August, 1988
Order Number: 311019-003

iPSC®/2 FORTRAN PROGRAMMER'S REFERENCE MANUAL

intel Corporation

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

CHAPTER 1 - INTRODUCTION

CHAPTER 2 - iPSC/2 COMMANDS

CHAPTER 3 - iPSC/2 FORTRAN ROUTINES

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

CHAPTER 4 - iPSC/2 VECLIB ROUTINES

CHAPTER 4 - iPSC/2 VECLIB ROUTINES (cont.)

CHAPTER 4 - iPSC/2 VECLIB ROUTINES (cont.)

TABLES

CHAPTER 1

INTRODUCTION

PURPOSE

The purpose of this manual is to provide information for all the FORTRAN 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/fortran*" 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 FORTRAN Routines	These are program calls for the two FORTRAN 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.
4	iPSC/2 VecLib Routines	These routines provide a basic set of vector operations which can be used to replace FORTRAN DO loops.
A	TYPESEL Mask	Gives the procedure for building a TYPESEL mask which is used to select messages.
B	FORTRAN 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.
iPSC/2 Green Hills FORTRAN Language Reference Manual	Describes the FORTRAN compiler for the 80386. It explains language features and extensions that are supported, as well as various restrictions. It also describes the calling conventions, register and memory allocation strategies, and program optimization.

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. Names longer than 15 characters are truncated to 15 characters. *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 only 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`

ATTACHCUBE(1) (cont.) iPSC/2 ATTACHCUBE(1) (cont.)

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 SRM's. 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 currenct 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

CUBEINFO(1) (cont.) iPSC/2 CUBEINFO(1) (cont.)

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 TTYs 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 entire network are allocated	**cubeinfo -s**

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]

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. Names longer than 15 characters are truncated to 15 characters. *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) are 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). Strings must be 15 characters or less. Strings longer than 15 characters are truncated to 15 characters.

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 allocated will always be a power of two. If the size you specify is not a power of two, it will be rounded up.

GETCUBE(1) (cont.) iPSC/2 GETCUBE(1) (cont.)

The choices for type are as follows:

nothing	gets cube of any type.
vx	vector
sx	nodes with SX installed
m1	nodes with at least 1M byte of memory each
m4	nodes with at least 4M bytes of memory each
m8	nodes with at least 8M bytes of memory each
m16	nodes with at least 16M bytes of memory each
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. One possible, 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* after *getcube*.

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(1) (cont.) iPSC/2 GETCUBE(1) (cont.)

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 4vx**
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`

LOAD(1) (cont.) iPSC/2 LOAD(1)(cont.)

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. Names longer than 15 characters are truncated to 15 characters. *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`

NEWSERVER(1) (cont.) iPSC/2 NEWSERVER(1) (cont.)

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 system resource manager (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 system resource manager. 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.

iPSC/2 **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, it is recommended that you do not try running iPSC programs on a remote development machine as they 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. Names longer than 15 characters are truncated to 15 characters. *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`

RELCUBE(1) (cont.) iPSC/2 RELCUBE(1) (cont.)

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 insure 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 FORTRAN ROUTINES

INTRODUCTION

This chapter documents each of the iPSC/2 routines in the two FORTRAN system interface libraries: host and node. These routines allow you to perform such tasks as accessing and releasing cubes, loading, starting, and killing processes, and passing messages between processes.

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

The routines are listed alphabetically. 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
- return values where applicable
- error messages returned by this routine
- list of related routines

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

Routine	Host/ Node	Synopsis	Description
ATTACHCUBE	host	SUBROUTINE ATTACHCUBE(CUBENAME) CHARACTER CUBENAME*(*)	Attach to a cube and make it the current cube.
CPROBE	host/node	SUBROUTINE CPROBE(TYPESEL) INTEGER TYPESEL	Wait for a message to arrive.
CRECV	host/node	SUBROUTINE CRECV(TYPESEL,BUF,LEN) INTEGER TYPESEL INTEGER BUF(*) INTEGER LEN	Receive a message and wait for completion.
CSEND	host/node	SUBROUTINE CSEND(TYPE,BUF,LEN, NODE, PID) INTEGER TYPE INTEGER BUF(*) INTEGER LEN, NODE, PID	Send a message and wait for completion.
CTOHD HTOCD CTOHF HTOCF CTOHL HTOCL CTOHS HTOCS	host	SUBROUTINE CTOHD(SV,N) SUBROUTINE HTOCD(SV,N) SUBROUTINE CTOHF(SV,N) SUBROUTINE HTOCF(SV,N) SUBROUTINE CTOHL(SV,N) SUBROUTINE HTOCL(SV,N) SUBROUTINE CTOHS(SV,N) SUBROUTINE HTOCS(SV,N) INTEGER SV INTEGER N	Convert node byte order to remote development machine byte order (or remote development machine to node byte order).
CUBEINFO	host	INTEGER FUNCTION CUBEINFO (CT, NUMSLOTS,GLOBAL) CHARACTER*16 CT (SLOTSIZE,NUMSLOTS) INTEGER NUMSLOTS, GLOBAL	Obtain information about allocated cubes.
FLICK	host/node	SUBROUTINE FLICK( )	Relinquish CPU to another process.
FLUSHMSG	host/node	SUBROUTINE FLUSHMSG(TYPESEL, NODE, PID) INTEGER TYPESEL, NODE, PID	Flush specified messages from the system.

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

Routine	Host/ Node	Synopsis	Description
GETCUBE	host	SUBROUTINE GETCUBE (CUBENAME,CUBETYPE, SRMNAME, KEEP) CHARACTER CUBENAME*(*) CHARACTER CUBETYPE*(*) CHARACTER SRMNAME*(*) INTEGER KEEP	Allocate a cube.
GINV	host/node	INTEGER FUNCTION GINV(J) INTEGER J	Returns the inverse of GRAY.
GRAY	host/node	INTEGER FUNCTION GRAY(J) INTEGER J	Returns the binary-reflected Gray code for an integer.
HANDLER	node	EXTERNAL IFUNC SUBROUTINE HANDLER (TYPE, IFUNC) INTEGER TYPE	Provide user-written exception handler.
HRECV	node	INTEGER FUNCTION HRECV (TYPESEL, BUF, LEN, PROC) INTEGER TYPESEL INTEGER BUF(*) INTEGER LEN EXTERNAL PROC	Provide user-written exception handler for receive traps.
INFOCOUNT INFONODE INFOPID INFOTYPE	host/node	INTEGER FUNCTION INFOCOUNT( ) INTEGER FUNCTION INFONODE( ) INTEGER FUNCTION INFOPID( ) INTEGER FUNCTION INFOTYPE( )	Returns information about a pending or received message.
IPROBE	host/node	INTEGER FUNCTION IPROBE(TYPESEL) INTEGER TYPESEL	Determine whether a message of a selected type is pending.
IRECV	host/node	INTEGER FUNCTION IRECV (TYPESEL, BUF, LEN) INTEGER TYPESEL INTEGER BUF(*) INTEGER LEN	Receive a message.

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

Routine	Host/ Node	Synopsis	Description
ISEND	host/node	INTEGER FUNCTION ISEND(TYPE, BUF, LEN, NODE, PID) INTEGER TYPE INTEGER BUF(*) INTEGER LEN, NODE, PID	Send a message.
KILLCUBE	host/node	SUBROUTINE KILLCUBE(NODE, PID) INTEGER NODE, PID	Terminate and clear out a process(es).
KILLPROC	host/node	SUBROUTINE KILLPROC(NODE, PID) INTEGER NODE, PID	Terminate a process.
KILLSYSLOG	host	SUBROUTINE KILLSYSLOG( )	Terminate a SYSLOG process.
LED	node	SUBROUTINE LED(LSTATE) INTEGER LSTATE	Turn the node board's green LED on or off.
LOAD	host/node	SUBROUTINE LOAD(FILENAME, NODE, PID) CHARACTER FILENAME*(*) INTEGER NODE, PID	Load a node process.
MASKTRAP	node	INTEGER FUNCTION MASKTRAP (MASK) INTEGER MASK	Enable or disable a mask trap.
MCLOCK	host/node	INTEGER FUNCTION MCLOCK( )	Return the time.
MSGCANCEL	host/node	SUBROUTINE MSGCANCEL(ID) INTEGER ID	Cancel a send or receive operation.
MSGDONE	host/node	INTEGER FUNCTION MSGDONE(ID) INTEGER ID	Determine whether a send or receive operation has completed.
MSGWAIT	host/node	SUBROUTINE MSGWAIT(ID) INTEGER ID	Wait for completion of a send or receive operation.
MYHOST	host/node	INTEGER FUNCTION MYHOST( )	Obtain node id of the host machine.

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

Routine	Host/ Node	Synopsis	Description
MYNODE	host/node	INTEGER FUNCTION MYNODE( )	Obtain the node id of the calling process.
MYPID	host/node	INTEGER FUNCTION MYPID( )	Obtain process id of the calling process.
NEWSERVER	host	SUBROUTINE NEWSERVER (CUBENAME) CHARACTER CUBENAME*(*)	Start a new file server for the specified cube.
NODEDIM	host/node	INTEGER FUNCTION NODEDIM( )	Returns the dimension of the allocated cube.
NUMNODES	host/node	INTEGER FUNCTION NUMNODES( )	Obtain the number of nodes in the cube.
RELCUBE	host	SUBROUTINE RELCUBE(CUBENAME) CHARACTER CUBENAME*(*)	Release a cube.
SETPID	host	SUBROUTINE SETPID(PID) INTEGER PID	Sets process id of a host process.
SETSYSLOG	host	SUBROUTINE SETSYSLOG(STDFD) INTEGER STDFD	Start the SYSLOG program.
WAITALL	host/node	SUBROUTINE WAITALL(NODE, PID) INTEGER NODE, PID	Wait for all the specified processes to complete.
WAITONE	host/node	SUBROUTINE WAITONE(NODE, PID, CNODE, CPID, CCODE) INTEGER NODE, PID, 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

```
SUBROUTINE ATTACHCUBE( CUBENAME)
CHARACTER CUBENAME*(*)
```

Description

CUBENAME	is 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 argument is blank, 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.

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

Example

```
      CHARACTER*15 CUBE1, CUBE2
              .
              .
      CUBE1 = 'ALPHA'
      CUBE2 = 'BETA'

      CALL GETCUBE(CUBE1, '32', ' ', 0)
      CALL GETCUBE(CUBE2, '8VX', 'TRAINON', 1)
C     THE CURRENT CUBE IS 'BETA'
              .
              .
      CALL ATTACHCUBE(CUBE1)
C     THE CURRENT CUBE IS NOW 'ALPHA'
              .
              .
      CALL ATTACHCUBE (CUBE2)
C     THE CURRENT CUBE IS NOW 'BETA' AGAIN
```

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

SUBROUTINE CPROBE(TYPESEL)
INTEGER 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 TYPE's 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 TYPE's will be recognized. Refer to Appendix A 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

```
      PARAMETER(ITYPE=10)

      INTEGER MSGID
      CHARACTER MSGBUF*80
          .
          .
C     WAIT FOR A TYPE 10 MESSAGE TO ARRIVE, THEN INITIATE A MESSAGE
C     RECEIVE.

      CALL CPROBE(ITYPE)
      IF (INFOCOUNT().LE.LEN(MSGBUF)) THEN
          CRECV (ITYPE, MSGBUF, LEN(MSGBUF))
      ENDIF
```

Return Values

None.

Errors

CPROBE: no pid defined	Use SETPID to define a host process id.
CPROBE: too many requests	Use MSGWAIT or MSGDONE 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

```
SUBROUTINE CRECV(TYPESEL, BUF, LEN)
INTEGER TYPESEL
INTEGER BUF(*)
INTEGER 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 TYPE's 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 TYPE's will be recognized. Refer to Appendix A for information on building a TYPESEL mask.

BUF

is the buffer where the received message will be stored. Valid entries include any legal FORTRAN data type. It is recommended that data types match in send and receive operations.

LEN

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

CRECV initiates the receipt of a message. The CRECV call waits for a message whose TYPE matches the TYPE(s) specified in the TYPESEL argument. 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 subroutine 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

```
      PARAMETER(BUFLEN1=256)

      CHARACTER BUF1*BUFLEN1

C     RECEIVE THE FIRST MESSAGE TO ARRIVE AND STORE IT IN 'BUF1'.

      CALL CRECV(-1, BUF1, BUFLEN1)
```

Return Values

None.

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

Errors

CRECV: no pid defined	Use SETPID to define a host process id.
CRECV: invalid length	Use a non-negative number or a length less than or equal to maximum message length.
CRECV: too many requests	Use MSGWAIT or MSGDONE 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 valid data buffer.
CRECV:invalid node	Use NUMNODES to determine cube size and MYHOST to determine host number.
CRECV:invalid type	Use a non-negative number.
CRECV:received message too long for buffer	Make sure the buffer is large enough to hold the message.

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

```
SUBROUTINE CSEND(TYPE, BUF, LEN, NODE, PID)
INTEGER TYPE
INTEGER BUF(*)
INTEGER 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 buffer which contains the message to be sent. Valid entries include any legal FORTRAN 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

```
      CHARACTER MSG*100
      INTEGER MSGLEN

      MSG = 'HELLO NODE 0'
      MSGLEN = LEN(MSG)

C     SEND A TYPE 5 MESSAGE TO PROCESS 1 ON NODE 0.

      CALL CSEND(5, MSG, MSGLEN, 0, 1)

C     RE-USE THE MESSAGE BUFFER AND SEND THE NEW TYPE 2 MESSAGE
C     TO HOST PROCESS 10.

      MSG = 'HELLO HOST'
      CALL CSEND(2, MSG, MSGLEN, MYHOST(), 10)
```

Return Value

None.

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

Errors

CSEND: no pid defined	Use SETPID to define a host process id.
CSEND: invalid type	Use a non-negative number.
CSEND: invalid length	Use a non-negative number or a length less than or equal to maximum message length.
CSEND: invalid node	Use NUMNODES to determine cube size or MYHOST to determine host number.
CSEND: buffer length exceeds allocation	Make sure length does not exceed buffer size.
CSEND: invalid buffer pointer	Specify 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 development machine byte order or remote development machine 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

INTEGER SV
INTEGER N

Description

SV starting address of swap variable(s). Individual variables of any type or arrays can be used. When you want to swap an entire array, use **N** to specify number elements and **SV** as the array starting element.

N number of **SV's (elements)** 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. 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 in both SRM and remote development machine libraries but 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.

iPSC/2

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

Example

```
      INTEGER DATAIN(100)
      INTEGER DATAOUT(100)
      INTEGER NODETYPE, NODEPID
           .
           .
C     SWAP BYTES TO NODE ORDER BEFORE SENDING MESSAGE TO THE NODES

      CALL HTOCL(DATAIN,100)
      CALL CSEND(NODETYPE, DATAIN, LEN(DATAIN), -1 NODEPID)
           .
           .
      CALL CRECV(NODETYPE, DATAOUT, LEN(DATAOUT) )

C     SWAP BYTES BACK TO HOST ORDER BEFORE USING THE CONTENTS OF THE MESSAGE
C     BUFFER

      CALL CTOHL(DATAOUT,100)
           .
           .
      END
```

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

```
INTEGER FUNCTION CUBEINFO( CT, NUMSLOTS, GLOBAL)
INTEGER NUMSLOTS, GLOBAL
CHARACTER*16 CT(SLOTSIZE, NUMSLOTS)
```

Description

CT — is a buffer table where you want the cube information to be returned. It is declared as a two-dimensional array of character strings. SLOTSIZE is a fixed value that is defined in the *fcube.h* INCLUDE file. NUMSLOTS is a user-defined parameter.

NUMSLOTS — is the number of slots available in the buffer table for cube information.

GLOBAL — is a value that indicates which cube(s) on the system you want information about. The possible values are:

- **0** returns information about the current attached cube.
- **1** All cubes owned by user.
- **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 buffer table for all of the selected cubes, or for NUMSLOTS, whichever is less. If there are fewer selected cubes than NUMSLOTS, the remaining slots are cleared to blanks.

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

Example

A number of parameters are defined in the *fcube.h* INCLUDE file to help you define and access the buffer table. These parameters are:

```
INTEGER CUBENAME, USERNAME, SRMNAME, HOSTNAME, TTY, CUBETYPE
INTEGER CUBEINDEX, TTY1, TTY2, TTY3, TTY4, SLOTSIZE

PARAMETER (CUBENAME = 1)
PARAMETER (USERNAME = 2)
PARAMETER (SRMNAME = 3)
PARAMETER (HOSTNAME = 4)
PARAMETER (TTY = 5)
PARAMETER (CUBETYPE = 6)
PARAMETER (CINDEX = 7)
PARAMETER (TTY1 = 8)
PARAMETER (TTY2 = 9)
PARAMETER (TTY3 = 10)
PARAMETER (TTY4 = 11)
PARAMETER (SLOTSIZE = 11)
```

For example, to access the HOSTNAME of the second cube, you could write:

```
CHARACTER*16 CUBE2HOSTNAME
CUBE2HOSTNAME = CT(HOSTNAME, 2)
```

The fields of the buffer table contain the following information:

CUBENAME	Name of the cube.
USERNAME	Name of user.
SRMNAME	Name of the resource manager where the cube is attached.
HOSTNAME	Name of the host where GETCUBE was invoked. Name is either the remote system name or SRMNAME.
TTY	TTY where GETCUBE was invoked .
CUBETYPE	Type of this subcube.
CINDEX	System table index (this is not meant to be user-written application programs).
TTY1-TTY4	Names of terminals attached to the cube. Unused entries are set to all blanks.

Return Values

An integer containing the number of cubes that information is available for. If the number of cubes was greater than NUMSLOTS, this value will be larger than NUMSLOTS. The amount of information actually returned, however, never exceeds NUMSLOTS.

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

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

```
SUBROUTINE 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

```
      INTEGER NODE, PID
      INTEGER BUF1(2), BUF2(10)
      INTEGER ID1, ID2

      ID1 = ISEND( 101, BUF1, LEN(BUF1), NODE, PID )
      ID2 = ISEND( 102, BUF2, LEN(BUF2), NODE, PID )
C     DO SOME OTHER PROCESSING HERE.
          .
          .
C     NOW, WE NEED A MESSAGE BUFFER.

10    IF ((MSGDONE(ID1).NE.0) THEN
C     PROCESS MESSAGE 1
          .
          .
      ELSE IF(MSGDONE(ID2).NE.0) THEN
C     PROCESS MESSAGE 2
          .
          .
      ELSE
         CALL FLICK()
         GOTO 10
      ENDIF
```

Return Values

None.

FLUSHMSG(3) iPSC/2 FLUSHMSG(3)

Flush specified messages from the system.

Synopsis

SUBROUTINE FLUSHMSG(TYPESEL, NODE, PID)
INTEGER 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 TYPE's 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 is **-1**, all messages to the node process will be flushed.
- If TYPESEL is any negative number other than **-1**, a set of message TYPE's will be flushed. Refer to Appendix A 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 PID's 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 conjuction with MSGCANCEL or KILLPROC to ensure tht all specified messages are flushed.

Example

```
        CHARACTER MSG1, MSG2, MSG3
        INTEGER NODE, PID, MSGID
                .
                .
        NODE = 1
        PID = 0

C       SEND TWO SYNCHRONOUS MESSAGES FOLLOWED BY ONE ASYNCHRONOUS
C       MESSAGE TO PROCESS 0 ON NODE 1.  THEN FLUSH MESSAGES OF ALL
C       TYPES WAITING TO BE RECEIVED BY PROCESS 0 ON NODE 1.


        CALL CSEND(10, MSG1, LEN(MSG1), NODE, PID)
        CALL CSEND(11, MSG2, LEN(MSG2), NODE, PID)
        MSGID = ISEND(12, MSG3, LEN(MSG3), NODE, PID)
        CALL MSGCANCEL(MSGID)
        CALL FLUSHMSG(-1, NODE, PID)
```

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

SUBROUTINE GETCUBE(CUBENAME, CUBETYPE, SRMNAME, KEEP)
CHARACTER CUBENAME* (*)
CHARACTER CUBETYPE*(*)
CHARACTER SRMNAME*(*)
INTEGER KEEP

Description

CUBENAME — is 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. CUBENAME's cannot start with a dash ('-') or a space. If the argument is blank, 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 character string which 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 argument is blank, 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 ge 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, it will be rounded up.

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

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 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 a cube.

KEEP

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

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 arguments 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 exiting 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

```
      CALL GETCUBE('ALPHA', '16vx', ' ', 0)
      CALL GETCUBE('BETA', '8m8', 'TRAINON', 1

C     THE CURRENT CUBE IS 'BETA'
               .
               .
      CALL ATTACHCUBE ('ALPHA')

C     THE CURRENT CUBE IS NOW 'ALPHA'
               .
               .
```

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

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

```
INTEGER FUNCTION GINV(J)
INTEGER J
```

Description

J is a positive integer value.

GINV takes a Gray code element (positive 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

```
INTEGER FUNCTION GRAY (J)
INTEGER J
```

Description

J is a positive 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	2	3	4	5	6	7	→	position of each element in sequence
0	1	3	2	6	7	5	4	→	Gray code sequence in decimal

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

Synopsis

```
EXTERNAL PROC
SUBROUTINE HANDLER(TYPE, PROC)
INTEGER TYPE
```

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.

HANDLER allows a node process to assign a user-written exception handler to the occurrence of specific exceptions or traps. 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.

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

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.

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

Return Values

None.

Errors

HANDLER:invalid handler type	Specify one of types listed.
HANDLER:invalid length	Correct the 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

```
INTEGER FUNCTION HRECV (TYPESEL, BUF, LEN, PROC)
INTEGER TYPESEL
INTEGER BUF(*)
INTEGER LEN
EXTERNAL 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 TYPE's 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 TYPE's will be recognized. Refer to Appendix A for information on building a TYPESEL mask.

BUF
: is a buffer where the received message will be stored. Valid entries include any legal FORTRAN data type. It is recommended that data types 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 BUF in bytes. The maximum number of bytes that you can send in a single message to or from a host process is 256K. There is no limit on message length between nodes.

PROC
: is the procedure executed when the receive completes. This procedure is user-written and must be a C procedure with four arguments.

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

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 must be written in C and is called with four arguments:

```
proc_(type, count, node, pid)
int 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 temporarily 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

```
        CHARACTER BUF*1000
        EXTERNAL PROC

        HRECV(101, BUF, LEN(BUF), PROC)
C       PROC WILL BE CALLED WHEN THE RECEIVE COMPLETES

        MASK = MASKTRAP(1)

C       PROCESSING HERE WILL NOT BE INTERRUPTED

        MASKTRAP(MASK)

C       PROC CAN BE CALLED AGAIN
```

Return Values

None.

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

Errors

HRECV:buffer length exceeds allocation	Make sure length does not exceed buffer size.
HRECV:invalid length	Use a non-negative number of a length less than or equal to maximum message length.
HRECV:not enough memory	Simplify the application program.
HRECV:too many requests	Use MSGWAIT or MSGDONE for ISEND or IRECV requests that are outstanding.
HRECV:invalid buffer pointer	Specify a pointer which contains the address of a valid data buffer or correct the PROC parameter.

See Also

iprobe, infocount, infopid, infonode, infotype, irecv, masktraps
386 Programmer's Reference Manual

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

Returns information about a pending or received message.

Synopsis

INTEGER FUNCTION INFOCOUNT( )

INTEGER FUNCTION INFONODE( )

INTEGER FUNCTION INFOPID( )

INTEGER FUNCTION 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

```
      PARAMETER(BIGNUM=65536)

      INTEGER BUF(BIGNUM)
      INTEGER MSGTYPE, MSGLEN, MSGID
      INTEGER FROMNODE, FROMPID
           .
           .
C     WAIT FOR ANY MESSAGE TO ARRIVE, AND GET ITS TYPE WHEN ONE DOES.

      CALL CPROBE(-1)
      MSGTYPE = INFOTYPE()

C     RECEIVE THE WAITING MESSAGE AND GET ITS LENGTH IN BYTES.

      CALL CRECV(MSGTYPE, BUF, 4*BIGNUM)
      MSGLEN = INFOCOUNT()
           .
           .
C     POST A RECEIVE FOR A TYPE 10 MESSAGE AND DETERMINE WHICH NODE
C     SENT IT WHEN IT ARRIVES.

      MSGID = IRECV(10, BUF, 4*BIGNUM)
           .
           .
      IF (MSGDONE(MSGID) .EQ. 1) THEN
           FROMNODE = INFONODE()
           FROMPID = INFOPID()
      END IF
```

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, HANDLER

IPROBE(3) iPSC/2 IPROBE(3)

Determine whether a message of a selected TYPE is pending.

Synopsis

```
INTEGER FUNCTION IPROBE( TYPESEL)
INTEGER 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 TYPE's 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 TYPE's will be recognized. Refer to Appendix A for information on building 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. 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 function 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.

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

Example

```
C       LOOP CONTINUOUSLY UNTIL A TYPE 1 OR TYPE 2 MESSAGE ARRIVES,
C       THEN PROCESS ACCORDINGLY.

10      IF (IPROBE(1) .EQ. 1) THEN

C               PROCESS TYPE 1 MESSAGE.
                         .
                         .
                GOTO 2
        ELSE IF (IPROBE(2) .EQ. 1) THEN

C               PROCESS TYPE 2 MESSAGE.
                         .
                         .
                GOTO 20
        ELSE
                CALL FLICK()
        END IF
        GOTO 1
20                       .
```

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

```
INTEGER FUNCTION IRECV(TYPESEL, BUF, LEN)
INTEGER TYPESEL
INTEGER BUF(*)
INTEGER 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 TYPE's 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 TYPE's will be recognized. Refer to Appendix A for information on building a TYPESEL mask.

BUF is a buffer where the received message will be stored. Valid entries include any legal FORTRAN data type. It is recommended that data types 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 BUF in bytes. The maximum number of bytes that you can send in a single message to or from a host process is 256K. There is no limit on message length between nodes.

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

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 argument. 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 function 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

```
      INTEGER BUF(50)
      INTEGER MSGID, TYPE
             .
             .
C     INITIATE RECEIPT OF A MESSAGE.

      MSGID = IRECV(TYPE,BUF,50*4)

C     PERFORM TASKS WHICH ARE NOT DEPENDENT ON THE MESSAGE BEING
C     RECEIVED.
             .
             .
C     NOW WAIT TO GUARANTEE THE RECEIPT OF A MESSAGE.

      CALL MSGWAIT(MSGID)

C     NOW YOU CAN USE THE CONTENTS OF 'BUF' AND/OR THE INFO ROUTINES
C     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.

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

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 valid data buffer.

See Also

CSEND, INFOCOUNT, INFONODE, INFOPID, INFOTYPE, ISEND, MSGCANCEL, MSGDONE, MSGWAIT

ISEND(3) iPSC/2 ISEND(3)

Send a message.

Synopsis

```
INTEGER FUNCTION ISEND(TYPE, BUF, LEN, NODE, PID)
INTEGER TYPE
INTEGER BUF(*)
INTEGER 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 buffer which contains the message that you want to send. Valid entries include any legal FORTRAN 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 PID's 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.

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

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

```
      CHARACTER MSG*132
      INTEGER MSGID, MSGLEN, PID, TYPE
              .
              .
      MSGLEN = LEN(MSG)
      NODE = 16
      PID = 10
      TYPE = 1

C     SEND A TYPE 1 MESSAGE TO PROCESS 10 ON NODE 16.

      MSGID = ISEND(TYPE, MSG, MSGLEN, NODE, PID)
              .
              .
C     PERFORM TASKS THAT DO NOT CHANGE THE CONTENTS OF MSG.
              .
              .
      CALL MSGWAIT(MSGID)

C     NOW YOU CAN RE-USE THE MESSAGE BUFFER.
```

Return Values

Returns a non-negative message id which is used by MSGCANCEL, MSGDONE, or MSGWAIT.

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

Errors

ISEND: no pid defined	Use SETPID to define a host process id.
ISEND: invalid type	Use a non-negative number.
ISEND: invalid length	Use a non-negative number or a length less than or equal to maximum message length.
ISEND: invalid node	Use NUMNODES to determine cube size and MYHOST to determine host number.
ISEND: buffer length exceeds allocation	Make sure length does not exceed buffer size.
ISEND: invalid buffer pointer	Specify a valid data buffer.
ISEND:not enough memory	Make sure message buffer starts on a boundary which is a multiple of 4 or make more memory available to this process.
ISEND:too many requests	Use MSGWAIT or MSGDONE for ISEND or IRECV requests that are outstanding.

See Also

CRECV, CSEND, IRECV, MSGCANCEL, MSGDONE, MSGWAIT

KILLCUBE(3) iPSC/2 KILLCUBE(3)

Terminate and clear out a process(es).

Synopsis

SUBROUTINE KILLCUBE(NODE, PID)
INTEGER 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:

```
CALL  KILLPROC(NODE, PID)
CALL  WAITALL(NODE, PID)
CALL  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

SUBROUTINE KILLPROC(NODE, PID)
INTEGER 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

```
C       KILL PROCESS 3 ON NODE 15.
        CALL KILLPROC(15, 3)

C       KILL PROCESS 10 ON ALL NODES.
        CALL KILLPROC(-1, 10)

C       KILL ALL PROCESSES ON ALL NODES.
        CALL KILLPROC(-1, -1)
```

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

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

```
SUBROUTINE 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

```
SUBROUTINE LED(LSTATE)
INTEGER 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

SUBROUTINE LOAD (FILENAME, NODE, PID)
CHARACTER FILENAME*(*)
INTEGER NODE, PID

Description

FILENAME	is 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, 255 characters or less.
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.

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

Example

Case 1

```
      CHARACTER*4  NAME
      INTEGER  NODE
      INTEGER PID
           .
           .
      NAME  = 'PROG1'
      NODE  = 0
      PID  = 1
C     LOAD NODE PROGRAM PROG 1
C     INTO NODE 0 AND ASSIGN IT
C     A PROCESS ID OF 1.
           .
           .
      CALL LOAD(NAME, NODE, PID)
           .
           .
```

Case 2

```
      INTEGER  NODE
      INTEGER  PID
           .
           .
      CALL  LOAD('PROG1', -1, 1)
           .
           .
C     LOAD NODE PROGRAM PROG1 INTO
C     ALL NODES WITH A PROCESS ID
C     OF 1.
```

Return Values

None.

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

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 of cube size.
LOAD: out of process slots	Use fewer processes.
LOAD: pid already in use	Use a different PID.
LOAD: not enough memory	Reduce the size of the process.
LOAD: invalid loader record	Internal error, try again. 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 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

INTEGER FUNCTION MASKTRAP(MASK)
INTEGER 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 for a process.

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

Example

```
      CHARACTER BUF*1000
      EXTERNAL PROC

      HRECV(101, BUF, LEN(BUF), PROC)
C     PROC WILL BE CALLED WHEN THE RECEIVE COMPLETES

      MASK = MASKTRAP(1)

C     PROCESSING HERE WILL NOT BE INTERRUPTED

      MASKTRAP(MASK)

C     PROC CAN BE CALLED AGAIN
```

Return Values

The previous value of the mask.

Errors

None.

See Also

HRECV

MCLOCK(3) iPSC/2 MCLOCK(3)

Return the time.

Synopsis

INTEGER FUNCTION 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. Progams 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 overflows to a negative number after about 25 days, and rolls over to zero approximately every 50 days.

Return Values

An integer containing the time value.

Errors

None.

MSGCANCEL(3) iPSC/2 MSGCANCEL(3)

Cancel an asynchronous send or receive operation.

Synopsis

SUBROUTINE MSGCANCEL(ID)
INTEGER 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 argument 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

```
      CHARACTER*100 BUF
      INTEGER MSGID, TYPE
      LOGICAL ERRORCODE
              .
              .
      MSGID = IRECV(TYPE, BUF, LEN(BUF))
              .
              .
C     PERFORM TASKS WHILE WAITING FOR THE RECEIVE TO COMPLETE.
              .
              .
      IF (ERRORCODE) CALL MSGCANCEL(MSGID)
```

Return Values

None.

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

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.
MSGDONE: received message too long for buffer	Make sure the buffer is large enough to hold the message.

See Also

IRECV, ISEND

MSGDONE(3) iPSC/2 MSGDONE(3)

Determine whether an asynchronous send or receive operation has completed.

Synopsis

```
INTEGER FUNCTION MSGDONE(ID)
INTEGER 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 argument 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. After MSGDONE returns, the message id is no longer valid and cannot be used again in MSGDONE or MSGWAIT.

Example

```
      CHARACTER*100 BUF
      INTEGER MSGID, TYPE
              .
              .
      MSGID = ISEND(TYPE, BUF, LEN(BUF), 1, 1)
              .
              .
      IF (MSGDONE(MSGID) .EQ. 1) THEN
C                YOU MAY NOW RE-USE THE MESSAGE BUFFER.
      ELSE
C                THE MESSAGE BUFFER STILL NEEDS TO BE PROTECTED.
      END IF
```

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

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 argument 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

SUBROUTINE MSGWAIT(ID)
INTEGER 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 argument 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 argument again because the send or receive operation is complete.

Example

```
      INTEGER RESULT
      INTEGER MSGID, TYPE
                    .
                    .

      MSGID = IRECV(TYPE, RESULT, 4)

C     PERFORM TASKS THAT DO NOT USE 'RESULT'.
                .
                .
C     STOP UNTIL THE MESSAGE IS RECEIVED.

      CALL MSGWAIT(MSGID)

C     NOW YOU CAN USE 'RESULT'.
```

MSGWAIT(3) iPSC/2 MSGWAIT(3)

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

INTEGER FUNCTION 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

```
      CHARACTER*100 MSG
      INTEGER HOSTID
               .
               .
      HOSTID = MYHOST()
C     SEND A TYPE 10 MESSAGE TO PROCESS 1 ON THE HOST.
      CALL CSEND(10, MSG, LEN(MSG), HOSTID, 1)
```

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

INTEGER FUNCTION 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

```
        INTEGER THISNODE
                .
                .
C       OBTAIN THE LOGICAL NODE ID OF THIS NODE AND KILL ALL PROCESSES
C       ON IT, INCLUDING THIS ONE.

        THISNODE = MYNODE()
        CALL KILLPROC(THISNODE, -1)
```

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

INTEGER FUNCTION 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

```
        CHARACTER*100 BUF
        INTEGER TYPE
                .
                .
C       SEND A MESSAGE TO THE PROCESSES ON ALL NODES WITH THE SAME PID
C       AS THIS PROCESS.

        CALL CSEND(TYPE, BUF, LEN(BUF), -1, MYPID())
```

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

SUBROUTINE NEWSERVER(CUBENAME)
CHARACTER CUBENAME*(*)

Description

CUBENAME	is 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 argument is blank, 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.

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 dimension of allocated cube in which process is running.

Synopsis

INTEGER FUNCTION 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.

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

NUMNODES

NUMNODES(3) iPSC/2 NUMNODES(3)

Obtain the number of nodes in the cube.

Synopsis

INTEGER FUNCTION 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

```
      INTEGER LASTNODE, TOTALNODES
                .
                .
      CALL GETCUBE('ALPHA', '8m4', 'SAXON', 1)
      CALL GETCUBE('BETA', '32vx', 'VIKING', 1)
      SETPID(1)

C     THE LAST NODE ON THE CUBE NAMED 'BETA' IS 31.

      LASTNODE = NUMNODES() - 1

C     FIND OUT THE TOTAL NUMBER OF NODES ON THE CUBE NAMED 'ALPHA'.

      CALL ATTACHCUBE('ALPHA')

      TOTALNODES = NUMNODES()
```

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

SUBROUTINE RELCUBE(CUBENAME)
CHARACTER CUBENAME*(*)

Description

CUBENAME is 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 argument is blank, 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(es) running on the cube.

Example

```
      CALL GETCUBE('BIG', '128', '', 0)
            .
            .
C     CLEAN UP AND RELEASE THE CUBE.

      CALL KILLCUBE(-1, -1)
      CALL 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

SUBROUTINE SETPID(PID)
INTEGER PID

Description

PID specifies the process id that you are assigning to a host process. Valid PID's 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

```
CALL GETCUBE('ALPHA', '16vx', ' ', 0)
CALL GETCUBE('BETA', '8m8', 'TRAINON', 1)
CALL SETPID(99)
         .
         .
CALL ATTACHCUBE('ALPHA')
CALL SETPID(99)
         .
         .
CALL ATTACHCUBE('BETA')
CALL SETPID(99)
         .
         .
```

Return Values

None.

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

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

SUBROUTINE SETSYSLOG(STDFD)
INTEGER 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

```
CALL GETCUBE('ALPHA', '16', ' ', 0)
CALL SETPID(99)
CALL SETSYSLOG(1)
CALL 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

```
SUBROUTINE WAITALL(NODE, PID)
INTEGER NODE
INTEGER 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

```
      CALL WAITALL(10, 3)
C     WAIT FOR PROCESS 3 ON NODE 10 TO COMPLETE.

      CALL WAITALL(15, -1)
C     WAIT FOR ALL PROCESSES ON NODE 15 TO COMPLETE.

      CALL WAITALL(-1, -1)
C     WAIT FOR ALL PROCESSES ON ALL NODES TO COMPLETE.
```

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

Return Values

None.

Errors

WAITALL: no pid defined	Use SETPID to define a host process id.
WAITALL: invalid node	Use NUMNODES to determine cube size and MYHOST to determine host number.

See Also

WAITONE

WAITONE(3) iPSC/2 WAITONE(3)

Wait for a specified process to complete.

Synopsis

SUBROUTINE WAITONE(NODE, PID, CNODE, CPID, CCODE)
INTEGER 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 PID's 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 terminates normally, the exit code is zero. 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

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

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 specified process(es) on the first node completes.

Example

```
      INTEGER CNODE, CPID, CCODE
             .
             .
             .
      WAITONE(-1, -1, CNODE, CPID, CCODE)
C     WAIT FOR THE FIRST PROCESS ON ANY NODE TO COMPLETE.

      WRITE(*,*) 'PROCESS', CPID, 'ON NODE', CNODE,
1                'FINISHED WITH COMPLETION CODE', CCODE
```

Return Values

None.

Errors

WAITONE: no pid defined	Use SETPID to define a host process id.
WAITONE: invalid node	Use NUMNODES to determine cube size and MYHOST to determine host number.

See Also

WAITALL, WAITCUBE command (Chapter 2)

CHAPTER 4

iPSC/2 VECLIB ROUTINES

INTRODUCTION

The Vector Library (VecLib) contains a set of routines that support Intel's numeric products. These routines provide a basic set of vector operations which can be used to replace FORTRAN DO loops. There are six versions of the VecLib routines contained in the following libraries:

	Library	Operations Implemented With	Function
1.	/usr/lib/libvec.a	FORTRAN	Executes on standard node board (80387) or system resource manager (80387).
2.	/usr/lib/libsxvec.a	FORTRAN	Executes on SX option board.
3.	/usr/lib/libvxvec.a	MICROCODE	Executes on vector processor board
4.	/usr/lib/libvxdbvec.a	MICROCODE	Executes on vector processor board. Provides debug checks.
5.	/usr/lib/libvxsxvec.a	MICROCODE	Executes on vector processor board and nodes that have SX option.
6.	/usr/lib/libvxsxdbvec.a	MICROCODE	Executes on vector processor board and nodes that have SX option. Provides debug checks

The "/usr/lib/libsxvec.a" library can be used only if you have the SX option installed. The "/usr/lib/libvxsxvec.a" and "/usr/lib/libvxsxdbvec.a" libraries can be used only if you have both the SX option and vector processor boards installed. The last four libraries can be used only if you have vector processor boards installed. (Refer to the iPSC/2 VX User's Guide for more information.)

The VecLib routines are listed alphabetically by its root name. Each routine is defined. Then the calling sequence, FORTRAN equivalent, and mathematical description are given. Error messages are listed on page 4-7 and a summary list of all VecLib routines appears on page 4-9.

NAMING CONVENTIONS

Routines exist for both single precision and double precision functions. In some cases, routines exist for integer, logical, and complex functions. The data type a function operates on is indicated by the first or second letter of the routine name; "**D**" for double, "**S**" for single, "**I**" for integer, "**L**" for logical, and "**C**" for complex. For example:

Root	Double Precision	Single Precision	Integer
_COPY	DCOPY	SCOPY	ICOPY

The vector library includes most of the Basic Linear Algebra Subprograms (BLAS) and uses the BLAS calling sequence and naming conventions.

NOTE

For more information on the Basic Linear Algebra Subprograms, refer to the following publications:

Bunch, J.R., Dongarra, J.J., Moler, C.B., Stewart, G.W., *LINPACK User's Guide*, 1979, Appendix A.

Lawson, C., Hanson, R. Kincaid, D., Krogh, F., "Basic linear algebra subprograms for FORTRAN usage," *ACM Trans. Math. Software*, 1979, Vol. 5, No. 3, pp. 308-371.

Some of the vector routines use the Reverse Polish Notation (RPN) naming convention. An example is shown in Table 4-1.

Table 4-1
Reverse Polish Notation

Routine	Notation	Description
_SVTVM	S = Scalar V = Vector T = Times M = Minus	Scalar times vector quantity minus vector
_SVVTP	S = Scalar V = Vector T = Times P = Plus	Scalar plus the quantity of vector times vector

FORMAL PARAMETERS

The calling sequence for each routine always includes the number of iterations as the first parameter, followed by any scalar and vector values. Vector arguments in the parameter list are in bold print to distinguish them from scalar arguments. Each vector argument is followed immediately in the argument list by its stride (i.e., **Z**, INCZ). Notice that some routines have an output vector while other routines overwrite one of the input vectors.

Routines which take one vector as input use "**X**". Routines which use two vectors use "**X**" and "**Y**". Routines which use three vectors use "**X**", "**Y**", and "**Z**". Routines which use four vectors use "**W**", "**X**", "**Y**", and "**Z**".

Routines which use one scalar use "ALPHA". Routines which use two scalars, use "ALPHA" and "BETA".

In general, formal parameters can be described as follows:

N	number of vector elements to process
X	vector with at least (n-1) * ABS(incx) + 1 elements
INCX	stride of the vector **X**
ALPHA	scalar constant
BETA	scalar constant

GENERAL RULES

- All floating point arguments and results match the type (either double or single) of the subprogram.

- Each routine, as documented in this chapter, includes three parameters:
  the number of iterations
  vector and scalar values, if appropriate
  strides

- In the calling sequence, if the vector "type" differs from the type indicated by the routine name, it is indicated by a single letter preceded by an "S" for single, "D" for double, "I" for integer, "L" for logical, or a "C" for complex type vectors. For example, the vector "**LZ**" in the call below has type logical, "**X**" and "**Y**" have type single precision.

  ```
  CALL SGT (N, X, INCX, Y, INCY, LZ, INCZ)
  ```

- Data types follow ANSI FORTRAN standards. All integer references are INTEGER*4, all complex references are COMPLEX*8, and all logical references are LOGICAL*4.

- The trigonometric routines (__vatn2, __vcos, __vsin) expect their arguments to be in radians, or return a vector whose values are expressed in radians.

- Limitations specified in some routines are not checked at runtime. If violated, erroneous results are produced. The only routines which provide some parameter-checking are in "libvxdbvec.a" or "libvxsxdbvec.a".

GENERAL RULES (cont.)

- Unless otherwise noted, vector elements which are not contiguous, but are separated by equal distances (either positive or negative), can be accessed by using a stride other than one. Non-unit strides follow the BLAS parameter-passing convention which does not allow negative indices to arrays.

  The formula for indexing through the vectors [n,x(lowest$index),incx] for i from 1 to 'n' is:

  ```
  if (incx > 0) then
     x((i-1) * incx + lowest$index)
  else
     x((i-n) * incx + lowest$index)
  endif
  ```

  Example of positive and negative strides not equal to 1:

  ```
  Real x (10)
  Real y (10)
  Data x /1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0/
  Data y /.01,.02,.03,.04,.05,.06,.07,.08,.09,.10/

  call saxpy (3,1.0,x,2,y,1)
     returns with
  y = (1.01,3.02,5.03,.04,.05,.06,.07,.08,.09,.10)

  call saxpy (3,1.0,x,-2,y,1)
     returns with
  y = (5.01,3.02,1.03,.04,.05,.06,.07,.08,.09,.10)

  call saxpy (3,1.0,x(3),-2,y(2),2)
     returns with
  y = (.01,7.02,.03,5.04,.05,3.06,.07,.08,.09,.10)
  ```

GENERAL LIMITATIONS

The following limitations apply to vector routines described in this chapter.

- Routines do nothing for "n" less than or equal to zero.
- Output vector strides of zero produce undefined results if "n" is greater than 1.
- Care must be used when overlapping input and output vectors. Because of the pipelined nature of the VX arithmetic, the results for one iteration ("i") are not always written to memory before the next iteration ("i + 1") is begun.
- FORTRAN and MICROCODE versions of libraries may not produce exactly the same results.

    a. 80387 hardware supports denormals, VX and SX hardward does not.
    b. VX hardware does not do IEEE divide or square root.
    c. 80387, VX, and SX use different algorithms for transcendental functions.
    d. Because of the pipelined nature of the VX arithmetic hardware, the accumulation of round off error for reduction operations is not the same as with the 80387 or SX.

ERROR MESSAGES

Error messages are generated only when running VecLib routines in libvxdbvec.a or libvxsxdbvec.a. They are listed below alphabetically with a description following each message. "<x>" represents vector or scalar parameter names and "<ddot>" is used to represent any of the VecLib routine names.

`VX veclib: Error in length for argument <x> in call to <ddot>`

Cause: Internal error message, libvxdbvec.a or libvxsxdbvec.a error.
Action: Call Intel

`VX veclib: Error in memory space for argument <x> in call to <ddot>`

Description: The data is not on vector board memory, but is on the node board.

Cause: Incorrect use of vxld.
Action: Move the data section containing the vector to the VX memory using vxld.

`VX veclib: Error in physical alignment for argument <x> in call to <ddot>`

Cause: Internal error message, libvxdbvec.a or libvxsxdbvec.a error.
Action: Call Intel

`VX veclib: Error in physical bounds check for argument <x> in call to <ddot>`

Cause: Internal error message, libvxdbvec.a or libvxsxdbvec.a error.
Action: Call Intel

`VX veclib: Error in type for check of argument <x> in call from <ddot>`

Cause: Internal error message, libvxdbvec.a or libvxsxdbvec.a error.
Action: Call Intel

`VX veclib: Error in virtual alignment for argument <x> in call to <ddot>`

Description: The virtual address's offset modulo the data size in 8 bit bytes is not equal to zero.

Cause: Ignored warning from compiler about equivalence misalignment.
Action: Change equivalence statement.

ERROR MESSAGES (cont.)

VX veclib: Error in virtual bounds check for argument <x> in call to <ddot>

Description: The data is not contiguous in virtual memory.

Cause: Requesting vector operations outside the bounds of an array.
Action: Locate error in your FORTRAN program.

VX veclib: The following enabled exceptions have occurred: NaN (not-a-number), overflow, underflow.

Cause: Requested modification of exceptions using vpexcept and algorithm-produced exception.
Action: Modify algorithm used so exception does not occur.

ROUTINES

A summary list of VecLib routines follows. The routines are listed alphabetically by root name. An underscore "_" represents where the data type is specified in the routine name. The data type a function operates on is indicated by the first or second letter of the routine name; "D" for double, "S" for single, "I" for integer, "L" for logical, and "C" for complex. For example, DASUM is the double precision routine to calculate the sum of absolute values, while SASUM is the single precision version.

The first three columns contain the routine's root name, data types and a brief description. The fourth column gives the calling sequence and FORTRAN equivalent. A "d" is used in the examples to represent the double precision version of the routine. When "i" is used as an index in the FORTRAN equivalent statement, "i" ranges from 1 to "N", where "N" is the length of the vector operands.

Table 4-2 (Page 1 of 7)
VecLib Routine Summary

Root Name	Data Type	Description	Calling Sequence/ FORTRAN Equivalent

_asum	d,s	Sum of absolute values	`d = dasum (n,`**x**`,incx)`
			`dasum = 0.0`
			`dasum = dasum + abs(x(i))`

_axpy	d,s	Scalar times a vector plus a vector to itself	`call daxpy (n,alpha,`**x**`,incx,`**y**`,incy)` `y(i) = y(i) + alpha*x(i)`
_clip	d,s	Clip to interval [alpha,beta]	`call dclip (n,`**x**`,incx,alpha,beta,`**y**`,incy)` `y(i) = min(max(x(i),alpha),beta)`
_cndst	d,s	Conditional assignment	`call dcndst(n,`**x**`,incx,`**ly**`,incy,`**z**`,incz)` `if (ly(i)) z(i) = x(i)`
_copy	d,s,i,l	Copy vector	`call dcopy (n,`**x**`,incx,`**y**`,incy)` `y(i) = x(i)`

_dot	d,s	Dot product of two vectors	`d = ddot (n,`**x**`,incx,`**y**`,incy)`
			`ddot = 0.0`
			`ddot = ddot + x(i)*y(i)`

_eq	d,s,i	Vector element equality	`call deq (n,`**x**`,incx,`**y**`,incy,`**lz**`,incz)` `lz(i) = x(i) .eq. y(i)`
_fft	c	Fast Fourier Transform	`call cfft (n,`**x**`,incx,`**y**`,incy)` `y = fft(x)`
_fill	d,s,i,l	Fill vector	`call dfill (n,alpha,`**x**`,incx)` `x(i) = alpha`
_gathr	d,s	Vector gather	`call dgathr (n,`**x**`,incx,`**iy**`,incy,`**z**`,incz)` `z(i) = x(iy(i))`
_ge	d,s,i	Vector element greater than or equal to	`call dge (n,`**x**`,incx,`**y**`,incy,`**lz**`,incz)` `lz(i) = x(i) .ge. y(i)`
_gt	d,s,i	Vector element greater than	`call dgt (n,`**x**`,incx,`**y**`,incy,`**lz**`,incz)` `lz(i) = x(i) .gt. y(i)`

Table 4-2 (Page 2 of 7)
VecLib Routine Summary

Root Name	Data Type	Description	Calling Sequence/ FORTRAN Equivalent

i_amax	d,s	Index of maximum absolute value	`i = idamax (n,`**`x`**`,incx)`
			`idamax = min (1, max(0,n))`
			`if (abs(x(i)).gt.abs(x(idamax)))idamax = i`

i_amin	d,s	Index of minimum absolute value	`i = idamin (n,`**`x`**`,incx)`
			`idamin = min (1, max(0,n))`
			`if (abs(x(i)).lt.abs(x(idamin)))idamin = i`

_iclip	d,s	Inverted clip	`call diclip (n,`**`x`**`,incx,alpha,beta,`**`y`**`,incy)`
			`if  (x(i).lt.(alpha + beta)/2.0) then`
			` y(i) = min (x(i),alpha)`
			`else`
			` y(i) = max (x(i),beta)`
			`endif`

icount	l	Number of logical true values	`i = icount(n, `**`lx`**`, incx)`
			`icount = 0`
			`if (lx(i)) icount = icount + 1`

_ifft	c	Inverse Fast Fourier Transform	`y = cifft (n,`**`x`**`,incx,`**`y`**`,incy)` $y = ifft^{-1}(x)$

ilast	l	Index of last logical true value	`i = ilast(n,`**`lx`**`,incx)`
			`ilast = 0`
			`if (lx(i)) ilast = i`

i_max	d,s	Index of maximum value	`i = idmax (n,`**`x`**`,incx)`
			`idmax = min (1, max(0,n))`
			`if (x(i) .gt. x(idmax)) idmax = i`

i_min	d,s	Index of minimum value	`i = idmin (n,`**`x`**`,incx)`
			`idmin = min (1, max(0,n))`
			`if (x(i) .lt. x(idmin)) idmin = i`

land	l	Vector logical and with vector	`call land(n,`**`x`**`,incx,`**`y`**`,incy,`**`z`**`,incz)` `z(i) = x(i) .and. y(i)`

lany	l	Logical true if any of x are true	`l = lany (n,`**`x`**`,incx)`
			`lany = .false.`
			`if (x(i)) lany = .true.`

lnot	l	Vector logical negation	`call lnot(n,`**`x`**`,incx,`**`y`**`,incy)` `y(i) = .not. x(i)`

Table 4-2 (Page 3 of 7)
VecLib Routine Summary

Root Name	Data Type	Description	Calling Sequence/ FORTRAN Equivalent
lor	I	Vector logical or with vector	`call lor(n,x,incx,y,incy,z,incz)` `z(i) = x(i) .or. y(i)`
lsand	I	Scalar logical and with scalar	`call lsand(n,alpha,x,incx,y,incy)` `y(i) = alpha .and. x(i)`
lsor	I	Scalar logical or with vector	`call lsor(n,alpha,x,incx,y,incy)` `y(i) = alpha .or. x(i)`

_mask	d,s	Conditional assignment	`call dmask(n,w,incw,x,incx,ly,incy,z,incz)`
			`if (ly(i))`
			` z(i) = w(i)`
			`else`
			` z(i) = x(i)`

_ne	d,s,i	Vector element inequality	`call dne(n,x,incx,y,incy,lz,incz)` `lz(i) = x(i) .ne. y(i)`
_neg	d,s.i	Change sign	`call dneg(n,x,incx)` `x(i) = -x(i)`
_nrm2	d,s	Euclidean vector norm	`d = dnrm2(n,x,incx)` `d = sqrt(ddot(n,x,incx,x,incx))`
_ramp	d,s,i	Ramp function	`call dramp(n,alpha,beta,x,incx)` `x(i) = alpha + (i-1)*beta`
_sadd	d,s,i	Scalar plus vector	`call dsadd(n,alpha,x,incx,y,incy)` `y(i) = alpha + x(i)`
_scal	d,s	Scalar times a vector to itself	`call dscal (n,alpha,x,incx)` `x(i) = alpha * x(i)`
_scatr	d,s	Vector scatter	`call dscatr(n,x,incx,iy,incy,z,incz)` `z(iy(i)) = x(i)`
_sdiv	d,s,i	Scalar divided by vector	`call dsdiv(n,alpha,x,incx,y,incy)` `y(i) = alpha / x(i)`
_seq	d,s,i	Vector equal to scalar	`call dseq(n,alpha,x,incx,ly,incy)` `ly(i) = alpha .eq. x(i)`
_sge	d,s,i	Scalar greater than or equal to vector	`call dsge(n,alpha,x,incx,ly,incy)` `ly(i) = alpha .ge. x(i)`

Table 4-2 (Page 4 of 7) VecLib Routine Summary

Root Name	Data Type	Description	Calling Sequence/ FORTRAN Equivalent
_sgt	d,s,i	Scalar greater than vector	call dsgt(n,alpha,**x**,incx,**ly**,incy) ly(i) = alpha .gt. x(i)
_sle	d,s,i	Scalar less than or equal to vector	call dsle(n,alpha,**x**,incx,**ly**,incy) ly(i) = alpha .le. x(i)
_slt	d,s,i	Scalar less than vector	call dslt(n,alpha,**x**,incx,**ly**,incy) ly(i) = alpha .lt. x(i)
_smul	d,s,i	Scalar times vector	call dsmul(n,alpha,**x**,incx,**y**,incy) y(i) = alpha * x(i)
_sne	d,s,i	Vector not equal to scalar	call dsne(n,alpha,**x**,incx,**ly**,incy) ly(i) = alpha .ne. x(i)
_ssub	d,s,i	Scalar vector subtraction	call dssub(n,alpha,**x**,incx,**y**,incy) y(i) = alpha - x(i)
_sum	d,s	Vector sum	s = dsum (n,**x**,incx) dsum = 0.0 dsum = dsum + x(i)
_svmvt	d,s	Scalar minus vector quantity times vector	call dsvmvt(n,alpha,**x**,incx,**y**,incy,**z**,incz) z(i) = (alpha - x(i)) * y(i)
_svpvt	d,s	Scalar plus vector quantity times vector	call dsvpvt(n,alpha,**x**,incx,**y**,incy,**z**,incz) z(i) = (alpha + x(i)) * y(i)
_svtsp	d,s	Scalar times vector quantity plus scalar	call dsvtsp(n,alpha,beta,**x**,incx,**y**,incy) y(i) = alpha * x(i) + beta
_svtvm	d,s	Scalar times vector quantity minus vector	call dsvtvm(n,alpha,**x**,incx,**y**,incy,**z**,incz) z(i) = alpha * x(i) - y(i)
_svtvp	d,s	Scalar times quantity of vectorplus vector	call dsvtvp(n,alpha,**x**,incx,**y**,incy,**z**,incz) z(i) = alpha * x(i) + y(i)
_svvmt	d,s	Scalar times quantity of vector minus vector	call dsvvmt(n,alpha,**x**,incx,**y**,incy,**z**,incz) z(i) = alpha * (x(i) - y(i))
_svvpt	d,s	Scalar times quantity of vector plus vector	call dsvvpt(n,alpha,**x**,incx,**y**,incy,**z**,incz) z(i) = alpha * (x(i) + y(i))

Table 4-2 (Page 5 of 7) VecLib Routine Summary

Root Name	Data Type	Description	Calling Sequence/ FORTRAN Equivalent
_svvtm	d,s	Scalar minus quantity of vector times vector	call dsvvtm(n,alpha,**x**,incx,**y**,incy,**z**,incz) z(i) = alpha - x(i) * y(i)
_svvtp	d,s	Scalar plus quantity of vector times vector	call dsvvtp(n,alpha,**x**,incx,**y**,incy,**z**,incz) z(i) = alpha + x(i) * y(i)
_swap	d,s	Swap vectors	call dswap (n,**x**,incx,**y**,incy) t = y(i) y(i) = x(i) x(i) = t
_vabs	d,s,i	Element-wise absolute value	call dvabs (n,**x**,incx,**y**,incy) y(i) = abs(x(i))
_vadd	d,s,i	Vector addition	call dvadd (n,**x**,incx,**y**,incy,**z**,incz) z(i) = x(i) + y(i)
_vamax	d,s	Vector element-wise maximum absolute value	call dvamax (n,**x**,incx,**y**,incy,**z**,incz) z(i) = max(abs(x(i)),abs(y(i)))
_vamin	d,s	Vector element-wise minimum absolute value	call dvamin (n,x,incx,y,incy,z,incz) z(i) = min(abs(x(i)),abs(y(i)))
_vatan	d,s	Element-wise inverse-tangent	call dvatan (n,**x**,incx,**y**,incy) y(i) = atan(x(i))
_vatn2	d,s	Element-wise inverse-tangent	call dvatn2 (n,**x**,incx,**y**,incy,**z**,incz) z(i) = atan2(x(i),y(i))
_vcos	d,s	Element-wise cosine	call dvcos (n,**x**,incx,**y**,incy) y(i) = cos(x(i))
vdble	d	Single to double precision	call vdble (n,**sx**,incx,**dy**,incy) dy(i) = dble(sx(i))
_vdiv	d,s,i	Element-wise vector division	call dvdiv (n,**x**,incx,**y**,incy,**z**,incz) z(i) = x(i) / y(i)
_vexp	d,s	Element-wise exponential	call dvexp (n,**x**,incx,**y**,incy) y(i) = exp(x(i))
_vfix	d,s	Truncate elements to integer values	call dvfix (n,**x**,incx,**iy**,incy) iy(i) = int(x(i))

Table 4-2 (Page 6 of 7)
VecLib Routine Summary

Root Name	Data Type	Description	Calling Sequence/ FORTRAN Equivalent
_vfloa	d,s	Convert integer to floating point	call dvfloa (n,**ix**,incx,**y**,incy) y(i) = ix(i)
_vlg10	d,s	Element-wise base 10 logarithm.	call dvlg10 (n,**x**,incx,**y**,incy) y(i) = log10(x(i))
_vlog	d,s	Element-wise natural logarithm	call dvlog (n,**x**,incx,**y**,incy) y(i) = log(x(i))
_vmax	d,s	Vector element-wise maximum	call dvmax (n,**x**,incx,**y**,incy,**z**,incz) z(i) = max(x(i),y(i))
_vmin	d,s	Vector element-wise minimum	call dvmin (n,**x**,incx,**y**,incy,**z**,incz) z(i) = min(x(i),y(i))
_vmul	d,s,i	Vector element-wise multiplication	call dvmul (n,**x**,incx,**y**,incy,**z**,incz) z(i) = x(i) * y(i)
_vneg	d,s,i	Negate vector	call dvneg(n,**x**,incx,**y**,incy) y(i) = -x(i)
_vpoly	d,s	Vector polynomial evaluation	call dvpoly (n,**x**,incx,m,**c**,incc,**y**,incy) y(i)=c(1)+c(2)*x(i)+...+c(m)*x(i)**(m-1)
_vpow	d,s	Element-wise power function	call dvpow (n,**x**,incx,**y**,incy,**z**,incz) z(i) = x(i) ** y(i)
_vrecp	d,s	Vector reciprocal	call dvrecp(n,**x**,incx,**y**,incy) y(i) = 1.0 / x(i)
_vsin	d,s	Element-wise sine	call dvsin (n,**x**,incx,**y**,incy) y(i) = sin(x(i))
vsngl	s	Double to single precision	call vsngl (n,**dx**,incx,**sy**,incy) sy(i) = sngl(dx(i))
_vsqrt	d,s	Element-wise square root	call dvsqrt (n,**x**,incx,**y**,incy) y(i) = sqrt(x(i))
_vsub	d,s,i	Vector subtraction	call dvsub (n,**x**,incx,**y**,incy,**z**,incz) z(i) = x(i) - y(i)

Table 4-2 (Page 7 of 7)
VecLib Routine Summary

Root Name	Data Type	Description	Calling Sequence/ FORTRAN Equivalent
_vvmvt	d,s	Vector minus vector quantity times vector	call dvvmvt(n,**w**,incw,**x**,incx,**y**,incy,**z**,incz) z(i) = (w(i) - x(i)) * y(i)
_vvpvt	d,s	Vector plus vector quantity times vector	call dvvpvt(n,**w**,incw,**x**,incx,**y**,incy,**z**,incz) z(i) = (w(i) + x(i)) * y(i)
_vvtvm	d,s	Vector times vector quantity minus vector	call dvvtvm(n,**w**,incw,**x**,incx,**y**,incy,**z**,incz) z(i) = w(i) * x(i) - y(i)
_vvtvp	d,s	Vector times vector quantity plus vector	call dvvtvp(n,**w**,incw,**x**,incx,**y**,incy,**z**,incz) z(i) = w(i) * x(i) + y(i)
_vvvtm	d,s	Vector minus quantity of vector times vector	call dvvvtm(n,**w**,incw,**x**,incx,**y**,incy,**z**,incz) z(i) = w(i) - x(i) * y(i)

_ASUM(3) iPSC/2 _ASUM(3)

Sum of the absolute values. (Function returning a value.)

Calling Sequence

Double precision:

```
DOUBLE PRECISION D, X(*), DASUM
INTEGER*4 N, INCX
    .
    .
    .
D = DASUM(N, X, INCX)
```

Single precision:

```
REAL S, X(*), SASUM
INTEGER*4 N, INCX
    .
    .
    .
S = SASUM(N, X, INCX)
```

FORTRAN Equivalent

```
      DASUM = 0.0
      DO 10 I = 1, N
         DASUM = DASUM + ABS(X(I))
10    CONTINUE
```

Description

$$\underline{d}asum = \sum_{i=1}^{n} | x(i) |$$

_AXPY(3) iPSC/2 _AXPY(3)

Scalar times a vector plus a vector to itself.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), ALPHA, Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DAXPY(N, ALPHA, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), ALPHA, Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SAXPY(N, ALPHA, X, INCX, Y, INCY)
```

FORTRAN Equivalent

```
     DO 10 I  = 1, N
        Y(I) = ALPHA * X(I) + Y(I)
10   CONTINUE
```

__CLIP(3) iPSC/2 __CLIP(3)

Clip to interval [alpha, beta]

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), ALPHA, Y(*), BETA
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DCLIP(N, X, INCX, ALPHA, BETA, Y, INCY)
```

Single precision:

```
REAL X(*), ALPHA, Y(*), BETA
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SCLIP(N, X, INCX, ALPHA, BETA, Y, INCY)
```

FORTRAN Equivalent

```
      DO 10 I  = 1, N
         Y(I)  = MIN (MAX(X(I), ALPHA), BETA)
10    CONTINUE
```

Description

$$y(i) = \begin{cases} \mathrm{ALPHA} & \text{IF } x(i) < \mathrm{ALPHA} \\ x(i) & \text{IF } \mathrm{ALPHA} \le x(i) \le \mathrm{BETA} \\ \mathrm{BETA} & \text{IF } x(i) > \mathrm{BETA} \end{cases}$$

Note: ALPHA must be less than or equal to BETA.

_CNDST(3) iPSC/2 _CNDST(3)

Conditional assignment.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Z(*)
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL DCNDST(N, X, INCX, LY, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Z(*)
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL SCNDST (N, X, INCX, LY, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
      DO 10 I  = 1, N
         IF (LY(I)) Z(I) = X(I)
10    CONTINUE
```

_COPY(3) iPSC/2 _COPY(3)

Copy vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DCOPY(N, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SCOPY(N, X, INCX, Y, INCY)
```

Integer:

```
INTEGER*4 X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL ICOPY(N, X, INCX, Y, INCY)
```

Logical:

```
LOGICAL*4 X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL LCOPY(N, X, INCX, Y, INCY)
```

FORTRAN Equivalent

```
      DO 10 I  = 1, N
         Y(I)  = X(I)
10    CONTINUE
```

_DOT(3) iPSC/2 _DOT(3)

Dot product of two vectors. (Function returning a value.)

Calling Sequence

Double precision:

```
DOUBLE PRECISION D, X(*), Y(*), DDOT
INTEGER*4 N, INCX, INCY
    .
    .
    .
D = DDOT(N, X, INCX, Y INCY)
```

Single precision:

```
REAL S, X(*), Y(*), SDOT
INTEGER*4 N, INCX, INCY
    .
    .
    .
S = SDOT(N, X, INCX, Y INCY)
```

FORTRAN Equivalent

```
      DDOT = 0.0
      DO 10 I = 1, N
         DDOT = DDOT + X(I)*Y(I)
10    CONTINUE
```

Description

$$\underline{d}dot = \sum_{i=1}^{n} (x(i) \cdot y(i))$$

_EQ(3) iPSC/2 _EQ(3)

Vector element equality.

Calling Sequence

Double precision:

```
DOUBLE PRECISION  X(*), Y(*)
LOGICAL*4 LZ(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL DEQ(N, X, INCX, Y, INCY, LZ, INCZ)
```

Single precision:

```
REAL  X(*), Y(*)
LOGICAL*4 LZ(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL SEQ(N, X, INCX, Y, INCY, LZ, INCZ)
```

Integer:

```
INTEGER*4 X(*), Y(*)
LOGICAL*4 LZ(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL IEQ(N, X, INCX, Y, INCY, LZ, INCZ)
```

FORTRAN Equivalent

```
     DO 10 I = 1, N
        LZ(I) = X(I) .EQ. Y(I)
10   CONTINUE
```

_FFT(3) iPSC/2 _FFT(3)

Fast Fourier Transform

Calling Sequence

Single Precision Complex:

```
COMPLEX*8 X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL CFFT (N, X, INCX, Y, INCY)
```

FORTRAN Equivalent

```
Y = FFT (X)
```

Description

One-dimensional FFT with a maximum vector length of 16384. "N" has to be a power of 2. The vectors "X" and "Y" may not overlap because the transform is not done in place.

_FILL(3) iPSC/2 _FILL(3)

Fill vector with scalar.

Calling Sequence

Double precision:

```
DOUBLE PRECISION ALPHA, X(*)
INTEGER*4 N, INCX
    .
    .
    .
CALL DFILL(N, ALPHA, X, INCX)
```

Single precision:

```
REAL ALPHA, X(*)
INTEGER*4 N, INCX
    .
    .
    .
CALL SFILL(N, ALPHA, X, INCX)
```

Integer:

```
INTEGER*4 ALPHA, X(*)
INTEGER*4 N, INCX
    .
    .
    .
CALL IFILL(N, ALPHA, X, INCX)
```

Logical:

```
LOGICAL*4 ALPHA, X(*)
INTEGER*4 N, INCX
    .
    .
    .
CALL LFILL(N, ALPHA, X, INCX)
```

FORTRAN Equivalent

```
     DO 10 I  = 1, N
        X(I)  = ALPHA
10   CONTINUE
```

_GATHR(3) iPSC/2 _GATHR(3)

Vector gather.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ, IY(*)
    .
    .
    .
CALL DGATHR(N, X, INCX, IY, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ, IY(*)
    .
    .
    .
CALL SGATHR(N, X, INCX, IY, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         Z(I) = X(IY(I))
10    CONTINUE
```

Description

NOTE: IY must contain values between 1 and 'N'.

_GE(3) iPSC/2 _GE(3)

Vector element greater than or equal to vector element.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
LOGICAL*4 LZ(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL DGE(N, X, INCX, Y, INCY, LZ, INCZ)
```

Single precision:

```
REAL X(*), Y(*)
LOGICAL*4 LZ(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL SGE(N, X, INCX, Y, INCY, LZ, INCZ)
```

Integer:

```
INTEGER*4 X(*), Y(*)
LOGICAL*4 LZ(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL IGE(N, X, INCX, Y, INCY, LZ, INCZ)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         LZ(I) = X(I) .GE. Y(I)
10    CONTINUE
```

_GT(3) iPSC/2 _GT(3)

Vector element greater than vector element.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
LOGICAL*4 LZ(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL DGT(N, X, INCX, Y, INCY, LZ, INCZ)
```

Single precision:

```
REAL X(*), Y(*)
LOGICAL*4 LZ(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL SGT(N, X, INCX, Y, INCY, LZ, INCZ)
```

Integer:

```
INTEGER*4 X(*), Y(*)
LOGICAL*4 LZ(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL IGT(N, X, INCX, Y, INCY, LZ, INCZ)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         LZ(I) = X(I) .GT. Y(I)
10    CONTINUE
```

I_AMAX(3) iPSC/2 I_AMAX(3)

Index of maximum absolute value. (Integer function returning a value.)

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*)
INTEGER*4 N, INCX, I, IDAMAX
    .
    .
    .
I = IDAMAX(N, X, INCX)
```

Single precision:

```
REAL X(*)
INTEGER*4 N, INCX, I, ISAMAX
    .
    .
    .
I = ISAMAX(N, X, INCX)
```

FORTRAN Equivalent

```
      IDAMAX = 0
      IF (N .GT. 0) IDAMAX = 1
      DO 10 I = 2, N
         IF (ABS(X(I)) .GT. ABS(X(IDAMAX)))  IDAMAX = I
10    CONTINUE
```

I_AMIN(3) iPSC/2 I_AMIN(3)

Index of minimum absolute value. (Integer function returning a value.)

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*)
INTEGER*4 N, INCX, I, IDAMIN
    .
    .
    .
I = IDAMIN(N, X, INCX)
```

Single precision:

```
REAL X(*)
INTEGER*4 N, INCX, I, ISAMIN
    .
    .
    .
I = ISAMIN(N, X, INCX)
```

FORTRAN Equivalent

```
      IDAMIN = 0
      IF (N .GT. 0) IDAMIN = 1
      DO 10 I= 2, N
         IF (ABS(X(I)) .LT. ABS(X(IDAMIN)))  IDAMIN = I
10    CONTINUE
```

_ICLIP(3) iPSC/2 _ICLIP(3)

Inverted clip.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), ALPHA, BETA, Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DICLIP(N, X, INCX, ALPHA, BETA, Y, INCY)
```

Single precision:

```
REAL X(*), ALPHA, BETA, Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SICLIP(N, X, INCX, ALPHA, BETA, Y, INCY)
```

FORTRAN Equivalent

```
      DO 10   I = 1, N
         IF   (X(I) .LT. (ALPHA + BETA)/2.0) THEN
              Y(I) = MIN(X(I), ALPHA)
         ELSE
              Y(I) = MAX(X(I), BETA)
         ENDIF
10    CONTINUE
```

Description

$$y(i) = \begin{cases} \text{ALPHA} & \text{IF ALPHA} < x(i) < (\text{ALPHA} + \text{BETA})/2.0 \\ \text{BETA} & \text{IF } (\text{ALPHA} + \text{BETA})/2.0 \leq x(i) < \text{BETA} \\ x(i) & \text{OTHERWISE} \end{cases}$$

Note: ALPHA must be less than or equal to BETA.

ICOUNT(3) iPSC/2 ICOUNT(3)

Number of logical true values. (Integer function returning a value.)

Calling Sequence

```
LOGICAL*4 LX (*)
INTEGER*4 N, INCX, I, ICOUNT
    .
    .
    .
I = ICOUNT (N, LX, INCX)
```

FORTRAN Equivalent

```
      ICOUNT = 0
      DO 10 I = 1, N
         IF (LX(I)) ICOUNT = ICOUNT + 1
10    CONTINUE
```

Inverse Fast Fourier Transform

Calling Sequence

Single Precision Complex:

```
COMPLEX*8 X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL CIFFT (N, X, INCX, Y, INCY)
```

FORTRAN Equivalent

$Y = FFT^{-1}(X)$

Description

One-dimensional FFT with a maximum vector length of 16384. "N" has to be a power of 2. The vectors "X" and "Y" may not overlap because the transform is not done in place.

ILAST(3) iPSC/2 ILAST(3)

Index of last logical true value. (Integer function returning a value.)

Calling Sequence

```
LOGICAL*4 LX (*)
INTEGER*4 N, INCX, I, ILAST
    .
    .
    .
I = ILAST (N, LX, INCX)
```

FORTRAN Equivalent

```
      ILAST = 0
      DO 10 I = 1, N
         IF (LX(I)) ILAST =  I
10    CONTINUE
```

I_MAX(3) iPSC/2 I_MAX(3)

Index of maximum value. (Integer function returning a value.)

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*)
INTEGER*4 N, INCX, I, IDMAX
    .
    .
    .
I = IDMAX(N, X, INCX)
```

Single precision:

```
REAL X(*)
INTEGER*4 N, INCX, I, ISMAX
    .
    .
    .
I = ISMAX(N, X, INCX)
```

FORTRAN Equivalent

```
      IDMAX = 0
      IF (N .GT. 0) IDMAX = 1
      DO 10 I = 2, N
         IF (X(I) .GT. X(IDMAX))  IDMAX = I
10    CONTINUE
```

I_MIN(3) iPSC/2 I_MIN(3)

Index of minimum value. (Integer function returning a value.)

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*)
INTEGER*4 N, INCX, I, IDMIN
    .
    .
    .
I = IDMIN(N, X, INCX)
```

Single precision:

```
REAL X(*)
INTEGER*4 N, INCX, I, ISMIN
    .
    .
    .
I = ISMIN(N, X, INCX)
```

FORTRAN Equivalent

```
      IDMIN = 0
      IF (N .GT. 0) IDMIN = 1
      DO 10 I = 2, N
         IF (X(I) .LT. X(IDMIN))  IDMIN = I
10    CONTINUE
```

LAND(3) iPSC/2 LAND(3)

Vector logical AND with vector

Calling Sequence

```
LOGICAL*4 X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL LAND (N, X, INCX, Y, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         Z(I) = X(I) .AND. Y(I)
10    CONTINUE
```

LANY(3) iPSC/2 LANY(3)

Logical true if any elements in X are true. (Logical function returning a value.)

Calling Sequence

```
LOGICAL*4 X(*), L, LANY
INTEGER*4 N, INCX
    .
    .
    .
L = LANY (N, X, INCX)
```

FORTRAN Equivalent

```
      LANY = .FALSE.
      DO 10 I = 1, N
         IF (X(I)) LANY = .TRUE.
10    CONTINUE
```

LNOT(3) iPSC/2 LNOT(3)

Vector logical negation

Calling Sequence

```
LOGICAL*4 X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL LNOT(N, X, INCX, Y, INCY)
```

FORTRAN Equivalent

```
     DO 10 I = 1, N
        Y(I) = .NOT. X(I)
10   CONTINUE
```

LOR(3) iPSC/2 LOR(3)

Vector logical OR with vector

Calling Sequence

```
LOGICAL*4 X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL LOR(N, X, INCX, Y, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         Z(I) =  X(I) .OR. Y(I)
10    CONTINUE
```

LSAND(3) iPSC/2 LSAND(3)

Scalar logical AND with vector

Calling Sequence

```
LOGICAL*4 X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY
   .
   .
   .
CALL LSAND (N, ALPHA, X, INCX, Y, INCY)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         Y(I) = ALPHA .AND. X(I)
10    CONTINUE
```

LSOR(3) iPSC/2 LSOR(3)

Scalar logical OR with vector.

Calling Sequence

```
LOGICAL*4 X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL LSOR(N, ALPHA, X, INCX, Y, INCY)
```

FORTRAN Equivalent

```
     DO 10 I = 1, N
        Y(I) =  ALPHA .OR. X(I)
10   CONTINUE
```

_MASK(3) iPSC/2 _MASK(3)

Conditional assignment.

Calling Sequence

Double precision:

```
DOUBLE PRECISION W(*), X(*), Z(*)
LOGICAL*4 LY(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ
    .
    .
    .
CALL DMASK(N, W, INCW, X, INCX, LY, INCY, Z, INCZ)
```

Single precision:

```
REAL W(*), X(*), Z(*)
LOGICAL*4 LY(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ
    .
    .
    .
CALL SMASK(N, W, INCW, X, INCX, LY, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
     DO 10 I = 1, N
        IF (LY(I)) THEN
              Z(I) = W(I)
        ELSE
              Z(I) = X(I)
        ENDIF
10   CONTINUE
```

_NE(3) iPSC/2 _NE(3)

Vector element inequality.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
LOGICAL*4 LZ(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL DNE(N, X, INCX, Y, INCY, LZ, INCZ)
```

Single precision:

```
REAL X(*), Y(*)
LOGICAL*4 LZ(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL SNE(N, X, INCX, Y, INCY, LZ, INCZ)
```

Integer:

```
INTEGER*4 X(*), Y(*)
LOGICAL*4 LZ(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL INE(N, X, INCX, Y, INCY, LZ, INCZ)
```

FORTRAN Equivalent

```
   DO 10 I = 1, N
      LZ(I) = X(I) .NE. Y(I)
10 CONTINUE
```

_NEG(3) iPSC/2 _NEG(3)

Change sign.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*)
INTEGER*4 N, INCX
    .
    .
    .
CALL DNEG(N, X, INCX)
```

Single precision:

```
REAL X(*)
INTEGER*4 N, INCX
    .
    .
    .
CALL SNEG(N, X, INCX)
```

Integer:

```
INTEGER*4 X(*)
INTEGER*4 N, INCX
    .
    .
    .
CALL INEG(N, X, INCX)
```

FORTRAN Equivalent

```
      DO 10 I  = 1, N
         X(I)  = -X(I)
10    CONTINUE
```

_NRM2(3) iPSC/2 _NRM2(3)

Euclidean vector norm. (Function returning a value)

Calling Sequence

Double precision:

```
DOUBLE PRECISION D, X(*), DNRM2
INTEGER*4 N, INCX
    .
    .
    .
D = DNRM2(N, X, INCX)
```

Single precision:

```
REAL S, X(*), SNRM2
INTEGER*4 N, INCX
    .
    .
    .
S = SNRM2(N, X, INCX)
```

FORTRAN Equivalent

```
      DNRM2 = 0.0
      DO 10 I = 1, N
         DNRM2 = DNRM2 + X(I)**2
10    CONTINUE
      DNRM2 = SQRT(DNRM2)
```

Description

$$DNRM2 = \sqrt{x(1)^2 + X(2)^2 + \ldots + x(n)^2}$$

Note: The summation of the squares operation is performed with scaling to protect from overflow or underflow of intermediate results.

_RAMP(3) iPSC/2 _RAMP(3)

Ramp function.

Calling Sequence

Double precision:

```
DOUBLE PRECISION ALPHA, BETA, X(*)
INTEGER*4 N, INCX
   .
   .
   .
CALL DRAMP(N, ALPHA, BETA, X, INCX)
```

Single precision:

```
REAL ALPHA, BETA, X(*)
INTEGER*4 N, INCX
   .
   .
   .
CALL SRAMP(N, ALPHA, BETA, X, INCX)
```

Integer:

```
INTEGER*4 ALPHA, BETA, X(*)
INTEGER*4 N, INCX
   .
   .
   .
CALL IRAMP(N, ALPHA, BETA, X, INCX)
```

FORTRAN Equivalent

```
     DO 10 I = 1, N
        X(I) = ALPHA + (I-1)*BETA
10   CONTINUE
```

_SADD(3) iPSC/2 _SADD(3)

Scalar plus vector

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DSADD(N, ALPHA, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SSADD(N, ALPHA, X, INCX, Y, INCY)
```

Integer:

```
INTEGER*4 X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL ISADD(N, ALPHA, X, INCX, Y, INCY)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         Y(I) = ALPHA + X(I)
10    CONTINUE
```

_SCAL(3) iPSC/2 _SCAL(3)

Scalar times a vector to itself.

Calling Sequence

Double precision:

```
DOUBLE PRECISION ALPHA, X(*)
INTEGER*4 N, INCX
    .
    .
    .
CALL DSCAL(N, ALPHA, X, INCX)
```

Single precision:

```
REAL ALPHA, X(*)
INTEGER*4 N, INCX
    .
    .
    .
CALL SSCAL(N, ALPHA, X, INCX)
```

FORTRAN Equivalent

```
     DO 10 I = 1, N
        X(I) = ALPHA * X(I)
10   CONTINUE
```

_SCATR(3) iPSC/2 _SCATR(3)

Vector scatter.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ, IY(*)
    .
    .
    .
CALL DSCATR(N, X, INCX, IY, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ, IY(*)
    .
    .
    .
CALL SSCATR(N, X, INCX, IY, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         Z(IY(I)) = X(I)
10    CONTINUE
```

Note: IY must contain values between 1 and 'N'. If the values of IY are not distinct, the result is undefined.

_SDIV(3) iPSC/2 _SDIV(3)

Scalar divided by vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DSDIV(N, ALPHA, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SSDIV(N, ALPHA, X, INCX, Y, INCY)
```

Integer:

```
INTEGER*4 X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL ISDIV(N, ALPHA, X, INCX, Y, INCY)
```

FORTRAN Equivalent

```
     DO 10 I = 1, N
        Y(I) = ALPHA / X(I)
10   CONTINUE
```

_SEQ(3) iPSC/2 _SEQ(3)

Vector equal to scalar.

Calling Sequence

Double precision:

```
DOUBLE PRECISION  X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DSEQ(N, ALPHA, X, INCX, LY, INCY)
```

Single precision:

```
REAL  X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SSEQ(N, ALPHA, X, INCX, LY, INCY)
```

Integer:

```
INTEGER*4 X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL ISEQ(N, ALPHA, X, INCX, LY, INCY)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         LY(I) = ALPHA .EQ. X(I)
10    CONTINUE
```

_SGE(3) iPSC/2 _SGE(3)

Scalar greater than or equal to vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DSGE(N, ALPHA, X, INCX, LY, INCY)
```

Single precision:

```
REAL X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SSGE(N, ALPHA, X, INCX, LY, INCY)
```

Integer:

```
INTEGER*4 X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL ISGE(N, ALPHA, X, INCX, LY, INCY)
```

FORTRAN Equivalent

```
     DO 10 I = 1, N
        LY(I) = ALPHA .GE. X(I)
10   CONTINUE
```

_SGT(3) iPSC/2 _SGT(3)

Scalar greater than vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DSGT(N, ALPHA, X, INCX, LY, INCY)
```

Single precision:

```
REAL X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SSGT(N, ALPHA, X, INCX, LY, INCY)
```

Integer:

```
INTEGER*4 X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL ISGT(N, ALPHA, X, INCX, LY, INCY)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         LY(I) = ALPHA .GT. X(I)
10    CONTINUE
```

_SLE(3) iPSC/2 _SLE(3)

Scalar less than or equal to vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DSLE(N, ALPHA, X, INCX, LY, INCY)
```

Single precision:

```
REAL X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SSLE(N, ALPHA, X, INCX, LY, INCY)
```

Integer:

```
INTEGER*4 X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL ISLE(N, ALPHA, X, INCX, LY, INCY)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         LY(I) = ALPHA .LE. X(I)
10    CONTINUE
```

__SLT(3) iPSC/2 __SLT(3)

Scalar less than vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DSLT(N, ALPHA, X, INCX, LY, INCY)
```

Single precision:

```
REAL X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SSLT(N, ALPHA, X, INCX, LY, INCY)
```

Integer:

```
INTEGER*4 X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL ISLT(N, ALPHA, X, INCX, LY, INCY)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         LY(I) = ALPHA .LT. X(I)
10    CONTINUE
```

_SMUL(3) iPSC/2 _SMUL(3)

Scalar times vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DSMUL(N, ALPHA, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SSMUL(N, ALPHA, X, INCX, Y, INCY)
```

Integer:

```
INTEGER*4 X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL ISMUL(N, ALPHA, X, INCX, Y, INCY)
```

FORTRAN Equivalent

```
    DO 110 I = 1, N
       Y(I) = ALPHA * X(I)
110 CONTINUE
```

_SNE(3) iPSC/2 _SNE(3)

Vector not equal to scalar.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DSNE(N, ALPHA, X, INCX, LY, INCY)
```

Single precision:

```
REAL X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SSNE(N, ALPHA, X, INCX, LY, INCY)
```

Integer:

```
INTEGER*4 X(*), ALPHA
LOGICAL*4 LY(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL ISNE(N, ALPHA, X, INCX, LY, INCY)
```

FORTRAN Equivalent

```
     DO 10 I = 1, N
        LY(I) = ALPHA .NE. X(I)
10   CONTINUE
```

_SSUB(3) iPSC/2 _SSUB(3)

Scalar vector subtraction

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DSSUB(N, ALPHA, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SSSUB(N, ALPHA, X, INCX, Y, INCY)
```

Integer:

```
INTEGER*4 X(*), Y(*), ALPHA
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL ISSUB(N, ALPHA, X, INCX, Y, INCY)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         Y(I) = ALPHA - X(I)
10    CONTINUE
```

_SUM(3) iPSC/2 _SUM(3)

Vector sum. (Function returning a value.)

Calling Sequence

Double precision:

```
DOUBLE PRECISION D, X(*), DSUM
INTEGER*4 N, INCX
    .
    .
    .
D = DSUM(N, X, INCX)
```

Single precision:

```
REAL S, X(*), SSUM
INTEGER*4 N, INCX
    .
    .
    .
S = SSUM(N, X, INCX)
```

FORTRAN Equivalent

```
      DSUM = 0.0
      DO 10 I = 1, N
         DSUM = DSUM + X(I)
10    CONTINUE
```

Description

$$\underline{d}sum = \sum_{i=1}^{n} x(i)$$

_SVMVT(3) iPSC/2 _SVMVT(3)

Scalar minus vector quantity times vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL DSVMVT(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL SSVMVT(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
      DO 10 I =  1, N
         Z(I) = (ALPHA - X(I)) *  Y(I)
10    CONTINUE
```

_SVPVT(3) iPSC/2 _SVPVT(3)

Scalar plus vector quantity times vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL DSVPVT(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL SSVPVT(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
      DO 10 I =  1, N
         Z(I) = (ALPHA + X(I)) *  Y(I)
10    CONTINUE
```

_SVTSP(3) iPSC/2 _SVTSP(3)

Scalar times vector quantity plus scalar.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), ALPHA, BETA
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DSVTSP(N, ALPHA, BETA, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*), ALPHA, BETA
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SSVTSP(N, ALPHA, BETA, X, INCX, Y, INCY)
```

FORTRAN Equivalent

```
      DO 10 I =  1, N
         Y(I) = ALPHA * X(I) + BETA
10    CONTINUE
```

_SVTVM(3) iPSC/2 _SVTVM(3)

Scalar times vector quantity minus vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL DSVTVM(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL SSVTVM(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
      DO 10 I =  1, N
         Z(I) = ALPHA * X(I) -  Y(I)
10    CONTINUE
```

_SVTVP(3) iPSC/2 _SVTVP(3)

Scalar times vector quantity plus vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL DSVTVP(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL SSVTVP(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
     DO 10 I =  1, N
        Z(I) = ALPHA * X(I) +  Y(I)
10   CONTINUE
```

_SVVMT(3) iPSC/2 _SVVMT(3)

Scalar times the quantity of vector minus vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL DSVVMT(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL SSVVMT(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
      DO 10 I =  1, N
         Z(I) = ALPHA * (X(I) -  Y(I))
10    CONTINUE
```

Scalar times the quantity of vector plus vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL DSVVPT(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL SSVVPT(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
      DO 10 I =  1, N
         Z(I) = ALPHA * (X(I) +  Y(I))
10    CONTINUE
```

_SVVTM(3) iPSC/2 _SVVTM(3)

Scalar minus the quantity of vector times vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION  X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL DSVVTM(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL SSVVTM(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
     DO 10 I =  1, N
        Z(I) = ALPHA - X(I) *  Y(I)
10   CONTINUE
```

_SVVTP(3) iPSC/2 _SVVTP(3)

Scalar plus the quantity of vector times vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL DSVVTP(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*), ALPHA
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL SSVVTP(N, ALPHA, X, INCX, Y, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
     DO 10 I =  1, N
        Z(I) = ALPHA + X(I) *  Y(I)
10   CONTINUE
```

_SWAP(3) iPSC/2 _SWAP(3)

Swap vectors.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DSWAP(N, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SSWAP(N, X, INCX, Y, INCY)
```

FORTRAN Equivalent

```
     DO 10 I = 1, N
        T = Y(I)
        Y(I) = X(I)
        X(I) = T
10   CONTINUE
```

Description

y(i) <=> x(i)

_VABS(3) iPSC/2 _VABS(3)

Element-wise absolute value.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DVABS (N, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SVABS(N, X, INCX, Y, INCY)
```

Integer:

```
INTEGER*4 X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL IVABS(N, X, INCX, Y, INCY)
```

FORTRAN Equivalent

```
      DO 10 I  = 1, N
         Y(I)  = ABS (X(I))
10    CONTINUE
```

Description

$y(i) = | x(i) |$

_VADD(3) iPSC/2 _VADD(3)

Vector addition.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL DVADD(N, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL SVADD(N, X, INCX, Y, INCY, Z, INCZ)
```

Integer:

```
INTEGER*4 X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL IVADD(N, X, INCX, Y, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
     DO 10 I = 1, N
        Z(I) = X(I) + Y(I)
10   CONTINUE
```

_VAMAX(3) iPSC/2 _VAMAX(3)

Vector element-wise maximum absolute value.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL DVAMAX(N, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL SVAMAX(N, X, INCX, Y, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         Z(I) = MAX(ABS(X(I)),ABS(Y(I)))
10    CONTINUE
```

Description

$z(i) = \max(|x(i)|, |y(i)|)$

_VAMIN(3) iPSC/2 _VAMIN(3)

Vector element-wise minimum absolute value.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL DVAMIN(N, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL SVAMIN(N, X, INCX, Y, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         Z(I) = MIN(ABS(X(I)),ABS(Y(I)))
10    CONTINUE
```

Description

$z(i) = \min(|x(i)|, |y(i)|)$

_VATAN(3) iPSC/2 _VATAN(3)

Element-wise inverse-tangent.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DVATAN(N, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SVATAN(N, X, INCX, Y, INCY)
```

FORTRAN Equivalent

```
     DO 10 I = 1, N
        Y(I) = ATAN (X(I))
10   CONTINUE
```

Description

y(i) = arctangent (in radians) of x(i)

_VATN2(3) iPSC/2 _VATN2(3)

Element-wise inverse-tangent.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL DVATN2(N, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL SVATN2(N, X, INCX, Y, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
     DO 10 I = 1, N
        Z(I) = ATAN2 (X(I),Y(I))
10   CONTINUE
```

Description

z(i) = arctangent (in radians) of x(i)/y(i)

_VCOS(3) iPSC/2 _VCOS(3)

Element-wise cosine.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DVCOS(N, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SVCOS(N, X, INCX, Y, INCY)
```

FORTRAN Equivalent

```
     DO 10 I = 1, N
        Y(I) = COS(X(I))
10   CONTINUE
```

Description

y(i) = cosine of x(i) in radians.

VDBLE(3) iPSC/2 VDBLE(3)

Single to double precision.

Calling Sequence

```
DOUBLE PRECISION DY(*)
INTEGER*4 N, INCX, INCZ
REAL SX(*)
   .
   .
   .
CALL VDBLE(N, SX, INCX, DY, INCY)
```

FORTRAN Equivalent

```
     DO 10 I = 1, N
        DY(I) = DBLE(SX(I))
10   CONTINUE
```

Description

dy(i) = conversion of sx(i) to double precision

Element-wise vector divison.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL DVDIV(N, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL SVDIV(N, X, INCX, Y, INCY, Z, INCZ)
```

Integer:

```
INTEGER*4 X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL IVDIV(N, X, INCX, Y, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         Z(I) = X(I) / Y(I)
10    CONTINUE
```

_VEXP(3) iPSC/2 _VEXP(3)

Element-wise exponential.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DVEXP(N, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SVEXP(N, X, INCX, Y, INCY)
```

FORTRAN Equivalent

```
     DO 10 I = 1, N
        Y(I) = EXP(X(I))
10   CONTINUE
```

Description

$y(i) = e^{x(i)}$

_VFIX(3) iPSC/2 _VFIX(3)

Truncate elements to integer values.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*)
INTEGER*4 N, IY(*), INCX, INCY
    .
    .
    .
CALL DVFIX(N, X, INCX, IY, INCY)
```

Single precision:

```
REAL X(*)
INTEGER*4 N, IY(*), INCX, INCY
    .
    .
    .
CALL SVFIX(N, X, INCX, IY, INCY)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         IY(I) = INT(X(I))
10    CONTINUE
```

Description

iy(i) = value resulting from truncation of the fractional part of x(i)

_VFLOA(3) iPSC/2 _VFLOA(3)

Convert integer to floating point.

Calling Sequence

Double precision:

```
DOUBLE PRECISION Y(*)
INTEGER*4 N, INCX, INCY, IX(*)
    .
    .
    .
CALL DVFLOA(N, IX, INCX, Y, INCY)
```

Single precision:

```
REAL Y(*)
INTEGER*4 N, INCX, INCY, IX(*)
    .
    .
    .
CALL SVFLOA(N, IX, INCX, Y, INCY)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         Y(I) = IX(I)
10    CONTINUE
```

Description

y(i) = conversion of ix(i) to floating point

_VLG10(3) iPSC/2 _VLG10(3)

Element-wise base 10 logarithm.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DVLG10(N, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SVLG10(N, X, INCX, Y, INCY)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         Y(I) = LOG10(X(I))
10    CONTINUE
```

Description

$y(i) = \log_{10}(x(i))$

_VLOG(3) iPSC/2 _VLOG(3)

Element-wise natural logarithm.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DVLOG(N, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SVLOG(N, X, INCX, Y, INCY)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         Y(I) = LOG(X(I))
10    CONTINUE
```

Description

y(i) = ln (x(i))

_VMAX(3) iPSC/2 _VMAX(3)

Vector element-wise maximum.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL DVMAX(N, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL SVMAX(N, X, INCX, Y, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
     DO 10 I = 1, N
        Z(I) = MAX(X(I),Y(I))
10   CONTINUE
```

_VMIN(3) iPSC/2 _VMIN(3)

Vector element-wise minimum.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL DVMIN(N, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL SVMIN(N, X, INCX, Y, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         Z(I) = MIN(X(I),Y(I))
10    CONTINUE
```

_VMUL(3) iPSC/2 _VMUL(3)

Element-wise vector multiplication.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
   .
   .
   .
CALL DVMUL(N, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
   .
   .
   .
CALL SVMUL(N, X, INCX, Y, INCY, Z, INCZ)
```

Integer:

```
INTEGER*4 X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
   .
   .
   .
CALL IVMUL(N, X, INCX, Y, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         Z(I) = X(I) * Y(I)
10    CONTINUE
```

_VNEG(3) iPSC/2 _VNEG(3)

Negate vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DVNEG(N, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SVNEG(N, X, INCX, Y, INCY)
```

Integer:

```
INTEGER*4 X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL IVNEG(N, X, INCX, Y, INCY)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         Y(I) = -X(I)
10    CONTINUE
```

_VPOLY(3) iPSC/2 _VPOLY(3)

Vector polynomial evaluation.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), C(*), Y(*)
INTEGER*4 N, M, INCX, INCC, INCY
    .
    .
    .
CALL DVPOLY(N, X, INCX, M, C, INCC, Y, INCY)
```

Single precision:

```
REAL X(*), C(*), Y(*)
INTEGER*4 N, M, INCX, INCC, INCY
    .
    .
    .
CALL SVPOLY(N, X, INCX, M,  C, INCC, Y, INCY)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         Y(I) = C(M)
10    CONTINUE
      DO 30 J = M-1, 1, -1
         DO 20 I = 1, N
            Y(I) = X(I)*Y(I) + C(J)
20       CONTINUE
30    CONTINUE
```

Description

C is a vector of length 'M' containing the coefficients of the polynomial.

Vector polynominal evaluation

$y(i) = c(1) + c(2)x(i) + c(3)x(i)^2 ... + c(m)x(i)^{m-1}$

NOTE: If $m \leq 0$ $y(i) = 0.0$ for $i = 1, 2, \ldots, n$

_VPOW(3) iPSC/2 _VPOW(3)

Element-wise power function.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL DVPOW(N, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL SVPOW(N, X, INCX, Y, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         Z(I) = X(I) ** Y(I)
10    CONTINUE
```

Description

$z(i) = x(i)^{y(i)}$

Note: Exponentiation is performed using logarithms. X(i) cannot be a negative number since logarithms of negative values are undefined. Thus, x(i) > 0. Also, if x(i) = 0, y(i) >0 or division by 0 results.

_VRECP(3) iPSC/2 _VRECP(3)

Vector reciprocal.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DVRECP(N, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SVRECP(N, X, INCX, Y, INCY)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         Y(I) = 1.0 / X(I)
10    CONTINUE
```

_VSIN(3) iPSC/2 _VSIN(3)

Element-wise sine.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DVSIN(N, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SVSIN(N, X, INCX, Y, INCY)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
        Y(I) = SIN(X(I))
10    CONTINUE
```

Description

y(i) = sine of x(i) in radians

VSNGL(3) iPSC/2 VSNGL(3)

Double to single precision.

Calling Sequence

```
DOUBLE PRECISION DX(*)
INTEGER*4 N, INCX, INCY
REAL SY(*)
    .
    .
    .
CALL VSNGL(N, DX, INCX, SY, INCY)
```

FORTRAN Equivalent

```
     DO 10 I = 1, N
        SY(I) = SNGL(DX(I))
10   CONTINUE
```

Description

sy(i) = conversion of dx(i) to type real.

_VSQRT(3) iPSC/2 _VSQRT(3)

Element-wise square root.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL DVSQRT(N, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, INCX, INCY
    .
    .
    .
CALL SVSQRT(N, X, INCX, Y, INCY)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         Y(I) = SQRT(X(I))
10    CONTINUE
```

Description

$$y(i) = \sqrt{x(i)}$$

_VSUB(3) iPSC/2 _VSUB(3)

Vector subtraction.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL DVSUB(N, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL SVSUB(N, X, INCX, Y, INCY, Z, INCZ)
```

Integer:

```
INTEGER*4 X(*), Y(*), Z(*)
INTEGER*4 N, INCX, INCY, INCZ
    .
    .
    .
CALL IVSUB(N, X, INCX, Y, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
      DO 10 I =  1, N
         Z(I) = X(I) - Y(I)
10    CONTINUE
```

_VVMVT(3) iPSC/2 _VVMVT(3)

Vector minus vector quantity times vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION W(*), X(*), Y(*), Z(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ
    .
    .
    .
CALL DVVMVT(N, W, INCW, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL W(*), X(*), Y(*), Z(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ
    .
    .
    .
CALL SVVMVT(N, W, INCW, X, INCX, Y, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
      DO 10 I =  1, N
         Z(I) = (W(I) - X(I)) *  Y(I)
10    CONTINUE
```

_VVPVT(3) iPSC/2 _VVPVT(3)

Vector plus vector quantity times vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION W(*), X(*), Y(*), Z(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ
    .
    .
    .
CALL DVVPVT(N, W, INCW, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL  W(*),X(*), Y(*), Z(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ
    .
    .
    .
CALL SVVPVT(N, W, INCW, X, INCX, Y, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
      DO 10 I = 1, N
         Z(I) = (W(I) + X(I)) *  Y(I)
10    CONTINUE
```

_VVTVM(3) iPSC/2 _VVTVM(3)

Vector times vector quantity minus vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION W(*), X(*), Y(*), Z(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ
    .
    .
    .
CALL DVVTVM(N, W, INCW, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL W(*), X(*), Y(*), Z(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ
    .
    .
    .
CALL SVVTVM(N, W, INCW, X, INCX, Y, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
      DO 10 I =  1, N
         Z(I) = W(I) * X(I) -  Y(I)
10    CONTINUE
```

_VVTVP(3) iPSC/2 _VVTVP(3)

Vector times vector quantity plus vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION W(*), X(*), Y(*), Z(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ
    .
    .
    .
CALL DVVTVP(N, W, INCW, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL W(*), X(*), Y(*), Z(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ
    .
    .
    .
CALL SVVTVP(N, W, INCW, X, INCX, Y, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
      DO 10 I =  1, N
         Z(I) = W(I) * X(I) +  Y(I)
10    CONTINUE
```

_VVVTM(3) iPSC/2 _VVVTM(3)

Vector minus the quantity of vector times vector.

Calling Sequence

Double precision:

```
DOUBLE PRECISION W(*), X(*), Y(*), Z(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ
    .
    .
    .
CALL DVVVTM(N, W, INCW, X, INCX, Y, INCY, Z, INCZ)
```

Single precision:

```
REAL W(*), X(*), Y(*), Z(*)
INTEGER*4 N, INCW, INCX, INCY, INCZ
    .
    .
    .
CALL SVVVTM(N, W, INCW, X, INCX, Y, INCY, Z, INCZ)
```

FORTRAN Equivalent

```
      DO 10 I =  1, N
         Z(I) = W(I) - X(I) *  Y(I)
10    CONTINUE
```

APPENDIX A

TYPESEL MASK

The TYPESEL argument is an integer value that specifies the TYPE(s) of message that 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 TYPE's 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 TYPE's 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 argument negative and indicates that it is a mask.

To generate a mask, add the numbers associated with the TYPE's you want to select to the number that corresponds to bit **31** (See Table A-1). There are a number of ways to construct a TYPESEL mask. The following examples show three different ways you could construct a TYPESEL mask to select message types **1**, **2**, **5**, and **12**.

```
ITYPESEL = 2 + 4 + 32 + 4096 + 2147483648
```

or

```
ITYPESEL = Z'2' = Z'4' + Z'20' + Z'1000' + Z'80000000'
```

or

```
ITYPESEL = 2**1 + 2**2 + 2**5 + 2**12 + 2**31
```

then enter:

```
CALL CRECV( ITYPESEL, BUF, LEN )
```

The following table gives the decimal and hexadecimal numbers associated with bit positions 0-31:

Table A-1
Number Representation

Type	Decimal Number	Hexadecimal Number
0	1	1
1	2	2
2	4	4
3	8	8
4	16	10
5	32	20
6	64	40
7	128	80
8	256	100
9	512	200
10	1024	400
11	2048	800
12	4096	1000
13	8192	2000
14	16384	4000
15	32768	8000
16	65536	10000
17	131072	20000
18	262144	40000
19	524288	80000
20	1048576	100000
21	2097152	200000
22	4194304	400000
23	8388608	800000
24	16777216	1000000
25	33554432	2000000
26	67108864	4000000
27	134217728	8000000
28	268435456	10000000
29	536870912	20000000
30	1073741824	40000000
31	2147483648	80000000

APPENDIX B

FORTRAN COMMUNICATION UTILITIES

DESCRIPTION

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.

Global Commutative Operations (cont.)

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.

DESCRIPTION (cont.)

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 f77 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.

```
      INTEGER I, MYNODE, CNT
      DOUBLE PRECISION X(M), Y(N), DOT, NORM, DUMMY
      PARAMETER (DPSIZE = 8)
C
C
C  Assume that X, M, and N have been initialized
C
C
C  Compute the local dot product
C
      DOT = 0
      DO 1 I = 1, M
      DOT = DOT + X(I)*X(I)
    1 CONTINUE
C
C  Sum global contributions
C
      CALL gdsum (DOT, 1, DUMMY)
C
C  Compute global NORM
C
      NORM = DSQRT(DOT)
C
C  Do the local normalization
C
      DO 2 I = 1, M
      X(I) = X(I)/NORM
    2 CONTINUE
C
C  Collect vector
C
      CALL 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

```
subroutine gcol(x, xlen, y, ylen, ncnt)
character x(xlen)
integer xlen
character y(ylen)
integer ylen
integer ncnt
```

Description

x the input buffer to be used in the operation.

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

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

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

ncnt 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 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.

GDHIGH(3) iPSC/2 GDHIGH(3)

Global vector double precision MAX operation.

Synopsis

```
subroutine gdhigh(x, n, work)
double precision x(*)
integer n
double precision work(*)
```

Description

x 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 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.

GDLOW(3) iPSC/2 GDLOW(3)

Global vector double precision MIN operation.

Synopsis

```
subroutine gdlow(x, n, work)
double precision x(*)
integer n
double precision work(*)
```

Description

x 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 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

gdhigh:received message too long for buffer	Make sure vector length is the same for all nodes.
gdhigh:received message too short for buffer	Make sure vector length is the same for all nodes.
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.

GDPROD(3) iPSC/2 GDPROD(3)

Global vector double precision MULTIPLY operation.

Synopsis

```
subroutine gdprod(x, n, work)
double precision x(*)
integer n
double precision work(*)
```

Description

x 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 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

```
subroutine gdsum(x, n, work)
double precision x(*)
integer n
double precision work(*)
```

Description

x 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 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

```
subroutine giand(x, n, work)
integer x(*)
integer n
integer work(*)
```

Description

x 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 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

```
subroutine gihigh(x, n, work)
integer x(*)
integer n
integer work(*)
```

Description

x   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   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

```
subroutine gilow(x, n, work)
integer x(*)
integer n
integer work(*)
```

Description

x 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 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

```
subroutine gior(x, n, work)
integer x(*)
integer n
integer work(*)
```

Description

x 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 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

```
subroutine giprod(x, n, work)
integer x(*)
integer n
integer work(*)
```

Description

x 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 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

```
subroutine gisum(x, n, work)
integer x(*)
integer n
integer work(*)
```

Description

x 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 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

```
subroutine gixor(x, n, work)
integer x(*)
integer n
integer work(*)
```

Description

x 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 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

```
subroutine gland(x, n, work)
logical x(*)
integer n
logical work(*)
```

Description

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

Gland calculates the logical 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.

GLOR(3) iPSC/2 GLOR(3)

Global vector logical inclusive OR operation.

Synopsis

```
subroutine glor(x, n, work)
logical x(*)
integer n
logical work(*)
```

Description

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

Glor calculates the logical 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

```
subroutine glxor(x, n, work)
logical x(*)
integer n
logical work(*)
```

Description

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

Glxor calculates the logical 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.

GOPF(3) iPSC/2 GOPF(3)

Arbitrary commutative function.

Synopsis

```
subroutine gopf(x, n, work, f)
integer x(*)
integer n
integer work(*)

c   f is a subroutine
c   f must be declared external in the calling program
```

Description

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

f the subroutineto 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:

```
      external max_node
c
c Assume n has been initialized
c
      real mine(2), work(2), x(n)
      integer i, max_loc

      mine(1) = abs(x(1))
      mine(2) = mynode()
      max_loc = 1
c
c Find the maximum and save the index of the maximum
c
      do 2 i = 2, n
              if(mine(1) .lt. abs(x(i))) then
                      max_loc = i
                      mine(1) = abs(x(loc))
              endif
    2 continue
      call gopf(mine, n, work, max_node)
      if (mine(1) .eq. mynode) then
c             I have the maximum
      ...
      endif

      ...

      return
      end
```

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

```
    subroutine max_node(mine, n, work)
             integer n
             real mine(n), work(n)

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

GSENDX(3) iPSC/2 GSEND(3)

Send a vector to a list of nodes.

Synopsis

```
subroutine gsendx(type, x, len, nodenums, nlen)
integer type
integer x(*)
integer len
integer nodenums(*)
integer 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	the input vector to be sent.
len	the number of bytes of data in vector.
nodenums	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

```
subroutine gshigh(x, n, work)
real x(*)
integer n
real work(*)
```

Description

x — 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 — 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.

GSLOW(3) iPSC/2 GSLOW(3)

Global vector real MIN operation.

Synopsis

```
subroutine gslow(x, n, work)
real x(*)
integer n
real work(*)
```

Description

x 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 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

```
subroutine gsprod(x, n, work)
real x(*)
integer n
real work(*)
```

Description

x 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 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

```
subroutine gssum(x, n, work)
real x(*)
integer n
real work(*)
```

Description

x 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 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.

GSYNC(3) iPSC/2 GSYNC(3)

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.