until all other nodes have called gsync. All participating processes must have the same PID.

Errors

None.