July 1990
Order Number: 311767-002

iPSC®/2 ADA
PROGRAMMER'S REFERENCE MANUAL

intel® Corporation

REV.	REVISION HISTORY	DATE
-001	Original Issue	03/89
-002	Revision	07/90

RESTRICTED RIGHTS

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

PREFACE

PURPOSE/SCOPE

This manual is the programmer's reference manual for Ada on iPSC®/2 and iPSC/860 systems. The manual assumes that you are an application programmer proficient in the Ada language and the UNIX operating system. The manual provides syntax information and descriptions of the Ada system calls.

ORGANIZATION

Chapter 1	Introduces Ada in the iPSC/2 and iPSC/860 environments and describes iPSC/2 and iPSC/860 message-passing in an Ada application.
Chapter 2	Describes the calls that are used by both nontasking (i.e., single-task) and tasking (i.e., multiple-task) Ada programs.
Chapter 3	Describes the calls that are specific to single-task Ada programs.
Chapter 4	Describes the calls that are specific to multiple-task Ada programs.
Chapter 5	Describes the cross-debugger commands.
Appendix A	Summarizes the syntax of the common, nontasking, and tasking Ada library calls.

APPLICABLE DOCUMENTS

For more information, refer to these other members of the iPSC/2 and iPSC/860 manual set:

iPSC®/2 Ada Program Development Guide
Describes the iPSC/2 Ada development environment.

iPSC®/2 and iPSC®/860 C Language Reference Manual
Describes the C compiler for the iPSC/2 and iPSC/860 system.

iPSC®/2 and iPSC®/860 DECON User's Guide
Tells how to use DECON, the iPSC/2 and iPSC/860 concurrent debugger.

iPSC®/2 and iPSC®/860 Fortran Language Reference Manual
Describes the Fortran compiler for the iPSC/2 and iPSC/860 system.

iPSC®/2 and iPSC®/860 Programmer's Reference Manual
Describes iPSC/2 and iPSC/860 commands, C system calls, and Fortran system calls.

iPSC®/2 and iPSC®/860 Simulator Manual
Tells how to use the iPSC/2 and iPSC/860 Simulator for software development.

iPSC®/2 and iPSC®/860 System Administrator's Guide
Describes the system administration tasks related to operating and maintaining an iPSC/2 and iPSC/860 system.

iPSC®/2 and iPSC®/860 User's Guide
Tells how to use the iPSC/2 and iPSC/860 systems.

iPSC®/2 Lisp Language Reference Manual
Describes the Lisp implementation that runs on the iPSC/2 nodes and its extensions.

iPSC®/2 Lisp Programmer's Reference Manual
Describes the iPSC/2 Lisp user interface and the iPSC/2 Lisp concurrent constructs.

UNIX System V Manual Set
Describes the UNIX System V operating system.

NOTATIONAL CONVENTIONS

This section describes the notational conventions for the following:

- Type style usage
- Examples in text
- System call syntax

Type Style Usage

This manual uses the following type style conventions:

- **Bold** type style is used for anything that must be entered exactly as shown, including:

  Command names (including absolute pathname versions)
  Switches, flags, and options
  System Call names
  Routine names (predefined *and* user-defined)
  Reserved words

- *Italic* type style is used for:

  Variables
  File names (including absolute pathname versions)
  Directory names
  Process names
  User names
  Emphasis

- ***Bold-italic*** type style is used for user input described in text (what you enter in response to some prompt).

- `Plain-Monospace` type style is used for:

  Computer output (prompts and messages)
  Code
  Error messages
  Values of variables

- ***Bold-Italic-Monospace*** type style is used for user input displayed in an example (what you enter in response to some prompt)

- **Bold-Monospace** type style is used for the names of keyboard keys (which are also enclosed in angle brackets). For example:

<Alt>	**<Backspace>**
<Break>	**<Ctrl>**
<Delete> or **<Del>**	**<Enter>**
<Esc>	**<Tab>**

For a key that prints, the result of pressing the key is used. For example, **s** means press the key labeled **<S>**, and **S** means press and hold the **<Shift>** key while pressing the **<S>** key.

A dash indicates that the key preceding the dash is to be held down *while* the key following the dash is pressed. This has the effect of producing a single keystroke (which is why there is only one set of angle brackets). For example:

<Ctrl-S>
<Ctrl-@>
<Ctrl-Alt-Del>

Note that there are several ways to reference some keys. For example, **S** and **<Shift-s>** mean the same thing. In such cases, the simplest method (**S**) is usually used.

Examples in Text

This manual uses `monospace` type style for examples.

In examples of interactive sessions, user input appears in ***bold-italic-monospace*** type style to distiguish it from computer output. For example:

```
# getcube -tdiag                                    (user input)
getcube successful: cube type diag allocated        (computer output)
```

Many examples contain annotations that describe specific parts of the example. These annotations (which are not part of the example code or session) appear in *italic* type style.

System Call Syntax

You can use iPSC/2 system calls in programs in the same way that you use other system calls. In this manual, system calls are described as follows:

Synopsis

return_type **name** (*parameters*)

Note that the return type appears in regular type, the name of the call appears in bold type, the data type of the parameters appears in bold type, and the parameters appear in italic type. For example, the following is the synopsis for **getcube**():

```
procedure getcube(      cubename : in string;
                        cubetype : in string;
                        hostname : in string;
                            keep : in integer);
```

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

CHAPTER 2
COMMON LIBRARY CALLS

CHAPTER 3
NONTASKING LIBRARY CALLS

CHAPTER 4
TASKING LIBRARY CALLS

CHAPTER 5
CROSS-DEBUGGER COMMANDS

APPENDIX A
SUMMARY OF ADA LIBRARY CALLS

LIST OF TABLES

INTRODUCTION 1

iPSC/2 Ada is part of the Concurrent Workbench™. Ada includes facilities that enable it to interact with C, Fortran, and Lisp, which are also part of the Concurrent Workbench. Ada programs can be run on both a host such as the System Resource Manager (SRM) and on the cube nodes.

The model of execution is one Ada program per processor. On the SRM, each Ada program is mapped to a Unix process. On the cube, each Ada program is mapped to an NX/2 process.

The routines in the Ada system's host and node interface libraries let you perform tasks such as:

- Accessing and releasing cubes
- Loading, starting, and killing processes
- Passing messages between processes

Tasks, which are handled by the Ada runtime system, are not visible to the operating system. The Ada rendezvous is supported only between tasks on the same processor. Use iPSC/2 message-passing to communicate between tasks on different processors.

MESSAGE-PASSING LIBRARIES

Two sets of iPSC/2 message-passing libraries are provided:

- The nontasking library is for applications that contain only a single task per program. The routines for host programs reside in the *host_ipsc* library. The routines for node programs reside in the *node_ipsc* and *commutl* libraries:
  - The *node_ipsc* library contains node message-passing programs.
  - The *commutl* library contains global communication utilities.

- The tasking library is for applications that contain several tasks per program. The routines for host programs reside in the *thost_ipsc* library. The routines for node programs reside in the *tnode_ipsc* and *tcommutl* libraries:
  - The *tnode_ipsc* library contains node message-passing programs.
  - The *tcommutl* library contains global communication utilities.

The NX/2 operating system sends messages to and from processes running on the nodes. This means that it can send messages to and from iPSC/2 Ada programs, but not to and from individual Ada tasks. The libraries that allow message-passing between Ada tasks on different processors extend the basic message-passing constructs and incur additional overhead. For applications that contain only a single task per program, the nontasking versions of the libraries are more efficient. Use the same library throughout the application.

To make the iPSC message-passing libraries available, you must:

1. Install the standard VADS libraries in your directory:

```
a.mklib -i
```

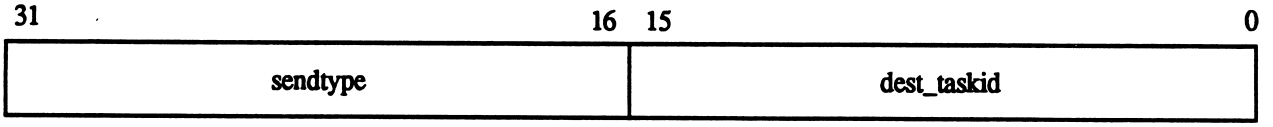
2. Use the following VADS command:

```
a.path -a /usr/ipsc/ada/lib
```

3. Compile within an Ada library, as described in Chapter 11, "Getting Started With VADS," in the *iPSC®/2 Ada Program Development Guide*.

Tasking Message Types

To ensure that messages are delivered to the correct task, the tasking version of the message-passing libraries encodes the user-defined task id into the type parameter as follows:



The libraries create this structure from the *sendtype* and *dest_taskid* parameters in the send calls, and from the type specified in receive and probe requests. Because the actual type seen by the system is the user-specified type shifted left by 16 bits, plus the user-specified task-id in the lower 16 bits, types in tasking calls are more restricted than their nontasking counterparts. Specifically:

- The type specified in tasking message-passing calls must be in the range 0 to 15,257 or the range 16,384 to 30,516. Types in the range 15,258 to 16,383 and greater than 30,516 are reserved for system use. (These types are translated into the ranges 1,000,000,000 to 1,073,741,823 and 2,000,000,000 and up, once the shifting is done).

- Negative type masks are not allowed in receive and probe operations. Because the task-id is in the lower 16 bits of the field, negative types would not produce the expected result.

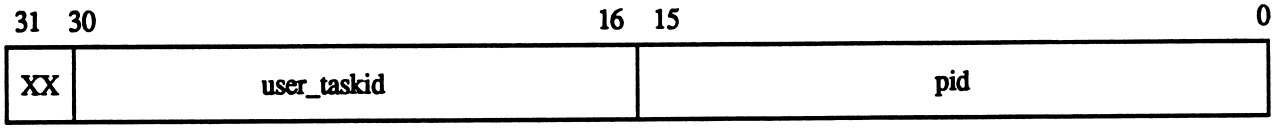
You must ensure that the requested message type matches the type that is actually sent after the translation takes place. This translation is also required when communicating between an Ada program using the tasking libraries and a C or Fortran program on a different node.

Task IDs and Process IDs

In the tasking message-passing libraries, the user-defined process-id consists of two parts:

user_taskid	Set by **set_taskid()**; returned by **infotaskid()**
pid	Set by **setpid()** (for a host process) or **load()** (for a node process); returned by **infopid()**

This encoding is shown as follows (bit 31 is reserved):



Because bit 31 is reserved, the *user_taskid* must be in the range 0 to 32,767. An unassigned *user_taskid* defaults to 0.

The *pid* must be in the range 0 to 65,535.

COMMON LIBRARY CALLS 2

This chapter documents the iPSC/2 routines that are *common* to both the nontasking and the tasking versions of the Ada system's host and node interface libraries.

Chapter 3 documents the iPSC/2 routines that are specific to the *nontasking* version of the Ada system's host and node interface libraries.

Chapter 4 documents the iPSC/2 routines that are specific to the *tasking* version of the Ada system's host and node interface libraries.

The routines are listed alphabetically and are documented as follows:

- Name and brief description
- Synopsis
- Return values (where applicable)
- Example
- Environment
- Languages
- Detailed description (including input parameters)
- iPSC exceptions raised
- List of related routines

ATTACHCUBE ATTACHCUBE

Make a different cube the current cube. This call is for use by host programs only; it is not available to node programs.

Synopsis

procedure **attachcube**(*cubename* : in string);

Return Value

None

Example

The following example allocates two cubes with **getcube()**. The last cube allocated becomes the current cube. To use the other cube, call **attachcube()**.

```
cube1 : string(1..NAMELEN);
cube2 : string(1..NAMELEN);
    •
cube1 := "alpha";
cube2 := "beta";
    •
getcube(cube1, "32", "", 0);
getcube(cube2, "8", "trainon", 1);          (Current cube is "beta")
    •
attachcube (cube1);                         (Current cube is "alpha")
    •
attachcube (cube2);                         (Current cube is "beta")
    •
```

ATTACHCUBE *(cont.)*

ATTACHCUBE *(cont.)*

Environment

Host

Languages

Ada, C, Fortran

Description of Parameters

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

Discussion

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

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

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

ATTACHCUBE *(cont.)*

ATTACHCUBE *(cont.)*

iPSC Exceptions Raised

`Cubename_does_not_exist`

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

`Commser_not_responding`

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

`Lifeline_not_responding`

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

See Also

cubeinfo(), getcube(), relcube()

CUBEINFO CUBEINFO

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

Synopsis

```
function cubeinfo(        ct : ct_ptr_type;
                  numslots : integer;
                    global : integer) return integer;
```

Return Value

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

Example

The following example uses **cubeinfo()** to find a specific cube ("alpha"). In this example, *numslots* is set to `10`, the maximum number of cubes allowed in a system. Then, to get information about all allocated cubes on the SRM, the example calls **cubeinfo()** with the *global* parameter set to `2`. The variable *numcubes* contains the number of cubes found and the table *cubetable* (defined in the package *cube*) contains the information about each cube. The example then searches *cubetable* for the cube named "alpha").

```
    •
ct_ptr   : ct_ptr_type;
numslots : integer := 10;
numcubes : integer;
    •
ct_ptr   := new cubetable_type;
numcubes := cubeinfo(ct_ptr, numslots, 2);
for i in 0..numcubes-1 loop
    if ct_ptr(i).cubename = "alpha" then
        •
```

CUBEINFO *(cont.)*

CUBEINFO *(cont.)*

Environment

Host

Languages

Ada, C, Fortran

Description of Parameters

ct — A pointer to the buffer table where cube information is to be returned.

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

global — The information to return:

0	Information about the currently attached cube.
1	Information about all cubes that you own that were allocated from the current host.
2	Information about all the cubes on the system from which the command was executed.

Discussion

Use **cubeinfo()** to obtain information about allocated cubes. The return value is the number of cubes with an array of information about each of these cubes. The parameter *numslots* limits the number of cubes for which information will be placed in an element. If the number of selected cubes is less than *numslots*, the remaining elements are set to zeros. If no cubes are found, **cubeinfo()** returns a `0`, and the buffer is cleared.

CUBEINFO *(cont.)*

CUBEINFO *(cont.)*

iPSC Exceptions Raised

`Commser_not_responding`

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

`Lifeline_not_responding`

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

`Global_value_invalid`

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

See Also

attachcube(), getcube(), relcube()

GETCUBE GETCUBE

Allocate a cube. This call is for use by host programs; it is not available for node programs.

Synopsis

```
procedure getcube(    cubename : in string;
                      cubetype : in string;
                      hostname : in string;
                          keep : in integer);
```

Return Value

None

Example

The following example gets a cube and names it "alpha." This cube is to be made up of 16 nodes. Using a null string for the *hostname*, causes **getcube()** to assume the System Resource Manager. A *keep* value of 0 releases the cube and terminates the node processes when the calling process exits or is killed.

```
getcube("alpha", "16", "", 0);
```

Environment

Host

Languages

Ada, C, Fortran

GETCUBE *(cont.)*

GETCUBE *(cont.)*

Description of Parameters

cubename The name of the cube that you want to allocate. Valid names include any ASCII string, 15 characters or less. Names longer than 15 characters will be truncated to 15 characters. If the name is a null string, "defaultname" is used. If you specify a name that you (or anyone else using your login name) is already using, it is an error. Use the **cubeinfo()** routine or the **cubeinfo** command to find the names of allocated cubes.

cubetype The size and type of the cube you want to allocate. The format of *cubetype* is size followed immediately by type in a contiguous string (no spaces). If you omit *cubetype*, you get the largest available cube of any type.

The number of nodes allocated is always a power of two. If the size you specify is not a power of two, it is rounded up. You can specify *size* in either of two ways:

n	Where *n* is the number of nodes (for example, 8)
d*n*	Where **d** indicates dimension and *n* is the size of the dimension (for example, d3 specifies a cube with eight nodes).

The *type* is optional. If you omit it, you get a cube of any type. The choices for *type* are:

f	Nodes with 387 coprocessor 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.
n*s*	Gets a contiguous set of nodes starting with slot *s*.

GETCUBE *(cont.)*

GETCUBE *(cont.)*

hostname The name of the host connected to the cube that you want to allocate. If you are on a System Resource Manager, the name defaults to your System Resource Manager's name. Other *hostnames* are ignored.

keep One of two settings that let you choose to keep the cube either until the process is terminated or until you release it. The possible settings are:

0	Releases the cube and terminates all node processes when the process that invoked **getcube()** exits or is killed.
1	Does not release the cube until the owner releases the cube with **relcube()** or the **relcube** command.

Discussion

Use **getcube()** to allocate a new cube, assigning a name, type, host, and lifetime to the cube. It also starts a file server that will handle I/O for the allocated cube. You assign names to cubes when you invoke **getcube()** so that the system will know which cube a program will be connected to. All cubes allocated by a single user must have unique names.

The *cubetype* parameter specifies the size and, if desired, the type of cube you want allocated. Cubes are always allocated such that the number of nodes is a power of 2. Thus, in addition to the higher powers of two, you may allocate a cube composed of 0 or 1 node. You may also specify the minimum memory required in addition to the type of node. The nodes allocated are guaranteed to have at least the required amount of memory, but may have more.

For example, `2` specifies a 2-node cube containing any type of node, `16f` specifies a 16-node cube containing 387 coprocessors, and `d3m4` specifies an 8-node cube where each node has at least 4M bytes of memory per node. To get a cube composed of different types of nodes, specify the different sizes and types, separated by a slash (/) in one string. For example, `4f/4m8` gets four 387 nodes and four nodes with at least 8M bytes of memory each.

You are automatically attached to the last cube allocated. Subsequent message-passing and loader calls made by the program which called **getcube()** will apply to the newly allocated cube. Use **attachcube()** when you want to access a different cube which has been previously allocated.

GETCUBE *(cont.)* GETCUBE *(cont.)*

All the cubes that you own are automatically released when you log out of the SRM unless you invoke your program with **nohup**. To invoke your program with **nohup**, enter the following:

```
nohup progname &
```

If any of the parameters are not specified, default values are used. With no arguments, **getcube()** reserves a cube for you named "defaultname." The number of nodes reserved for this cube is the largest integral power of 2 that is possible given the existing set of unallocated nodes.

If the cube cannot be allocated, an exception is raised. The **getcube** command returns the size of the cube that was actually allocated.

iPSC Exceptions Raised

`Internal_cube_usage_limit`

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

`Commser_not_responding`

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

`Lifeline_not_responding`

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

`Cubename_already_exists`

Use a different name.

`Cubetype_not_found`

Could not find a cube of the type requested.

See Also

attachcube(), **cubeinfo()**, **newserver()**, **numdim()**, **numnodes()**, **relcube()**

bootcube and **rebootcube** commands (in the *iPSC®/2 and iPSC®/860 Programmer's Reference Manual*).

GINV GINV

Returns the position of an element in the binary-reflected Gray code sequence.

Synopsis

function **ginv**(*j* : integer) return integer;

Return Value

An integer interpreted as the corresponding position of the Gray code value used as input.

Example

The following example returns the position of the Gray code value 6, and stores it in variable *x*:

```
x : integer;
    •
x := ginv(6);
```

(For this example, x will equal 4)

Environment

Host, Node

Languages

Ada, C, Fortran

Description of Parameters

j	A positive integer value.

GINV *(cont.)*

GINV *(cont.)*

Discussion

The **ginv()** function takes a Gray code element (positive integer) as input, and returns the position of the element in the binary-reflected Gray code sequence. This is the inverse of **gray()**. For the inverse function, obtaining the corresponding Gray code value with an integer value as input, use the **gray()** function. The following table shows the Gray code translation for the integer values from 0 through 7.

000	001	011	010	110	111	101	100	*(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)*

iPSC Exceptions Raised

None

See Also

gray()

GRAY GRAY

Returns the binary-reflected Gray code for an integer.

Synopsis

function **gray(** *j* : integer) return integer;

Return Value

An integer that is the binary-reflected Gray code for the input.

Example

The followin example returns the Gray code value that corresponds to a value of 4, and stores it in the variable *x*:

```
x : integer;
    •
x := gray(4);                    (In this example, the returned value is 6)
```

Environment

Host, Node

Languages

Ada, C, Fortran

Description of Parameters

j	A positive integer value.

GRAY *(cont.)*

GRAY *(cont.)*

Discussion

The Gray code is a useful tool in determining where a particular node is in a give application topology (ring, mesh, tree, or linear array) and in optimally assigning tasks to nodes. The **gray()** function takes a positive integer (*j*) as input, and returns the Gray code element corresponding to *j* in the binary-reflected Gray code sequence. For the inverse function, obtaining the position from a Gray code value, use the **ginv()** function. The following table shows the Gray code translation for values from 0 to 7.

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)*

iPSC Exceptions Raised

None

See Also

ginv()

GSENDX GSENDX

Send a vector to a list of nodes.

Synopsis

```
procedure gsendx(      type     : in integer;
                       x        : in system.address;
                       xlen     : in integer;
                       nodenums : in int_vector_type;
                       nlen     : in integer);
```

Return Value

None

Environment

Node

Languages

Ada, C, Fortran

Description of Parameters

type	Message type. Must be the same for all participating processes. There must be no other messages of this type in the system.
x	A pointer to the input vector to be sent. *x* can be of any type.
xlen	The number of bytes of data in vector *x*.
nodenums	A pointer to the list of node numbers to which data is to be sent.
nlen	The number of node numbers in the *nodenums* list.

GSENDX *(cont.)*

GSENDX *(cont.)*

Discussion

Use **gsendx()** to send a vector to a set of nodes specified by *nodenums*. All the nodes that are to receive the vector must call **crecv()** or **irecv()** to receive the message, using the type specified for **gsendx()**. All participating processes must have the same *pid* (and *taskid*).

iPSC Exceptions Raised

`Invalid_node`

Use **numnodes()** to determine cube size, and use **myhost()** to determine host node number.

`Invalid_buffer_pointer`

Specify a pointer that contains the address of a valid vector.

`Invalid_length`

Use a non-negative number for *xlen* or *nlen*.

`Buffer_length_exceeds_allocation`

Make sure the vector is large enough.

See Also

csend(), isend()

GSYNC GSYNC

Global synchronization operation.

Synopsis

procedure **gsync;**

Return Value

None

Environment

Node

Languages

Ada, C, Fortran

Discussion

Use **gsync()** to synchronize node processes (or tasks on different nodes). This is not a precise synchronization; when a node process executes **gsync()**, it waits until all other nodes in the cube have called **gsync()** before continuing to execute the process. All participating processes must have the same *pid* (and *taskid*).

IPSC Exceptions Raised

None

See Also

waitall(), waitone()

GxAND GxAND

Global AND operation for integer or logical vectors or scalars.

Synopsis

Integer (bitwise operation):

```
procedure giand(      x : in out int_vector_type;
                      n : in integer;
                   work : in int_vector type);
```

Logical (logical operation):

```
procedure gland(      x : in out bool_vector_type;
                      n : in integer;
                   work : in bool_vector_type);
```

Return Value

None

Environment

Node

Languages

Ada, C, Fortran

Description of Parameters

x	The input vector or scalar to be used in the operation. When the operation is complete, it contains the final result.
n	The number of elements in *x*.
work	An array that is used to receive the contributions from other nodes. The length of *work* must be at least *n*.

GxAND *(cont.)*

GxAND *(cont.)*

Discussion

Use **giand()** or **gland()** to calculate the bitwise or logical AND of each integer or logical component (depending on which routine you use) of *x* across all nodes. The result is returned in *x* to every node. Each node receives the results from all the other nodes in the cube in the *work* array. All participating processes must have the same *pid* and *taskid*.

This is a global operation; all nodes in a cube must execute this operation before any node can continue. When *x* is a vector, the resulting vector contains the accumulation of the corresponding elements of the input vectors from each node.

iPSC Exceptions Raised

`Received_message_too_long_for_buffer`

Make sure *n* is the same for all nodes.

`Received_message_too_short_for_buffer`

Make sure *n* is the same for all nodes.

`Invalid_buffer_pointer`

Specify an address for a valid vector.

`Invalid_length`

Use a non-negative number for *n*.

`Buffer_length_exceeds_allocation`

Make sure vector is large enough.

GxHIGH GxHIGH

Global MAX operation for double precision, integer, or real vectors or scalars.

Synopsis

Double precision:

```
procedure gdhigh(      x : in out float_vector_type;
                       n : in integer;
                    work : in float_vector_type);
```

Integer:

```
procedure gihigh(      x : in out int_vector_type;
                       n : in integer;
                    work : in int_vector_type);
```

Real:

```
procedure gshigh(      x : in out sfloat_vector_type;
                       n : in integer;
                    work : in sfloat_vector_type);
```

Return Value

None

Example

The following example uses **gihigh()** to find the largest among a list of numbers. This list is distributed among the nodes in straightforward domain decomposition; specifically, *list(i)* on node *p* represents the $(i + B*p)$th element of the global list, where B is the number of elements on each node. This example assumes that the size of the global list is $n*B = 2k$, where n is the number of nodes, and k is some positive integer.

GxHIGH *(cont.)*

GxHIGH *(cont.)*

```
                                    (Returns the largest value among all values of the global list)
function findMax (list        : in int_list_type;
                  numElements : integer) is
    return int_vector_type;
        •
    maxElement, temp : int_vector_type;
    index            : integer;
        •
    index := 0;
    for k in 1..numElements loop          (Find largest value for this node)
        if list(k) > list(index) then
            index := k;
        end if;
    end loop;
    maxElement := list(index);
    gihigh(maxElement, 1, temp);            (Now check across all nodes)
    return (maxElement);
end findMax;
```

Environment

Node

Languages

Ada, C, Fortran

Description of Parameters

x	The input vector or scalar to be used in the operation. When the operation is complete, it contains the final result.
n	The number of elements in *x*.
work	The array that is used to receive the contributions from other nodes. The length of *work* must be at least *n*.

GxHIGH *(cont.)*

GxHIGH *(cont.)*

Discussion

Use **gdhigh()**, **gihigh()** or **gshigh()** to calculate the maximum value of the double precision, integer, or real component (depending on which routine you use) of *x* across all nodes. The result is returned in *x* to every node. The *work* array is used to receive messages from other nodes. All participating nodes must have the same *pid* and *taskid*.

This is a global operation; all nodes in a cube must execute this operation before any node can continue. When *x* is a vector, each element of the resulting vector represents the accumulation of the corresponding index of the input vectors from each node.

IPSC Exceptions Raised

`Received_message_too_long_for_buffer`

Make sure *n* is the same for all nodes.

`Received_message_too_short_for_buffer`

Make sure *n* is the same for all nodes.

`Invalid_buffer_pointer`

Specify an address of a valid vector.

`Invalid_length`

Use a non-negative number for *n*.

`Buffer_length_exceeds_allocation`

Make sure the value of *n* does not exceed the vector size.

GxLOW GxLOW

Global MIN operation for double precision, integer, or real vectors or scalars.

Synopsis

Double precision:

```
procedure gdlow(        x : in out float_vector_type;
                        n : in integer;
                     work : in float_vector_type);
```

Integer:

```
procedure gilow(        x : in out int_vector_type;
                        n : in integer;
                     work : in int_vector_type);
```

Real:

```
procedure gslow(        x : in out sfloat_vector_type;
                        n : in integer;
                     work : in sfloat_vector_type);
```

Return Value

None

Example

The following example uses **gilow()** to find the smallest among a list of numbers. This list is distributed among the nodes in straightforward domain decomposition; specifically, *list(i)* on node *p* represents the $(i + B*p)$th element of the global list, where B is the number of elements on each node. This example assumes that the size of the global list is $n*B = 2k$ where n is the number of nodes and k is some positive integer.

GxLOW *(cont.)*

GxLOW *(cont.)*

```
                                    (Returns the smallest value among all values of the global list)
function findMin (list : in int_list_type;
                  numElements : integer) is
    return int_vector_type;
        •
    minElement, temp : int_vector_type;
    index            : integer;
        •
    index := 0;
    for k in 1..numElements loop        (Find smallest value for this process)
        if list(k) < list(index) then
            index := k;
        end if;
    end loop;
    minElement := list(index);
    gilow(minElement, 1, temp);                (Now check across all nodes)
    return (minElement);
end findMin;
```

Environment

Node

Languages

Ada, C, Fortran

Description of Parameters

x	The input vector or scalar to be used in the operation. When the operation is complete, it contains the final result.
n	The number of elements in *x*.
work	The array that is used to receive the contributions from other nodes. The length of *work* must be at least *n*.

GxLOW *(cont.)*

GxLOW *(cont.)*

Discussion

Use **gdlow()**, **gilow()**, or **gslow()** to calculate the minimum value of the double precision, integer, or real component (depending on which routine you use) of *x* across all nodes. The result is returned in *x* to every node. The *work* array is used to receive messages from other nodes. All participating processes must have the same *pid* and *taskid*.

This is a global operation; all nodes in a cube must execute this operation before any node can continue. When *x* is a vector, the resulting vector contains the accumulation of the corresponding elements of the input vectors from each node.

iPSC Exceptions Raised

`Received_message_too_long_for_buffer`

Make sure *n* is the same for all nodes.

`Received_message_too_short_for_buffer`

Make sure *n* is the same for all nodes.

`Invalid_buffer_pointer`

Specify an address of a valid vector.

`Invalid_length`

Use a non-negative number for *n*.

`Buffer_length_exceeds_allocation`

Make sure the value of *n* does not exceed the vector size.

GxOR GxOR

Global OR operation for integer or logical vectors or scalars.

Synopsis

Integer (bitwise operation):

```
procedure gior(      x : in out int_vector_type;
                     n : in integer;
                  work : in int_vector type);
```

Logical (logical operation):

```
procedure glor(      x : in out bool_vector_type;
                     n : in integer;
                  work : in bool_vector_type);
```

Return Value

None

Environment

Node

Languages

Ada, C, Fortran

Description of Parameters

x	The input vector or scalar to be used in the operation. When the operation is complete, it contains the final result.
n	The number of elements in *x*.
work	An array that is used to receive the contributions from other nodes. The length of *work* must be at least *n*.

GxOR *(cont.)*

GxOR *(cont.)*

Discussion

Use **gior()** or **glor()** to calculate the bitwise or logical OR of each integer or logical component (depending on which routine you use) of *x* across all nodes. The result is returned in *x* to every node. Each node receives the results from all the other nodes in the cube in the *work* array. All participating processes must have the same *pid* and *taskid*.

This is a global operation; all nodes in a cube must execute this operation before any node can continue. When *x* is a vector, the resulting vector contains the accumulation of the corresponding elements of the input vectors from each node.

IPSC Exceptions Raised

`Received_message_too_long_for_buffer`

Make sure *n* is the same for all nodes.

`Received_message_too_short_for_buffer`

Make sure *n* is the same for all nodes.

`Invalid_buffer_pointer`

Specify an address for a valid vector.

`Invalid_length`

Use a non-negative number for *n*.

`Buffer_length_exceeds_allocation`

Make sure vector is large enough.

GxPROD GxPROD

Global multiplication operation for double precision, integer, or real vectors or scalars.

Synopsis

Double precision:

```
procedure gdprod(      x : in out float_vector_type;
                       n : in integer;
                    work : in float_vector_type);
```

Integer:

```
procedure giprod(      x : in out int_vector_type;
                       n : in integer;
                    work : in int_vector_type);
```

Real:

```
procedure gsprod(      x : in out sfloat_vector_type;
                       n : in integer;
                    work : in sfloat_vector_type);
```

Return Value

None

Environment

Node

Languages

Ada, C, Fortran

GxPROD *(cont.)*

GxPROD *(cont.)*

Description of Parameters

x	The input vector or scalar to be used in the operation. When the operation is complete, it contains the final result.
n	The number of elements in *x*.
work	The array that is used to receive the contributions from other nodes. The length of *work* must be at least *n*.

Discussion

Use **gdprod()**, **giprod()**, or **gsprod()** to calculate the product value of the double precision, integer, or real component (depending on which routine you use) of *x* across all nodes. The result is returned in *x* to every node. The *work* array is used to receive messages from other nodes. All participating processes must have the same *pid* and *taskid*.

This is a global operation; all nodes in a cube must execute this operation before any node can continue. When *x* is a vector, the resulting vector contains the accumulation of the corresponding elements of the input vectors from each node.

IPSC Exceptions Raised

`Received_message_too_long_for_buffer`

Make sure *n* is the same for all nodes.

`Received_message_too_short_for_buffer`

Make sure *n* is the same for all nodes.

`Invalid_buffer_pointer`

Specify an address of a valid vector.

`Invalid_length`

Use a non-negative number for *n*.

`Buffer_length_exceeds_allocation`

Make sure the value of *n* does not exceed the vector size.

GxSUM GxSUM

Global sum operation for double precision, integer, or real vectors or scalars.

Synopsis

Double precision:

```
procedure gdsum(      x : in out float_vector_type;
                      n : in integer;
                   work : in float_vector_type);
```

Integer:

```
procedure gisum(      x : in out int_vector_type;
                      n : in integer;
                   work : in int_vector_type);
```

Real:

```
procedure gssum(      x : in out sfloat_vector_type;
                      n : in integer;
                   work : in sfloat_vector_type);
```

Return Value

None

Example

The following example uses **gdsum()** to obtain the sum of the dot products of a vector of length *n* that is distributed among the nodes of the cube. In this example, *x(m)* is a piece of the vector on a given node. *m* need not have the same value on all nodes.

GxSUM *(cont.)*

GxSUM *(cont.)*

```
    •
x, dummy : float_vector_type;
m        : integer;
dot      : float;
    •
for i in 1..m loop                    (Assume m and x have been initialized)
    dot := dot + x(i) * x(i);
end loop;
gdsum(x, 1, dummy);
```

Environment

Node

Languages

Ada, C, Fortran

Description of Parameters

x	The input vector or scalar to be used in the operation. When the operation is complete, it contains the final result.
n	The number of elements in *x*.
work	The array that is used to receive the contributions from other nodes. The length of *work* must be at least *n*.

Discussion

Use **gdsum()**, **gisum()**, or **gssum()** to calculate the sum of the double precision, integer, or real component (depending on which routine you use) of *x* across all nodes. The result is returned in *x* to every node. The *work* array is used to receive messages from other nodes. All participating processes must have the same *pid* and *taskid*.

This is a global operation; all nodes in a cube must execute this operation before any node can continue. When *x* is a vector, the resulting vector contains the accumulation of the corresponding elements of the input vectors from each node.

GxSUM *(cont.)*

GxSUM *(cont.)*

iPSC Exceptions Raised

`Received_message_too_long_for_buffer`

Make sure *n* is the same for all nodes.

`Received_message_too_short_for_buffer`

Make sure *n* is the same for all nodes.

`Invalid_buffer_pointer`

Specify an address of a valid vector.

`Invalid_length`

Use a non-negative number for *n*.

`Buffer_length_exceeds_allocation`

Make sure the value of *n* does not exceed the vector size.

GxXOR GxXOR

Global exclusive OR operation for integer or logical vectors or scalars.

Synopsis

Integer (bitwise operation):

```
procedure gixor(      x    : in out int_vector_type;
                      n    : in integer;
                      work : in int_vector type);
```

Logical (logical operation):

```
procedure glxor(      x    : in out bool_vector_type;
                      n    : in integer;
                      work : in bool_vector_type);
```

Return Value

None

Environment

Node

Languages

Ada, C, Fortran

Description of Parameters

x	The input vector or scalar to be used in the operation. When the operation is complete, it contains the final result.
n	The number of elements in *x*.
work	An array that is used to receive the contributions from other nodes. The length of *work* must be at least *n*.

GxXOR *(cont.)*

GxXOR *(cont.)*

Discussion

Use **gixor()** or **glxor()** to calculate the bitwise or logical exclusive OR of each integer or logical component (depending on which routine you use) of *x* across all nodes. The result is returned in *x* to every node. Each node receives the results from all the other nodes in the cube in the *work* array. All participating processes must have the same *pid* and *taskid*.

This is a global operation; all nodes in a cube must execute this operation before any node can continue. When *x* is a vector, the resulting vector contains the accumulation of the corresponding elements of the input vectors from each node.

IPSC Exceptions Raised

`Received_message_too_long_for_buffer`

Make sure *n* is the same for all nodes.

`Received_message_too_short_for_buffer`

Make sure *n* is the same for all nodes.

`Invalid_buffer_pointer`

Specify an address for a valid vector.

`Invalid_length`

Use a non-negative number for *n*.

`Buffer_length_exceeds_allocation`

Make sure vector is large enough.

KILLCUBE KILLCUBE

Terminate and clear out NX process(es).

Synopsis

```
procedure killcube(    node : in integer;
                       pid  : in integer);
```

Return Value

None

Environment

Host, Node

Languages

Ada, C, Fortran

Description of Parameters

node — An integer that specifies the node on which you want to terminate processes and flush messages (`-1` specifies all nodes). Nodes in a cube are numbered consecutively from 0. A host id is invalid.

pid — An integer value that specifies the process id that you want to terminate and flush. The *pid* must match a *pid* given for a previous **load()**. Calling **killcube()** with a *pid* assigned to an Ada process terminates all tasks associated with the process.

KILLCUBE *(cont.)*

KILLCUBE *(cont.)*

Discussion

Use **killcube()** to kill the specified node process(es) and flush messages related to those processes. The **killcube()** call is included for convenience; it is equivalent to the following sequence of calls:

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

It is not an error to use **killcube()** to kill a nonexistent process.

iPSC Exceptions Raised

`Invalid_node`

Use **numnodes()** to determine cube size, and use **myhost()** to determine host number.

`Invalid_pid`

Use a non-negative number.

See Also

load(), **killproc()**, **flushmsg()**, **waitall()**

KILLPROC KILLPROC

Terminate a process.

Synopsis

```
procedure killproc(     node : in integer;
                        pid  : in integer);
```

Return Value

None

Example

The following examples show how to use the **killproc()** call.

```
     •
killproc(15, 3);                    (Kill process 3 on node 15)
     •
killproc(-1, 10);                   (Kill process 10 on all nodes)
     •
```

Environment

Host, Node

Languages

Ada, C, Fortran

KILLPROC *(cont.)*

KILLPROC *(cont.)*

Description of Parameters

node — An integer value that specifies the node(s) on which you want to terminate a process. The host node number is invalid.

pid — An integer value that specifies the id of the process that you want to terminate. The *pid* must match a *pid* for a previous **load()**. Valid *pids* include any integer value, but negative numbers are reserved for system processes. When an Ada process is terminated, all tasks associated with the program are also terminated.

Discussion

Use **killproc()** to kill the node process specified by the *pid* and *node*. It has no effect on messages already sent from or in transit to the process. It causes the selected process to terminate, but does not wait for a process to complete, or flush messages related to that process.

If *node* is a positive integer, the specified process on that node is killed.

If *node* is a negative number up to a certain value, it causes a global kill. If *node* is `-1`, the specified process on all nodes in the cube is killed. Other negative numbers kill the process(es) in a set of the nodes surrounding the sender. The dimension of this cube is calculated as:

d - **nodedim()** - 1

where *d* is the dimension of the cube, and *d* is less than or equal to **nodedim()**.

iPSC Exceptions Raised

`Invalid_pid`

Use a non-negative number.

`Invalid_node`

Use **numnodes()** to determine cube size.

See Also

load(), **flushmsg()**, **killcube()**, **waitone()**, **waitall()**

KILLSYSLOG KILLSYSLOG

Terminate *syslog* process. This call is for use by host programs only; it is not available to node programs.

Synopsis

procedure **killsyslog;**

Return Value

None

Environment

Host

Languages

Ada, C, Fortran

Description of Parameters

None

Discussion

Use **killsyslog()** to kill the *syslog* process and reset the standard output to the terminal */dev/tty*. It can be used to kill the *syslog* process started by the **setsyslog()** routine; it cannot be used to kill the *syslog* process started by the **syslog** command.

The purpose of the *syslog* process is to make sure that output from host programs goes through the same file server used by the node programs. To change file servers to the current working environment, the *syslog* process must be killed and restarted by invoking **setsyslog()** to make sure the output goes to the correct file server.

KILLSYSLOG *(cont.)*

KILLSYSLOG *(cont.)*

iPSC Exceptions Raised

`No_active_syslog_exists`

To start the *syslog* process, use **setsyslog()**.

`There_is_no_attached_cube`

Use **getcube()** to get a cube.

See Also

setsyslog(), **newserver()**, **syslog** command (in the *iPSC®/2 and iPSC®/860 Programmer's Reference Manual*).

LED LED

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

Synopsis

procedure **led(** *lstate* : in integer);

Return Value

None

Example

The following example turns on the LED at the beginning of an operation, and turns off the LED when the operation completes.

```
      •
led(1);                                    (Turn LED on)
      •
      •                                    (Do operation here)
      •
led(0);                                    (Turn LED off)
```

Environment

Node

Languages

Ada, C, Fortran

LED *(cont.)* LED *(cont.)*

Description of Parameters

lstate An integer value specifying on or off state of the node board's green LED. The possible settings are:

`1`	Turns on the LED
`0`	Turns off the LED

Discussion

Use **led()** to turn on or off (depending on the value of *lstate*) the green LED. Behavior is not defined for values other than `0` or `1`.

iPSC Exceptions Raised

None

LOAD LOAD

Load a node process.

Synopsis

```
procedure load(     filename : in string;
                       node : in integer;
                        pid : in integer);
```

Return Value

None

Example

The following example loads the file named "nodefile" into all nodes in the current cube, and assigns it a process id of 0 on all nodes.

```
load ("nodefile", -1, 0);
```

Environment

Host, Node

Languages

Ada, C, Fortran

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

Description of Parameters

filename — A character string that specifies the pathname of the program file that you want to load into the node. Valid pathnames are ASCII character strings.

node — An integer value that specifies the node into which you want to load a process (`-1` specifies all nodes). Nodes in a cube are numbered consecutively from 0. A host node number is invalid. A process can use **mynode()** to obtain its node number after it begins execution.

pid — An integer value that specifies the process id assigned to the loaded node process.

Discussion

Use **load()** to load the file designated by *filename* onto the specified *node*(s), assign *pid* to the process(es), and begin execution. The host node id is invalid because this routine does not load processes onto the host. The *node* and *pid* specified in the load operation are used in send, receive, and kill operations.

Valid *pids* include any integer value, but negative numbers are reserved for system programs. Only one process per node is allowed. A process can use **mypid()** to obtain its process id after it begins execution.

iPSC Exceptions Raised

`There_is_no_attached_cube`

Use **getcube()** to allocate a cube, and then use **attachcube()** to change current cubes in your program.

`Invalid_pid`

A *pid* cannot be negative.

`Invalid_node`

A *node* number must be in range. Use **numnodes()** to determine cube size.

LOAD *(cont.)*

LOAD *(cont.)*

`Out_of_process_slots`

Use fewer processes.

`Pid_already_in_use`

Use a different *pid*.

`Not_enough_memory`

Reduce the size of the process.

`Invalid_loader_record`

Internal error. Try rebooting cube with **rebootcube** or **bootcube**.

`Invalid_loader_message`

Internal error. Try rebooting cube with **rebootcube** or **bootcube**.

`No_such_file_or_directory`

Use a valid *filename*.

`Invalid_object_file`

Specify a loadable file.

`There_are no_cubes_allocated`

Use **getcube()** to allocate a cube.

`No_pid_defined`

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

See Also

getcube(), killcube(), mynode(), mypid()

MCLOCK MCLOCK

Return elapsed time.

Synopsis

function **mclock** return integer;

Return Value

An integer value containing a relative time value.

Example

The following example uses **mclock()** to determine the elapsed time of an operation.

```
    •
start_time, end_time, elapsed_time : integer;
    •
start_time := mclock;                                  (Get starting time)
    •
    •                                                  (Operation you want to time)
    •
end_time := mclock;                                    (Get ending time)
elapsed_time := end_time - start_time;                 (Comute elapsed time)
```

Environment

Host, Node

Languages

Ada, C, Fortran

MCLOCK *(cont.)*

MCLOCK *(cont.)*

Description of Parameters

None

Discussion

Use **mclock()** as a mechanism to measure time intervals; it returns the value of a counter that reflects relative time in milliseconds. Programs that use **mclock()** should obtain an initial time value and interpret all other time values relative to this initial value.

When **mclock()** is called from a host program, the relative time (in milliseconds) it returns counts only the time that the UNIX system actually used to execute your host program and system operations related to your host program.

On the nodes, the value returned by **mclock()** represents the actual elapsed milliseconds since the node was initialized. The **bootcube** and **rebootcube** commands initialize the nodes.

It is recommended that you do not use **mclock()** to synchronize programs on different nodes because every node maintains its own clock value and values on different nodes may vary after initialization.

On both the host and the nodes, the **mclock()** value rolls over approximately every 25 days.

iPSC Exceptions Raised

None

MSGCANCEL MSGCANCEL

Cancel an asynchronous send or receive operation.

Synopsis

procedure **msgcancel(** *id* : in integer);

Return Value

None

Example

The following example uses **msgcancel()** (when an error occurs) to cancel the receipt of a message and clear its id.

```
    •
buf          : string(1..80);
msg_id, type : integer;
error        : boolean;
    •
msg_id := irecv(type, buf'address, 80);
    •
    •                          (Processing while waiting for receive to complete)
    •
if (error = TRUE) then                                    (If error condition,
    msgcancel(msg_id);                                     cancel message)
```

Environment

Host, Node

MSGCANCEL *(cont.)*

MSGCANCEL *(cont.)*

Languages

Ada, C, Fortran

Description of Parameters

id — The message id returned when you invoke an **isend()**, **irecv()**, or **isendrecv()** routine.

Discussion

Use **msgcancel()** to cancel an asynchronous send or receive operation. The *id* parameter, which specifies the operation to cancel, must correspond to the value returned by a previous **isend()**, **irecv()**, or **isendrecv()**. When **msgcancel()** returns, you do not know whether the send or receive operation is complete. However, you do know that the operation is no longer active, which means that:

- The buffer can be used in subsequent asynchronous send and receive operations.
- The message id can be used in subsequent asynchronous send and receive operations, but not in subsequent **msgdone()** or **msgwait()**.

iPSC Exceptions Raised

`Invalid_message_id`

Use message id returned by **irecv()**, **isend()**, or **isendrecv()**.

`No_pid_defined`

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

See Also

irecv(), **isend()**, **isendrecv()**

MSGDONE MSGDONE

Determine whether an asynchronous send or receive operation is complete.

Synopsis

function **msgdone**(*id* : in integer);

Return Value

1	Indicates that the message is complete and that the associated message buffer is available for reuse (in the case of a send operation), or that the buffer contains valid data (in the case of a receive operation). You cannot call **msgdone()** or **msgwait()** with the same id parameter after **msgdone()** returns a 1 because the send or receive operation is complete and the id is no longer valid.
0	Indicates that the send or receive operation is not yet complete. In this case, you can use the same id again (in **msgcancel()**, **msgdone()**, or **msgwait()** calls) until the send or receive operation completes.

Example

The following example uses **msgdone()** to determine whether an **isend()** buffer is available for reuse. If so, it modifies the message buffer. Otherwise, it does not since the message has not yet been copied out of the buffer.

```
buf     : string(1..100);
msg_id  : integer;
msgtype : integer;
done    : integer;
    •
msg_id := isend(msgtype, buf'address, buf'size / 8,
                1, mypid);
    •
done := msgdone(msg_id);
if done = 1 then
    •                                   (You may now reuse the message buffer)
else
    •                                   (Message buffer still needs to be protected)
end if;
```

MSGDONE *(cont.)*

MSGDONE *(cont.)*

Environment

Host, Node

Languages

Ada, C, Fortran

Description of Parameters

id	The message id returned when you invoke an asynchronous send or receive operation.

Discussion

Use **msgdone()** to determine whether an asynchronous message-passing operation (**isend()**, **irecv()**, or **isendrecv()**) is complete. When **msgdone()** returns a `1`, then the operation is complete. This means that the associated message buffer is available for reuse (in the case of a send operation), or the buffer contains valid data (in the case of a receive).

The id used by the asynchronous message-passing call is released when **msgdone()** returns a `1`. You must use **msgcancel()**, **msgdone()**, or **msgwait()** after each asynchronous message-passing call to ensure that the id is cleared. You cannot call **msgcancel()**, **msgdone()**, or **msgwait()** with the same id parameter after **msgdone()** returns a `1` because the send or receive operation is complete and the id is no longer valid. The id parameter specifies the operation on which you are checking status. The id must correspond to the value returned by a previous **irecv()**, **isend()**, or **isendrecv()** call.

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

MSGDONE *(cont.)*

MSGDONE *(cont.)*

IPSC Exceptions Raised

`Invalid_message_id`

Use message id returned by **irecv()**, **isend()**, or **isendrecv()**.

`No_pid_defined`

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

`Receive_message_too_long_for_buffer`

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

See Also

infocount(), **infonode()**, **infopid()**, **infotaskid()**, **infotype()**, **irecv()**, **isend()**, **isendrecv()**, **msgcancel()**, **msgwait()**

MSGWAIT MSGWAIT

Wait for completion of an asynchronous send or receive operation.

Synopsis

procedure **msgwait(** *id* : in integer);

Return Value

None

Example

The following example uses **msgwait()** to block execution of the task until the **irecv()** posted earlier has been completed. It also clears the id for use by subsequent sends and receives.

```
result   : integer;
msg_id   : integer;
msg_type : integer;
    •
msg_id := irecv(msg_type, result'address,
                result'size / 8);
    •
    •                                    (Perform tasks that do not use result)
    •
msgwait(msg_id);
    •
    •                                    (Now you can use the value of result)
    •
```

Environment

Host, Node

Languages

Ada, C, Fortran

MSGWAIT *(cont.)*

MSGWAIT *(cont.)*

Description of Parameters

id The message id returned when you invoke an asynchronous send or receive operation.

Discussion

Use **msgwait()** to block until the asynchronous operation (**isend()**, **irecv()**, or **isendrecv()**) is complete. When the operation is complete, the id number used by the asynchronous message-passing call is released. Since a limited number of message ids are available, you must use **msgcancel()**, **msgdone()**, or **msgwait()** after each asynchronous message-passing call to ensure that you do not run out of message ids, which would result in an error. The *id* parameter specifies the operation for which you are waiting. The *id* must correspond to a value returned by a previous **isend()**, **irecv()**, or **isendrecv()** call.

When the operation is complete, **msgwait()** returns and the associated message buffer is available for reuse. In the case of a send operation, the buffer can be modified without corrupting the message. In the case of a receive operation, the buffer contains valid data.

When **msgwait()** returns after waiting for an **irecv()** to complete, you can use any of the **info...()** calls to get more information about the message. You cannot use the *id* parameter again because the send or receive operation is complete.

iPSC Exceptions Raised

`Invalid_message_id`

Use the message id returned by **irecv()**, **isend()**, or **isendrecv()**.

`No_pid_defined`

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

`Received_message_too_long_for_buffer`

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

See Also

infocount(), **infonode()**, **infopid()**, **infotaskid()**, **infotype()**, **irecv()**, **isend()**, **isendrecv()**, **msgcancel()**, **msgdone()**

MYHOST MYHOST

Obtain the node number of the host machine.

Synopsis

function **myhost** return integer;

Return Value

The host node number is returned.

Example

The following example uses **myhost()** to obtain the host node number for a **csend()** from a node to the host.

```
msg     : string(1..100);
my_id   : integer;
host_id : integer;
    •
my_id   := mypid;
host_id := myhost;
    •
csend(10, msg'address, 100, host_id, my_id);     (Send message to host)
```

Environment

Host, Node

Languages

Ada, C, Fortran

MYHOST *(cont.)*

MYHOST *(cont.)*

Description of Parameters

None

Discussion

Use **myhost()** to return the host node number of the caller's host machine for use in send and receive operations. For host processes, **myhost()** returns the same id as **mynode()** (node number for a cube node). In a D4 cube, with nodes numbered 0 to 15, **myhost()** would return `16`.

iPSC Exceptions Raised

`No_pid_defined`

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

See Also

csend(), isend(), mynode()

MYNODE MYNODE

Obtain the node id of the calling program (or task).

Synopsis

function **mynode** return integer;

Return Value

The node id of the calling process (or task) that initiated the routine is returned.

Example

The following example uses **mynode()** to provide the node id to **killproc()**.

```
this_node : integer;
    •
this_node := mynode;                    (Obtain my logical node id)
killproc(this_node, -1);                (Kill all processes on this node)
```

Environment

Host, Node

Languages

Ada, C, Fortran

Description of parameters

None

MYNODE *(cont.)*

MYNODE *(cont.)*

Discussion

Use **mynode()** to return the node id of the calling program (or task). Nodes in a cube are numbered consecutively from 0. The host's id is the return value of **myhost()**. The id is used in send, kill, and wait operations.

iPSC Exceptions Raised

`No_pid_defined`

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

See Also

myhost(), mypid(), numnodes(), setpid()

MYPID MYPID

Obtain the NX/2 process id of the calling process.

Synopsis

function **mypid** return integer;

Return Value

The NX/2 process id of the calling process.

Example

The following example uses **mypid()** to return the process id to be used in a **csend()**.

```
buf   : string(1..100);
stype : integer;
    •
csend(stype, buf'address, buf'size / 8, -1, mypid);   (Sendmessage
                                                      to process with the same pid on all nodes)
```

Environment

Host, Node

Languages

Ada, C, Fortran

Description of Parameters

None

MYPID *(cont.)*

MYPID *(cont.)*

Discussion

Use **mypid()** to return the NX/2 process id of the Ada program. (This cube process id should not be confused with the UNIX process id).

IPSC Exceptions Raised

`No_pid_defined`

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

See Also

csend(), csendrecv(), flushmsg(), isend(), isendrecv(), setpid()

NEWSERVER NEWSERVER

Start a new file server for the specified cube.

Synopsis

procedure **newserver(** *cubename* : in string);

Return Value

None

Example

Before running a program named "host," enter the **getcube** command at the system prompt. With no redirection, the node output is the standard output (*stdio*) of the **getcube** command. When you run the host program with the **newserver()** call, the node output will go to the file indicated on the invocation line. Otherwise, the node output will go to the standard output, even if the "host" output is redirected to a different file.

At the system prompt:

```
% getcube
% host > file1          (Node output redirected to file1)
% host > file2          (Node output redirected to file2)
```

Within the "host" program:

```
newserver("defaultname");   (Now the node output will go to the file
                             that you named on the invocation line)
```

Environment

Host

Languages

Ada, C, Fortran

NEWSERVER *(cont.)*

NEWSERVER *(cont.)*

Description of Parameters

cubename The name of the cube (assigned with **getcube()**). Valid names include any ASCII string, 15 characters or less. Names longer than 15 characters are truncated to 15 characters. If the parameter is null or points to a null string, the currently attached cube is used. If you specify a cube that has not been allocated with **getcube()** or there is no cube currently attached, an exception results. Use the **cubeinfo** command or call to find the names of allocated cubes.

Discussion

When you allocate a cube with **getcube()**, a file server to handle I/O for the nodes is automatically created. If you want to redirect the output of your node process to a different file or device at program invocation, you must use **newserver()** in your program to allow the output to go to the desired place. The **newserver** command actually kills the original file server and starts a new one. When you issue **newserver()**, changes that you make to your environment are passed to the new file server.

For information on redirection, refer to the *Unix System V User's Guide.*

IPSC Exceptions Raised

`Cubename_does_not_exist`

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

`Commser_not_responding`

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

`Lifeline_not_responding`

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

See Also

cubeinfo(), getcube(), setsyslog(), newserver command (in the *iPSC®/2 and iPSC®/860 Programmer's Reference Manual*).

NODEDIM NODEDIM

Returns the dimension of the allocated cube.

Synopsis

function **nodedim** return integer;

Return Value

An integer from 0 to 7, representing the dimension of the cube.

Example

The following example uses **nodedim()** to return the dimension of the cube obtained by **getcube()**.

```
    •
cubedim : integer;
    •
getcube("alpha", "32", "", 0);
cubedim := nodedim;                    (Dimension of alpha is 5)
```

Environment

Host, Node

Languages

Ada, C, Fortran

Description of Parameters

None

NODEDIM *(cont.)*

NODEDIM *(cont.)*

Discussion

Use **nodedim()** to return the dimension of the allocated cube. For example:

- The dimension of a 64-node cube is 6 ($64 = 2^6$).
- If the number of nodes allocated is 8, **nodedim()** returns 3 ($8 = 2^3$).

Use **numnodes()** to return the number of allocated nodes.

iPSC Exceptions Raised

`No_pid_defined`

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

See Also

getcube(), **numnodes()**

NUMNODES NUMNODES

Obtain the number of nodes in the cube.

Synopsis

function **numnodes** return integer;

Return Value

An integer from 0 to 128, representing the number of nodes in a cube.

Example

The following example first gets two cubes ("alpha" and "beta"), and then uses **numnodes()** to find the highest-numbered node in the cube "beta," and the total number of nodes in the cube "alpha."

```
    •
last_node   : integer;
total_nodes : integer;
    •
getcube("alpha", "d3m4", "saxon", 1);
getcube("beta", "32", "viking", 1);
    •
last_node := numnodes - 1;              (The last node in cube "beta" is 31)
    •
attachcube("alpha");
    •
total_nodes := numnodes;        (The total number of nodes in cube "alpha" is 8)
```

Environment

Host, Node

Languages

Ada, C, Fortran

NUMNODES *(cont.)*

NUMNODES *(cont.)*

Description of Parameters

None

Discussion

Use **numnodes()** to return the number of nodes allocated to the cube. For example, if you allocate a cube of size 16 with **getcube()**, **numnodes()** will return 16 (2^4). Use **getcube()** to assign the number of nodes to a cube.

iPSC Exceptions Raised

`No_pid_defined`

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

See Also

getcube(), **nodedim()**

RELCUBE RELCUBE

Release a cube.

Synopsis

procedure **relcube(** *cubename* : in string);

Return Value

None

Example

The following example uses **getcube()** to get a cube, **killcube()** to terminate any remaining processes, and **relcube()** to release the cube.

```
    •
getcube("big", "32", NULL, 0);
    •
killcube(-1, -1);
relcube("big");
```

Environment

Host

Languages

Ada, C, Fortran

RELCUBE *(cont.)*

RELCUBE *(cont.)*

Description of Parameters

cubename 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 invalid. If the parameter is null or points to a null string, the currently attached cube is used. If you specify a cube that has not been allocated with **getcube()** or if there is no cube attached, an exception is raised. Use **cubeinfo()** to find the names of allocated cubes.

Discussion

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

iPSC Exceptions Raised

`Cubename_does_not_exist`

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

`Commser_not_responding`

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

`Lifeline_not_responding`

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

See Also

cubeinfo(), **getcube()**, **relcube** command (in the *iPSC®/2 and iPSC®/860 Programmer's Reference Manual*).

SETPID SETPID

Sets the process id of the host process.

Synopsis

procedure **setpid(** *pid* : in integer);

Return Value

None

Example

The following example sets the host *pid* for the cube to which you are attached.

```
    •
getcube("alpha", "16", "", 0);
getcube("beta", "16", "", 0);
    •
attachcube("alpha");
setpid(0);
    •
attachcube("beta");
setpid(99);
```

Environment

Host

Languages

Ada, C, Fortran

Description of Parameters

pid	The process id that you are assigning to a host process. Valid *pids* include any integer value, but negative numbers are reserved for system processes.

SETPID *(cont.)*

SETPID *(cont.)*

Discussion

Use **setpid()** to set the process id of the host process. To assign an id to a task, use **set_taskid()**. You must call **setpid()** before using any of the message-passing calls such as the send and receive operations.

You must allocate a cube before you call **setpid()**. If you change the current cube inside your user process by calling **getcube()** or **attachcube()**, you must call **setpid()**. If you attach to a cube to which you were previously attached, **setpid()** must be called to re-establish a process id. The *pid* used previously can be reused in **setpid()**. Use **mypid()** to obtain the *pid* of a host process after it has been set. The **setpid()** routine is not valid for a node process because the *pid* for a node process is set in **load()**.

iPSC Exceptions Raised

`Pid_already_in_use`

Select another *pid*.

`Pid_already_set`

A *pid* is already assigned for this process.

`There_is_no_attached_cube`

Use **cubeinfo()** to find which cubes you own.

`Commser_not_responding`

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

`Lifeline_not_responding`

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

See Also

getcube(), attachcube(), load(), mypid(), set_taskid()

SETSYSLOG SETSYSLOG

Start the *syslog* process.

Synopsis

procedure **setsyslog(** *outfile* : in integer);

Return Value

None

Example

The following example starts the *syslog* process (to redirect host process output to the standard output), and then loads a program.

```
     •
setsyslog(1);
load("prog1", -1, 1);
     •
```

Here's another example that redirects host program output to wherever node program output is going. This is useful for creating log files.

```
% getcube > logfile
```

In the host program:

```
     •                          (Normally, host program output goes to the terminal)
setsyslog(1);
     •                          (Now, host program output goes to the file "logfile")
```

Environment

Host

Languages

Ada, C, Fortran

SETSYSLOG *(cont.)*

SETSYSLOG *(cont.)*

Description of Parameters

outfile An integer value that indicates where the process output is to be sent:

1	Redirect output to standard output of the current node file server.
2	Redirect output to standard error of the current node file server.

Discussion

The *syslog* process is a system process that sends the output of a host process to either the standard output or standard error of the current node file server of the nodes. Use **setsyslog()** to start the *syslog* process and redirect the standard output of the calling program to the standard input of the *syslog* process. For more information on the *syslog* process, refer to the *iPSC®/2 and iPSC®/860 User's Guide*.

iPSC Exceptions Raised

`There_is_no_attached cube`

Use **cubeinfo()** to find the cubes you own.

`Active_syslog_exists`

A *syslog* process is already in progress.

`Cannot_start_syslog_process`

The system will not allow a *syslog* process.

See Also

killsyslog(), **syslog** command (in the *iPSC®/2 and iPSC®/860 Programmer's Reference Manual*).

WAITALL WAITALL

Wait for all specified processes to complete.

Synopsis

procedure **waitall(** *node* : in integer;
pid : in integer);

Return Value

None

Example

The following examples show how to use the **waitall()** call.

```
     •
waitall(10, 3);          (Wait for process 3 on node 10 to complete)
     •
waitall(15, -1);         (Wait for all processes on node 15 to complete)
     •
waitall(-1, -1);         (Wait for all processes on all nodes to complete)
     •
```

Environment

Host, Node

Languages

Ada, C, Fortran

WAITALL *(cont.)*

WAITALL *(cont.)*

Description of Parameters

node — An integer value that specifies the node on which the process you wish to wait for resides (`-1` specifies all nodes). Nodes in a cube are numbered consecutively from 0. A node number corresponding to the host is invalid.

pid — An integer value that specifies the node process id (`-1` specifies all processes). Valid *pids* can be any integer value, but negative numbers other than `-1` are reserved for system programs. The *pid* is assigned when you **load()** a process. It is not an error to wait for nonexistent processes; **waitall()** simply returns immediately.

Discussion

Use **waitall()** to block execution of the process until the selected processes complete on the selected node(s). Unlike **waitone()**, it will wait for all processes to complete if `-1` is specified. The procedure does not return until all specified processes on all specified nodes are completed. The completion codes are discarded.

IPSC Exceptions Raised

`Invalid_node`

Use **numnodes()** to determine cube size, or use `-1` to specify all nodes.

See Also

waitone(), load(), killcube(), killproc()

WAITONE WAITONE

Wait for a specified process to complete. If all nodes are specified, **waitone()** returns as soon as the process completes on any node.

Synopsis

```
procedure waitone(    node  : in integer;
                      pid   : in integer;
                      cnode : in integer_ptr;
                      cpid  : in integer_ptr;
                      ccode : in integer_ptr);
```

Return Value

None

Example

The following example waits for process 20 on any node to complete.

```
waitone(-1, 20, cnode, cpid, ccode);
```

Environment

Host, Node

Languages

Ada, C, Fortran

Description of Parameters

node An integer value that specifies the node on which the process you wish to wait for resides (`-1` specifies all nodes). Nodes are numbered consecutively from 0. A node number corresponding to the host is invalid.

WAITONE *(cont.)*

WAITONE *(cont.)*

pid An integer value that specifies the process id. Valid *pids* include any integer value, but negative numbers are reserved for system programs. An exception is raised if the *pid* is specified that does not correspond to an existing process. Processes are assigned a *pid* when you **load()** the program. An NX process can contain multiple Ada tasks.

cnode, cpid, ccode

The *node, pid,* and completion code of the completed process. The information returned in *ccode* depends on how the process terminated. If the process completes and exits normally, then the exit code (*ccode*) is undefined. However, if the process is killed or terminates with an exception, then *ccode* contains one of the codes listed below. When a process is terminated due to an exception, its completion code is the exception type minus 256. Some of these exceptions (NMI interrupt, Double fault, and Invalid TSS) are not possible unless there is a system error. The completion codes are:

-259	Out of buffers
-258	Invalid system call
-257	Process killed
-256	Integer divide by zero
-255	Single-step interrupt
-254	Nonmaskable interrupt
-253	Breakpoint interrupt
-252	Integer overflow exception
-251	Array bounds exception
-250	Invalid opcode
-249	Numeric processor not available
-248	Double fault
-247	Numeric processor overrun
-246	Invalid TSS fault
-245	Segment not present fault
-244	Stack overflow
-243	General protection fault
-242	Memory fault
-241	Vector processor error
-240	Floating point exception

WAITONE *(cont.)*

WAITONE *(cont.)*

Discussion

Use **waitone()** to wait for a process to complete on any of the nodes, and provide useful information on how the process terminated. If `-1` is used to select all nodes, **waitone()** returns as soon as the specified process on any node completes. This differs from **waitall()** in that **waitall()** does not return until all selected process on selected nodes complete.

iPSC Exceptions Raised

`No_pid_defined`

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

`Invalid_node`

Use **numnodes()** to determine cube size, and use **myhost()** to determine host node number.

See Also

waitall(), **waitcube** command (in the *iPSC®/2 and iPSC®/860 Programmer's Reference Manual*).

NONTASKING LIBRARY CALLS 3

This chapter documents the iPSC/2 routines that are specific to the nontasking version of the Ada system's host and node interface libraries.

The routines are listed alphabetically and are documented as follows:

- Name and brief description
- Synopsis
- Return values (where applicable)
- Example
- Environment
- Languages
- Detailed description (including input parameters)
- iPSC exceptions raised
- List of related routines

Table 3-1 lists all the nontasking calls and tells whether each is described in this chapter or in Chapter 2.

Table 3-1. Ada Nontasking Calls

Call	Chapter	Call	Chapter
attachcube()	2	gsync()	2
cprobe()	3	infocount()	3
crecv()	3	infonode()	3
csend()	3	infopid()	3
csendrecv()	3	infotype()	3
cubeinfo()	2	iprobe()	3
flick()	3	irecv()	3
flushmsg()	3	isend()	3
gdhigh()	2	isendrecv()	3
gdlow()	2	killcube()	2
gdprod()	2	killproc()	2
gdsum()	2	killsyslog()	2
getcube()	2	led()	2
giand()	2	load()	2
gihigh()	2	mclock()	2
gilow()	2	msgcancel()	2
ginv()	2	msgdone()	2
gior()	2	msgwait()	2
giprod()	2	myhost()	2
gisum()	2	mynode()	2
gixor()	2	mypid()	2
gland()	2	newserver()	2
glor()	2	nodedim()	2
glxor()	2	numnodes()	2
gray()	2	relcube()	2
gsendx()	2	setpid()	2
gshigh()	2	setsyslog()	2
gslow()	2	waitall()	2
gsprod()	2	waitone()	2
gssum()	2		

CPROBE CPROBE

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

Synopsis

procedure **cprobe**(*typesel* : in integer);

Return Value

None

Example

The following example uses **cprobe()** to block the calling process until the message of the given type is available to be received. Then, after using **infocount()** to ensure that the message is of the proper length, the example calls **crecv()** to receive the message.

```
    •
INIT_TYPE : constant integer := 10;
BUF_SIZE  : constant integer := 80;
    •
msg_id    : integer;
msglen    : integer;
msgbuf    : string(1..BUF_SIZE);
    •
cprobe(INIT_TYPE);                              (Wait for message of type 10)
msglen := infocount;
if (msglen <= BUF_SIZE) then                    (If message is right length,
    crecv(INIT_TYPE, msgbuf'address, BUF_SIZE);      receive message)
end if;
```

Environment

Host, Node

CPROBE *(cont.)*

CPROBE *(cont.)*

Languages

Ada, C, Fortran

Description of Parameters

typesel — A type selector, an integer value that specifies the type(s) of message that you are waiting to receive. You assign a type to a message when you initiate a send operation. The *typesel* parameter lets you select a specific message type or a set of message types based on a 32-bit mask.

The *typesel* parameter can be set as follows:

- If *typesel* is a non-negative integer, a specific message type will be recognized. All other messages will be ignored.
- If *typesel* is `-1`, the first non-force-type message to arrive for the process that initiated the probe operation will be recognized. After the first message has been received, you can use `-1` again to probe for the next message.
- If *typesel* is any negative number other than `-1`, a set of message types will be recognized. For information on building a *typesel* mask, refer to Appendix B in the *iPSC®/2 and iPSC®/860 Programmer's Reference Manual*.

Discussion

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

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

CPROBE *(cont.)*

CPROBE *(cont.)*

IPSC Exceptions Raised

`No_pid_defined`

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

`Too_many_requests`

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

`Invalid_type`

Cannot probe for force-type messages.

See Also

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

CRECV CRECV

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

Synopsis

```
procedure crecv(    typesel : in integer;
                    recvbuf : in system.address;
                    recvlen : in integer);
```

Return Value

None

Example

The following example uses **crecv()** to initiate the receive as soon as a message of any type is available. It will receive the message into the buffer named *buf*, of length 256.

```
    •
BUFLEN : constant integer := 256;
    •
buf : string(1..BUFLEN);
    •
crecv(-1, buf'address, BUFLEN);          (Receive the first message to arrive
                                          and store it in "buf")
```

Environment

Host, Node

Languages

Ada, C, Fortran

CRECV *(cont.)*

CRECV *(cont.)*

Description of Parameters

typesel — A type selector, an integer value that specifies the type(s) of message that you are waiting to receive. You assign a type to a message when you initiate a send operation. The *typesel* parameter lets you select a specific message type or a set of message types based on a 32-bit mask.

The *typesel* parameter 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 program that initiated the receive operation will be recognized. After the first message has been received, you can use `-1` again to receive the next message.
- If *typesel* is any negative number other than `-1`, a set of message types will be recognized. For information on building a *typesel* mask, refer to Appendix B in the *iPSC®/2 and iPSC®/860 Programmer's Reference Manual.*

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

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

Discussion

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

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

CRECV *(cont.)*

CRECV *(cont.)*

iPSC Exceptions Raised

`No_pid_defined`

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

`Invalid_length`

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

`Too_many_requests`

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

`Buffer_length_exceeds_allocation`

Make sure the length parameter does not exceed the buffer size.

`Invalid_buffer_pointer`

Specify the address of a valid data buffer.

`Received_message_too_long_for_buffer`

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

See Also

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

CSEND CSEND

Send a message. Blocks until execution is complete.

Synopsis

```
procedure csend(      type     : in integer;
                      sendbuf  : in system.address;
                      sendlen  : in integer;
                      destnode : in integer;
                      destpid  : in integer);
```

Return Value

None

Example

The following example defines a message ("Hello node 0") and uses **csend()** to send it to a process with the same id as the sender on node 0. Using the same message buffer, a new message is defined ("Hello host") and **csend()** is used again to send it to the host.

```
    •
msg : string(1..80);
len : integer;
    •
msg(1..12) := "Hello node 0";
len        := 12;
    •
csend(5, msg'address, len, 0, mypid);        (Send type 5 message
    •                                         to process on node 0)
msg(1..10) := "Hello host";
    •
csend(2, msg'address, 10, myhost, mypid);    (Send new type 2 message
                                              to process on host)
```

Environment

Host, Node

CSEND *(cont.)*

CSEND *(cont.)*

Languages

Ada, C, Fortran

Description of Parameters

type — An integer that specifies the type of the message that you are sending.

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

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

destnode — An integer that defines where the message is to be sent.

destpid — An integer that specifies the id of the process that is to receive the message. Valid *destpids* include any integer value. Negative numbers are reserved for system programs. The node *destpid* is assigned when you **load()** a process, and a host *destpid* is assigned with **setpid()**.

Discussion

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

Use a type with a value from 0 to 999,999,999. Types 1,073,741,824 to 1,999,999,999 are special force types that bypass the normal flow control mechanisms, do not match the `-1` wildcard, and are discarded if no receive is posted. Force types are intended to be used with **csendrecv()** and **isendrecv()**, but can be used by other message-passing calls. Do not use types in other ranges because they are used by the system and could produce unpredictable results.

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

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

If *destnode* is a negative number up to a certain value, it causes a global send. If *destnode* is `-1`, the message is sent to all nodes. Other negative numbers cause the message to be sent to a cube that is composed of a set of the nodes surrounding the sender. The dimension of this cube is calculated as:

d - **nodedim()** - 1

where *d* is the dimension of the cube, and *d* is less than or equal to **nodedim()**.

If a global send specifies its own process (**mypid()**), it does not receive its own message. However, if a process other than its own is specified, the destination process on the sending node also receives the message.

To initiate a send request when you want the calling program to continue during the send operation, use **isend()** or **isendrecv()**.

iPSC Exceptions Raised

`No_pid_defined`

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

`Invalid_type`

Use a non-negative number.

`Invalid_length`

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

`Invalid_node`

Use **numnodes()** to determine cube size, or use **myhost()** to determine host

`number.Buffer_length_exceeds_allocation`

Make sure the length parameter does not exceed the buffer size.

`Invalid_buffer_pointer`

Specify the address of a valid data buffer.

`Not_enough_memory`

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

CSEND *(cont.)*

CSEND *(cont.)*

See Also

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

CSENDRECV CSENDRECV

Send a message and receive a reply. Blocks the calling program until the reply is received.

Synopsis

```
function csendrecv(     sendtype : in integer;
                        sendbuf  : in system.address;
                        sendlen  : in integer;
                        destnode : in integer;
                        destpid  : in integer;
                        typesel  : in integer;
                        recvbuf  : in system.address;
                        recvlen  : in integer) return integer;
```

Return Value

The length of the received message.

Example

The following example shows how to use the **csendrecv()** call:

- Part 1 is a procedure that uses **csendrecv()** to make a remote procedure call and post a receive for the message containing the result.
- Part 2 is a portion of the server running on the receiving node that receives the message, preforms the operation, and sends back the result to the first procedure.

Part 1:

```
    •
x      : float;
result : float;
len    : integer;
    •
len := csendrecv(SIN_REQUEST_TYPE,        (Send value x to remote process,
                 x'address, 8, SERVER_NODE,                post a receive,
                 SERVER_PID, SIN_REPLY_TYPE,    and put result in buffer)
                 result'address, 8);
return result;
```

CSENDRECV *(cont.)*

CSENDRECV *(cont.)*

Part 2:

```
    •
x, result                      : float;
request_node, request_pid, i : integer;
    •
i := 1;
while (i = 1) loop
    crecv(SIN_REQUEST_TYPE, x'address, 8);
    request_node := infonode;
    request_pid  := infopid;
        •
        •                                          (Do processing here)
        •
    csend(SIN_REPLY_TYPE, result'address, 8,
          request_node, request_pid);                  (Return result)
end loop;
```

Environment

Host, Node

Languages

Ada, C, Fortran

Description of Parameters

sendtype — A non-negative integer that specifies the type of the message that you are sending.

sendbuf — A pointer to the buffer that contains the message to be sent.

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

CSENDRECV *(cont.)*

CSENDRECV *(cont.)*

destnode An integer that determines the node(s) to which the message is to be sent.

destpid An integer that specifies the id of the process that is to receive the message. Valid *pids* include any integer value. Negative numbers are reserved for system programs. A node *pid* is assigned when you **load()** a process, and a host *pid* is assigned with **setpid()**.

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

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

recvlen An integer value that specifies the size of the reply message buffer in bytes.

Discussion

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

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

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

- A message with a force type bypasses the normal flow control mechanisms, and is not delayed when normal messages have filled the message buffers.
- Force types do not match the `-1` wildcard type selector. This property can be used to guarantee that the message is received by the proper buffer, no matter what other messages are also received.
- If no receive is posted, as when the receiving process has been killed, a message with a force type is discarded. In general, bypassing the normal flow control causes no problem because **csendrecv()** guarantees that a receive is posted for the message.

CSENDRECV *(cont.)*

CSENDRECV *(cont.)*

Types 1,000,000,000 to 1,073,741,823, and greater than 2,000,000,000 are used by the system and should be avoided. Their use may produce unpredictable results.

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

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

If *destnode* is a negative number up to a certain value, it causes a global send. If *destnode* is `-1`, the message is sent to all nodes. Other negative numbers cause the message to be sent to a cube that is composed of a set of the nodes surrounding the sender. The dimension of this cube is calculated as:

d - **nodedim()** - 1

where *d* is the dimension of the cube, and *d* is less than or equal to **nodedim()**.

The *typesel* parameter describes the type(s) of the message(s) expected to be received. The *typesel* (type selector) lets you select a specific message type or a set of message types based on a 32-bit mask.

The *typesel* parameter can be set as follows:

- If *typesel* is a non-negative integer, a specific message type will be recognized. All other messages will be ignored. You can use a force type, as described previously.
- If *typesel* is `-1`, the first message to arrive for the process that initiated the receive operation will be recognized. After the first message has been received, you can use `-1` again to receive the next message.
- If *typesel* is any negative number other than `-1`, a set of message types will be recognized. For information on building a *typesel* mask, refer to Appendix B in the *iPSC®/2 and iPSC®/860 Programmer's Reference Manual.*

CSENDRECV *(cont.)*

CSENDRECV *(cont.)*

IPSC Exceptions Raised

`No_pid_defined`

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

`Invalid_type`

Use a non-negative number.

`Invalid_length`

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

`Invalid_node`

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

`Buffer_length_exceeds_allocation`

Make sure the length parameter does not exceed the buffer size.

`Invalid_buffer_pointer`

Specify the address of a valid data buffer.

`Not_enough_memory`

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

`Received_message_too_long_for_buffer`

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

See Also

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

FLICK FLICK

Relinquish node processor to another program.

Synopsis

procedure **flick;**

Return Value

None

Environment

Host, Node

Languages

Ada, C, Fortran

Discussion

The **flick()** call performs no operation on a host machine or in a nontasking program. It is included so that programs that call **flick()** will run in these environments.

iPSC Exceptions Raised

None

See Also

mclock()

FLUSHMSG FLUSHMSG

Flush specified messages from the system.

Synopsis

```
procedure flushmsg(   typesel : in integer;
                        node : in integer;
                         pid : in integer);
```

Return Value

None

Example

The following example sends two synchronous messages and one asynchronous message to process 0 on node 1. It then flushes all messages of all types waiting to be received for that process and node. Using **msgcancel()** to cancel the asynchronous message (*msg3*) before the **flushmsg()** call, assures that all messages are cancelled.

```
    •
node, pid, msg_id : integer;
    •
node := 1;
pid  := mypid;
    •
csend(10, msg1'address, msg1'size / 8, node, pid);
                                          (Blocking send to node 1)
csend(11, msg2'address, msg2'size / 8, node, pid);
                                          (Blocking send to node 1)
msg_id := isend(12, msg3'address, msg3'size / 8,
                node, pid);              (Asynchronous send to node 1)
    •
msgcancel(msg_id);                              (Cancel msg3 send)
flushmsg(-1, node, pid);                  (Now msg3 will be flushed)
```

FLUSHMSG *(cont.)*

FLUSHMSG *(cont.)*

Environment

Host, Node

Languages

Ada, C, Fortran

Description of Parameters

typesel — An integer value that specifies the type of message that you want to flush. The type depends on the type specified in the send. You can select a specific message type or a set of message types that are based on a 32-bit mask.

The *typesel* can be set as follows:

- If *typesel* is a non-negative integer, messages of the specified type will be flushed. All other messages will be ignored.
- If *typesel* is `-1`, all messages to the node will be flushed.
- If *typesel* is any negative number other than `-1`, a set of message types will be flushed. For information on building a *typesel* mask, refer to Appendix B in the *iPSC®/2 and iPSC®/860 Programmer's Reference Manual*.

node — A positive integer that specifies the receiving node (`-1` specifies all nodes and the host). Nodes in a cube are numbered consecutively from 0. If you want to flush messages only from the host, use **myhost()** to determine the host's node number.

pid — An integer that specifies the receiving process id (`-1` specifies all processes). Valid *pids* include any integer value, but negative numbers other than `-1` are reserved for system programs.

FLUSHMSG *(cont.)*

FLUSHMSG *(cont.)*

Discussion

Use **flushmsg()** to flush all message of the specified type from system buffers. All messages of the type(s) selected by *typesel* in the selected *node*(s) that have been sent to the selected *pid*(s) will be eliminated from the system.

Use of **flushmsg()** has no effect on node processes. It also has no effect on messages in transit to processes that do not have the selected *pid*, even if they were sent by a process that does have a matching *pid*. That is, **flushmsg()** flushes messages by the destination *pid*, not the source *pid*. If a particular node is selected, the only messages affected are those that are sent to the selected node, and which have arrived on the node, but have not been received.

To ensure that all messages are flushed, use **flushmsg()** with **msgcancel()**.

iPSC Exceptions Raised

`Invalid_node`

Use **numnodes()** to determine cube size, and use **myhost()** to determine host number.

See Also

msgcancel(), killcube()

INFO: INFOCOUNT, INFONODE, INFOPID, INFOTYPE

Return information about a pending or received message.

Synopsis

function **infocount** return integer;

function **infonode** return integer;

function **infopid** return integer;

function **infotype** return integer;

Return Value

The indicated information. The return value is defined only if the call is made immediately after a **crecv()**, **cprobe()**, or **msgwait()** completes, or after an **iprobe()** or **msgdone()** returns `1`, all of which indicate that a message is ready.

infocount Returns an integer value specifying the length of the message.

infonode Returns the node ID of the process that sent the message.

infopid Returns the process id of the process that sent the message.

infotype Returns the type of the message. Message type is specified in the send operation.

Example

The following example uses all of the **info...()** calls. First, it waits for a message of type 0 and then uses **infotype()** to verify the message type, and uses **infocount()** to determine the actual message length. Next, it uses that information to receive the waiting message. Finally, the example posts an **irecv()** for a message of type 10, and if it has been received, uses **infonode()** and **infopid()** to find the node and process id from which the message came.

INFO... *(cont.)*

INFO... *(cont.)*

```
    •
BIGNUM    : constant integer;
    •
type intbuf_type is array (1..BIGNUM) of integer;
    •
buf       : intbuf_type;
msgtype   : integer;
msglen    : integer;
msg_id    : integer;
from_node : integer;
from_pid  : integer;
status    : integer;
    •
cprobe(0);                                    (Wait for type 0 message to arrive)
msgtype := infotype;                                    (Get message type)
msglen  := infocount;                                   (Get message length)
crecv(msgtype, buf'address, msglen);          (Post receive for type 10 message)
    •
msg_id := irecv(10, buf'address, buf'size / 8);
    •
status := msgdone(msg_id);
if (status = 1) then
    from_node := infonode;
    from_pid  := infopid;
end if;
```

Environment

Host, Node

Languages

Ada, C, Fortran

Description of Parameters

None

INFO... *(cont.)*

INFO... *(cont.)*

Discussion

Use **infocount()**, **infonode()**, **infopid()**, and **infotype()** to return information about a pending or received message. They return information about the message detected when **cprobe()** completes or when **iprobe()** returns `1`. They return information about the message received when **crecv()** or **msgwait()** return or when **msgdone()** for a previous **irecv()** returns a `1`.

iPSC Exceptions Raised

None

See Also

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

IPROBE IPROBE

Determine whether a message of a selected type is pending.

Synopsis

function **iprobe**(*typesel* : in integer) return integer;

Return Value

0	A message of the selected type is not waiting to be received.
1	A message of the selected type is available to be received.

Example

The following example contains a continuous while-loop that uses **iprobe()** to test for the availability of two messages (type 1 and 2) and processes the first message to come in. If neither message is available, **flick()** allows another process on the same node to execute before looping again.

```
    •
i : integer := 0;
    •
while i = 0 loop
    if (iprobe(1) = 1) then
            •
            •                                    (Process type 1 message)
            •
        exit;
    elsif (iprobe(2) = 2) then
            •
            •                                    (Process type 2 message)
            •
        exit;
    else
        flick;
    end if;
end loop;
```

IPROBE *(cont.)*

IPROBE *(cont.)*

Environment

Host, Node

Languages

Ada, C, Fortran

Description of Parameters

typesel An integer value that specifies the type(s) of message for which you are waiting. You assign a type to a message when you initiate a send operation. The *typesel* (type selector) allows you to select a specific message type or a set of message types based on a 32-bit mask.
Typesel can be set as follows:

- If *typesel* is a non-negative integer, a specific message type will be recognized. All other messages will be ignored.
- If *typesel* is `-1`, the first non-force-type message to arrive for the process that initiated the probe operation will be recognized. After the first message has been received, you can use `-1` again to probe for the next message.
- If *typesel* is any negative number other than `-1`, a set of message types will be recognized. For information on building a *typesel* mask, refer to Appendix B in the *iPSC®/2 and iPSC®/860 Programmer's Reference Manual.*

Discussion

Use **iprobe()** to determine whether a message of a selected type is available for receipt. You specify the message type with the send operation, and **iprobe()** looks for a message whose type matches the type 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, it returns a `0`. When **iprobe()** returns a `1`, the **info...()** calls can be used to get more information about the message.

This is an asynchronous call; it does not block the process waiting for a message to arrive. Use **cprobe()** to block the process until a message of a selected type is available for receipt.

IPROBE *(cont.)*

IPROBE *(cont.)*

IPSC Exceptions Raised

`No_pid_defined`

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

`Too_many_requests`

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

`Invalid_type`

Cannot probe for force-type messages.

See Also

cprobe(), **infocount()**, **infonode()**, **infopid()**, **infotype()**

IRECV IRECV

Initiate the receipt of a message.

Synopsis

```
function irecv(     typesel : in integer;
                    recvbuf : in system.address;
                    recvlen : in integer) return integer;
```

Return Value

A non-negative message id to be used in **msgcancel()**, **msgdone()**, or **msgwait()**.

Example

The following example uses **irecv()** (to initiate receipt of a message) and **msgwait()** (to block execution of the process until the receive is complete).

The following example uses **irecv()** to initiate receipt of a message. and when the information in the message is required by the process, uses **msgwait()** to block execution of the process until the receive is complete, and to make the id used for the receive available for subsequent asynchronous message-passing calls.

```
    •
BUFLEN : constant integer := 256
    •
buf          : string(1..BUFLEN);
msg_id, type : integer;
    •
msg_id := irecv(type, buf'address, BUFLEN);     (Initiate message receipt)
    •
    •                  (Perform tasks that do not depend on the message being received)
    •
msgwait(msg_id);                                    (Wait to guarantee message receipt)
    •
    •                                  (Now use the contents of "buf" and the info calls)
```

IRECV *(cont.)*

IRECV *(cont.)*

Environment

Host, Node

Languages

Ada, C, Fortran

Description of Parameters

typesel — An integer value that specifies the type(s) of message for which you are waiting. You assign a type to a message when you initiate a send operation. The *typesel* (type selector) allows you to select a specific message type or a set of message types based on a 32-bit mask.

The *typesel* parameter can be set as follows:

- If *typesel* is a non-negative integer, a specific message type will be recognized. All other messages will be ignored.
- If *typesel* is `-1`, the first non-force-type message to arrive for the program that initiated the receive operation will be recognized. After the first message has been received, you can use `-1` again to receive the next message.
- If *typesel* is any negative number other than `-1`, a set of message types will be recognized. For information on building a *typesel* mask, refer to Appendix B in the *iPSC®/2 and iPSC®/860 Programmer's Reference Manual.*

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

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

IRECV *(cont.)*

IRECV *(cont.)*

Discussion

Use **irecv()** to post an asynchronous receive that initiates the receipt of a message from a process. It searches for a message whose type matches the type specified in the *typesel* parameter, and then immediately returns a message id that is used by **msgdone()** and **msgwait()** to determine whether the receive is complete. The **msgwait()** routine also clears the id assigned to the message as soon as the receive completes, as does **msgdone()** when it returns a `1`. You must use **msgwait()**, **msgdone()**, or **msgcancel()** to free this id for future sends and receives because the number of ids available is limited, and an error results if you use all the ids. When the message is received, it is stored in the buffer at address *recvbuf*.

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 the message.

The **irecv()** routine is asynchronous, allowing the calling process to continue even if the desired message is not yet available. It returns as soon as the receive request has been initiated. Use **crecv()** when you want the calling process to be blocked until the desired message is received.

iPSC Exceptions Raised

`No_pid_defined`

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

`Invalid_length`

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

`Too_many_requests`

Use **msgcancel()**, **msgdone()**, or **msgwait()** for outstanding asynchronous requests.

`Buffer_length_exceeds_allocation`

Make sure the length parameter does not exceed the buffer size.

`Received_message_too_long_for_buffer`

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

`Invalid_buffer_pointer`

Specify the address of a valid data buffer.

IRECV *(cont.)*

IRECV *(cont.)*

See Also

crecv(), csend(), infocount(), infonode(), infopid(), infotype(), isend()

ISEND ISEND

Initiate a message send.

Synopsis

```
function isend(      type     : in integer;
                     sendbuf  : in system.address;
                     msglen   : in integer;
                     destnode : in integer;
                     destpid  : in integer) return integer;
```

Return Value

A non-negative message id to be used in **msgcancel()**, **msgdone()**, or **msgwait()**.

Example

The following example uses **isend()** (to send a message of type 1 to process 10 on node 16 and **msgwait()** (to block execution of the process until the send is complete).

```
    •
msg                          : string(1..132);
type, len, node, pid, msg_id : integer;
    •
type := 1;
len  := msg'size / 8;
node := 16;
pid  := 10;
    •
msg_id := isend(type, msg, len, node, pid);      (Send type 1 message
    •                                             to process 10 on node 16)
msgwait(msg_id);                  (Now you can use the message buffer again)
```

ISEND *(cont.)*

ISEND *(cont.)*

Environment

Host, Node

Languages

Ada, C, Fortran

Description of Parameters

type — An integer that specifies the type of the message that you want to send. Type restrictions are described in the "Discussion".

sendbuf — The address of the buffer that contains the message you want to send. The buffer may be any legal data type. It is recommended that data types match in send and receive operations.

msglen — A positive integer that specifies the size (in bytes) of the message that you wish to send. Messages to or from a host process are limited to 256K bytes. There is no limit on message length between nodes. If *msglen* is greater than the maximum message length, an error message will be issued.

destnode — An integer that specifies the node or nodes to which the message is to be sent.

destpid — An integer that specifies the process id that is to receive the message. Valid *destpids* include any integer value. Negative numbers are reserved for system programs. A node *destpid* is assigned when you **load()** a process and a host *destpid* is assigned with **setpid()**.

Discussion

Use **isend()** to send a message to a node or host program. Completion of **isend()** does not imply that the message was received by the destination process, only that the send operation was initiated. When the message is sent, you can use the buffer for another message.

This call is asynchronous, returning as soon as the send request is initiated. Use **csend()** when you want to block the calling process until the message buffer is available for reuse.

ISEND *(cont.)*

ISEND *(cont.)*

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

The buffer can be of any type. You should not modify the buffer contents until **msgwait()** returns or **msgdone()** indicates that **isend()** is complete. Otherwise, corrupted data might be sent.

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

If *destnode* is a negative number up to a certain value, it causes a global send. If node is `-1`, the message is sent to all nodes. Other negative numbers cause the message to be sent to a cube composed of a set of the nodes surrounding the sender. The dimension of this cube is calculated as:

d - **nodedim()** - 1

where *d* is the dimension of the cube, and *d* is less than or equal to **nodedim()**.

If a global send specifies its own process (**mypid()**), it does not receive its own message. However, if a process other than its own is specified, the sending node also receives the message.

IPSC Exceptions Raised

`No_pid_defined`

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

`Invalid_type`

Use a non-negative number.

`Invalid_length`

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

`Invalid_node`

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

ISEND *(cont.)*

ISEND *(cont.)*

`Buffer_length_exceeds_allocation`

Make sure the length parameter does not exceed the buffer size.

`Invalid_buffer_pointer`

Specify the address of a valid data buffer.

`Not_enough_memory`

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

`Too_many_requests`

Use **msgcancel()**, **msgdone()**, or **msgwait()** for outstanding asynchronous requests.

See Also

cprobe(), **crecv()**, **csend()**, **csendrecv()**, **iprobe()**, **irecv()**, **isendrecv()**, **msgcancel()**, **msgdone()**, **msgwait()**

ISENDRECV ISENDRECV

Send a message and receive a reply.

Synopsis

```
function isendrecv(     sendtype : in integer;
                        sendbuf  : in system.address;
                        sendlen  : in integer;
                        destnode : in integer;
                        destpid  : in integer;
                        typesel  : in integer;
                        recvbuf  : in system.address;
                        recvlen  : in integer) return integer;
```

Return Value

A non-negative message id to be used in **msgcancel()**, **msgdone()**, or **msgwait()**.

Example

The following example shows how to use the **isendrecv()** call.

- Part 1 is a procedure that uses **isendrecv()** to make a remote procedure call and post a receive for the message containing the result. When the result is required, **msgwait()** is used to block the process until result is received.
- Part 2 is a portion of the server running on the receiving node that receives the message, performs the operation and returns the result to the first procedure.

ISENDRECV *(cont.)*

ISENDRECV *(cont.)*

Part 1:

```
        •
    x      : float;
    result : float;
    msg_id : integer;
        •
    msg_id := isendrecv(SIN_REQUEST_TYPE,      (Send value x to remote process,
                        x'address, 8, SERVER_NODE,      post a receive, and
                        SERVER_PID, SIN_REPLY_TYPE,     put result in buffer)
                        result'address, 8);
        •
    msgwait(msg_id);
    return result;
```

Part 2:

```
        •
    x, result                    : float;
    request_node, request_pid, i : integer;
        •
    i := 1;
    while (i = 1) loop
        crecv(SIN_REQUEST_TYPE, x'address, 8);
        request_node := infonode;
        request_pid  := infopid;
            •
            •                                              (Do processing here)
            •
        csend(SIN_REPLY_TYPE, result'address, 8,
              request_node, request_pid);                  (Return result)
    end loop;
```

Environment

Host, Node

Languages

Ada, C, Fortran

ISENDRECV *(cont.)*

ISENDRECV *(cont.)*

Description of Parameters

sendtype — A non-negative integer that specifies the type of the message that you are sending.

sendbuf — A pointer to the buffer that contains the message to be sent.

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

destnode — An integer that determines the node(s) to which the message is to be sent.

destpid — An integer that specifies the id of the process that is to receive the message. Valid *pids* include any integer value. Negative numbers are reserved for system programs. A node *pid* is assigned when you **load()** a process, and a host pid is assigned with **setpid()**.

typesel — An integer value that specifies the type(s) of the reply message.

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

recvlen — An integer value that specifies the size of the reply message buffer in bytes.

Discussion

Use **isendrecv()** to send a message to a node or host process, and to simultaneously post a receive for the reply. When a message arrives whose type matches the type(s) specified in the *typesel* parameter, the calling process receives the message, and stores it in the receive buffer whose address is specified by *recvbuf*. The calling process continues executing instructions while the send and receive operation is occurring.

When it is executed, **isendrecv()** returns a message id. Use **msgdone()** or **msgwait()** to free the message id numbers and to determine whether the **isendrecv()** routine has completed. You cannot use the **info...()** calls to get information about the reply message because **isendrecv()** does not set up this information.

ISENDRECV *(cont.)*

ISENDRECV *(cont.)*

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

- A message with a force type bypasses the normal flow control mechanisms, and is not delayed when normal messages have filled the message buffers.
- Force types do not match the `-1` wildcard type selector. This property can be used to guarantee that the message is received by the proper buffer, no matter what other messages are also received.
- If no receive is posted, as when the receiving process has been killed, a message with a force type is discarded. In general, bypassing the normal flow control causes no problem because **isendrecv()** guarantees that a receive is posted for the message.

Types 1,000,000,000 to 1,073,741,823 and 2,000,000,000 and up are used by the system, and should be avoided. Their use may produce unpredictable results.

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

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

If *destnode* is a negative number up to a certain value, it causes a global send. If *destnode* is `-1`, the message is sent to all nodes. Other negative numbers cause the message to be sent to a cube composed of a set of the nodes surrounding the sender. The dimension of this cube is calculated as:

d - **nodedim()** - 1

where *d* is the dimension of the cube, and *d* is less than or equal to **nodedim()**.

If a global send specifies its own process (**mypid()**), it does not receive its own message. However, if a process other than its own is specified, the sending node also receives the message.

ISENDRECV *(cont.)*

ISENDRECV *(cont.)*

The *typesel* parameter describes the type(s) of the message(s) expected to be received. The *typesel* (type selector) lets you select a specific message type or a set of message types based on a 32-bit mask.

The *typesel* parameter can be set as follows:

- If *typesel* is a non-negative integer, a specific message type will be recognized. All other messages will be ignored. You can use a force type, as described previously.
- If *typesel* is `-1`, the first message to arrive for the process that initiated the receive operation will be recognized. After the first message has been received, you can use `-1` again to receive the next message.
- If *typesel* is any negative number other than `-1`, a set of message types will be recognized. For information on building a *typesel* mask, refer to Appendix B in the *iPSC®/2 and iPSC®/860 Programmer's Reference Manual.*

iPSC Exceptions Raised

`No_pid_defined`

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

`Invalid_type`

Use a non-negative number.

`Invalid_length`

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

`Invalid_node`

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

`Buffer_length_exceeds_allocation`

Make sure the length parameter does not exceed the buffer size.

`Invalid_buffer_pointer`

Specify the address of a valid data buffer.

ISENDRECV *(cont.)*

ISENDRECV *(cont.)*

`Not_enough_memory`

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

`Received_message_too_long_for_buffer`

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

`Too_many_requests`

Use **msgcancel()**, **msgdone()**, or **msgwait()** for outstanding asynchronous requests.

See Also

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

TASKING LIBRARY CALLS 4

This chapter documents the iPSC/2 routines that are specific to the tasking version of the Ada system's host and node interface libraries.

The routines are listed alphabetically and are documented as follows:

- Name and brief description
- Synopsis
- Return values (where applicable)
- Example
- Environment
- Languages
- Detailed description (including input parameters)
- iPSC exceptions raised
- List of related routines

Table 4-1 lists all the tasking calls and tells whether each is described in this chapter or in Chapter 2.

Table 4-1. Ada Tasking Calls

Call	Chapter	Call	Chapter
atta			
cprobe()	4	infonode()	4
crecv()	4	infopid()	4
csend()	4	infotaskid()	4
csendrecv()	4	infotype()	4
cubeinfo()	2	iprobe()	4
flick()	4	irecv()	4
flushmsg()	4	isend()	4
gdhigh()	2	isendrecv()	4
gdlow()	2	killcube()	2
gdprod()	2	killproc()	2
gdsum()	2	killsyslog()	2
getcube()	2	led()	2
giand()	2	load()	2
gihigh()	2	mclock()	2
gilow()	2	msgcancel()	2
ginv()	2	msgdone()	2
gior()	2	msgwait()	2
giprod()	2	my_taskid()	4
gisum()	2	myhost()	2
gixor()	2	mynode()	2
gland()	2	mypid()	2
glor()	2	newserver()	2
glxor()	2	nodedim()	2
gray()	2	numnodes()	2
gsendx()	2	relcube()	2
gshigh()	2	set_taskid()	4
gslow()	2	setpid()	2
gsprod()	2	setsyslog()	2
gssum()	2	waitall()	2
gsync()	2	waitone()	2

CPROBE CPROBE

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

Synopsis

procedure **cprobe(** *type* : in integer);

Return Value

None

Example

The following example uses **cprobe()** to block the calling task until the message of the given type is available to be received. Then, after using **infocount()** to ensure that the message is of the proper length, the example calls **crecv()** to receive the message.

```
    •
INIT_TYPE : constant integer := 10;
BUF_SIZE  : constant integer := 80;
    •
msg_id    : integer;
msglen    : integer;
msgbuf    : string(1..BUF_SIZE);
    •
cprobe(INIT_TYPE);                          (Wait for message of type 10)
msglen := infocount;
if (msglen <= BUF_SIZE) then                (If message is right length,
    crecv(INIT_TYPE, msgbuf'address, BUF_SIZE);     receive message)
end if;
```

Environment

Host, Node

CPROBE *(cont.)*

CPROBE *(cont.)*

Languages

Ada, C, Fortran

Description of Parameters

type — An integer value that specifies the type of message that you are waiting to receive. You assign a type to a message when you initiate a send operation. **cprobe()** returns when a message arrives with the exact type as the type requested.

Discussion

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

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

IPSC Exceptions Raised

`No_pid_defined`

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

`Too_many_requests`

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

`Invalid_type`

Cannot probe for force-type messages.

See Also

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

CRECV CRECV

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

Synopsis

```
procedure crecv(      type    : in integer;
                      recvbuf : in system.address;
                      recvlen : in integer);
```

Return Value

None

Example

The following example uses **crecv()** to initiate the receive as soon as a message of type 1 is available. It will receive the message into the buffer named *buf*, of length 256.

```
    •
BUFLEN : constant integer := 256;
    •
buf : string(1..BUFLEN);
    •
crecv(1, buf'address, BUFLEN);      (Receive the first type 1 message to arrive
                                     and store it in "buf")
```

Environment

Host, Node

Languages

Ada, C, Fortran

CRECV *(cont.)*

CRECV *(cont.)*

Description of Parameters

type — An integer value that specifies the type of message that you are waiting to receive. You assign a type to a message when you initiate a send operation.

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

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

Discussion

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

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

CRECV *(cont.)*

CRECV *(cont.)*

iPSC Exceptions Raised

`No_pid_defined`

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

`Invalid_length`

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

`Too_many_requests`

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

`Buffer_length_exceeds_allocation`

Make sure the length parameter does not exceed the buffer size.

`Invalid_buffer_pointer`

Specify the address of a valid data buffer.

`Received_message_too_long_for_buffer`

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

See Also

csend(), **infocount()**, **infonode()**, **infopid()**, **infotaskid()**, **infotype()**, **irecv()**, **isend()**

CSEND CSEND

Send a message. Blocks the task until execution is complete.

Synopsis

```
procedure csend(      type : in integer;
                   sendbuf : in system.address;
                   sendlen : in integer;
                  destnode : in integer;
                   destpid : in integer;
                dest_taskid : in integer);
```

Return Value

None

Example

The following example defines a message ("Hello node 0") and uses **csend()** to send it to task 0 in an Ada program on node 0. Using the same message buffer, a new message is defined ("Hello host") and **csend()** is used again to send it to task 1 on the host.

```
    •
msg : string(1..80);
len : integer;
    •
msg(1..12) := "Hello node 0";
len        := 12;
    •
csend(5, msg'address, len, 0, mypid, 0);          (Send type 5
    •                                  message to task 0 in process on node 0)
msg(1..10) := "Hello host";
    •
csend(2, msg'address, 10, myhost, mypid, 1);      (Send new type 2
    •                                  message to task 1 in process on host)
```

CSEND *(cont.)*

CSEND *(cont.)*

Environment

Host, Node

Languages

Ada, C, Fortran

Description of Parameters

type	An integer that specifies the type of the message that you are sending.
sendbuf	A pointer to the buffer that contains the message to be sent. The buffer may be any legal data type. It is recommended that data types match in send and receive operations.
sendlen	A positive integer that specifies the size (in bytes) of the message that you wish to send. Messages to or from a host process are limited to 256K bytes. There is no limit on message length between nodes.
destnode	An integer that defines where the message is to be sent.
destpid	An integer that specifies the id of the process that is to receive the message.
dest_taskid	An integer that specifies the id of the Ada task that is to receive the message.

Discussion

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

Use a type with a value from 0 to 15,257. Types 16,384 to 30,517 are special force types that bypass the normal flow control mechanisms, and are discarded if no receive is posted. Force types are intended to be used with **csendrecv()** and **isendrecv()**, but can be used by other message-passing calls. Do not use types in other ranges because they are used by the system and could produce unpredictable results.

CSEND *(cont.)*

CSEND *(cont.)*

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

If *destnode* is a negative number up to a certain value, it causes a global send. If *destnode* is `-1`, the message is sent to all nodes. Other negative numbers cause the message to be sent to a cube that is composed of a set of the nodes surrounding the sender. The dimension of this cube is calculated as:

d - **nodedim()** - 1

where *d* is the dimension of the cube, and *d* is less than or equal to **nodedim()**.

If a global send specifies its own process (**mypid()**), it does not receive its own message. However, if a process other than its own is specified, the destination process on the sending node also receives the message.

Valid *destpids* include any integer value. Negative numbers are reserved for system programs. The node *destpid* is assigned when you **load()** a process, and a host *destpid* is assigned with **setpid()**.

To initiate a send request when you want the calling task to continue during the send operation, use **isend()**.

iPSC Exceptions Raised

`No_pid_defined`

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

`Invalid_type`

Use a non-negative number.

`Invalid_length`

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

`Invalid_node`

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

CSEND *(cont.)*

CSEND *(cont.)*

`Buffer_length_exceeds_allocation`

Make sure the length parameter does not exceed the buffer size.

`Invalid_buffer_pointer`

Specify the address of a valid data buffer.

`Not_enough_memory`

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

See Also

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

CSENDRECV CSENDRECV

Send a message and receive a reply. Block the calling task until the reply is received.

Synopsis

```
function csendrecv(     sendtype  : in integer;
                        sendbuf   : in system.address;
                        sendlen   : in integer;
                        destnode  : in integer;
                        destpid   : in integer;
                        dest_taskid : in integer;
                        recvtype  : in integer;
                        recvbuf   : in system.address;
                        recvlen   : in integer) return integer;
```

Return Value

The length of the received message.

Example

The following example shows how to use the **csendrecv()** call:

- Part 1 is a procedure that uses **csendrecv()** to make a remote procedure call and post a receive for the message containing the result.
- Part 2 is a portion of the server running on the receiving node that receives the message, performs the operation, and sends back the result to the first procedure.

CSENDRECV *(cont.)*

CSENDRECV *(cont.)*

Part 1:

```
    •
x      : float;
result : float;
len    : integer;
    •
len := csendrecv(SIN_REQUEST_TYPE,        (Send value x to remote process,
                 x'address, 8, SERVER_NODE,            post a receive,
                 SERVER_PID, SERVER_TASK,      and put result in buffer)
                 SIN_REPLY_TYPE, result'address, 8);
return result;
```

Part 2:

```
    •
x, result                                        : float;
request_node, request_pid, request_taskid, i : integer;
    •
i := 1;
while (i = 1) loop
    crecv(SIN_REQUEST_TYPE, x'address, 8);
    request_node   := infonode;
    request_pid    := infopid;
    request_taskid := infotaskid;
        •
        •                                          (Do processing here)
        •
    csend(SIN_REPLY_TYPE, result'address, 8, request_node,
          request_pid, request_taskid);                (Return result)
end loop;
```

Environment

Host, Node

Languages

Ada, C, Fortran

CSENDRECV *(cont.)*

CSENDRECV *(cont.)*

Description of Parameters

sendtype — A non-negative integer that specifies the type of the message that you are sending.

sendbuf — A pointer to the buffer that contains the message to be sent.

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

destnode — An integer that determines the node(s) to which the message is to be sent.

destpid — An integer that specifies the id of the process that is to receive the message. Valid *pids* include any integer value. Negative numbers are reserved for system programs. A node *pid* is assigned when you **load()** a process, and a host *pid* is assigned with **setpid()**.

dest_taskid — A positive integer in the range 0 to 32,767 that specifies the id of the task that is to receive the message.

recvtype — An integer value that specifies the type of the reply message.

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

recvlen — An integer value that specifies the size of the reply message buffer in bytes.

Discussion

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

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

CSENDRECV *(cont.)*

CSENDRECV *(cont.)*

The *sendtype* parameter is a user-defined variable that is used to identify the kind of information contained in the message being sent. Types 0 to 15,257 are normal types that can be used by any send or receive call, including this one. Types 16,384 to 30,516 are special force types that are available to the user specifically for the send and receive calls.

Force types have these special properties:

- A message with a force type bypasses the normal flow control mechanisms, and is not delayed when normal messages have filled the message buffers.
- If no receive is posted, as when the receiving process has been killed, a message with a force type is discarded. In general, bypassing the normal flow control causes no problem because **csendrecv()** guarantees that a receive is posted for the message.

Types 15,257 to 16,383 and greater than 30,517 are used by the system and should be avoided. Their use may produce unpredictable results.

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

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

If *destnode* is a negative number up to a certain value, it causes a global send. If *destnode* is `-1`, the message is sent to all nodes. Other negative numbers cause the message to be sent to a cube that is composed of a set of the nodes surrounding the sender. The dimension of this cube is calculated as:

d - **nodedim()** - 1

where *d* is the dimension of the cube, and *d* is less than or equal to **nodedim()**.

The *recvtype* parameter describes the type of the message expected to be received. The message type must match *recvtype* exactly.

CSENDRECV *(cont.)*

CSENDRECV *(cont.)*

iPSC Exceptions Raised

`No_pid_defined`

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

`Invalid_type`

Use a non-negative number.

`Invalid_length`

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

`Invalid_node`

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

`Buffer_length_exceeds_allocation`

Make sure the length parameter does not exceed the buffer size.

`Invalid_buffer_pointer`

Specify the address of a valid data buffer.

`Not_enough_memory`

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

`Received_message_too_long_for_buffer`

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

See Also

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

FLICK FLICK

Relinquish node processor to another task.

Synopsis

procedure **flick;**

Return Value

None

Example

The following example shows how to use the **flick()** call. Two asynchronous receives are posted and each returns a message id (*id1* and *id2*). After some other processing, a while-loop tests to see if either message has been received, and if so, processes the one received. If not, **flick()** defers execution of this process, to allow another task on the same node to execute while waiting for these messages to be received.

```
    •
node, id1, id2, i, buf1, buf2 : integer;
    •
id1 := irecv(101, buf1'address, 4);
id2 := irecv(102, buf2'address, 4);
    •                                          (Do some other processing here)
i := 0;
while i = 0 loop
    if (msgdone(id1) = 1) then
            •                                          (Process message 1)
        exit;
    elsif (msgdone(id2) = 1) then
            •                                          (Process message 2)
        exit;
    else
        flick;
    end if
end loop;
```

FLICK *(cont.)*

FLICK *(cont.)*

Environment

Host, Node

Languages

Ada, C, Fortran

Discussion

Use **flick()** on one task to temporarily relinquish the node processor to another task on the same node. The execution of the task is not deferred indefinitely. Instead, the program will resume execution during its next scheduled time frame. The **flick()** call allows other node processes to execute. The **csend()**, **crecv()**, **cprobe()**, or **msgwait()** call also defer execution of a calling task until they complete their task.

You should use the **flick()** call when the current task has no useful work to do, for example, when the task is waiting for any of several **irecv()** calls to complete. A task should never "busy-loop" without flicking. The task sending the message may be running on the same node, but be suspended waiting for the CPU.

The **flick()** call is a delay mechanism, not a synchronization mechanism.

(The **flick()** call performs no operation on a host machine or in a nontasking program. It is included so that programs that call **flick()** will run in these environments.)

IPSC Exceptions Raised

None

See Also

mclock()

FLUSHMSG FLUSHMSG

Flush specified messages from the system.

Synopsis

```
procedure flushmsg(    type : in integer;
                       node : in integer;
                        pid : in integer);
```

Return Value

None

Example

The following example sends two synchronous messages and one asynchronous message to the Ada program on node 1. It then flushes all messages for each type waiting to be received for that process and node. By issuing **msgcancel()** to cancel the asynchronous message (*msg3*) before the **flushmsg()** call, you are assured that all of the messages are cancelled.

```
    •
node, pid, msg_id : integer;
    •
node := 1;
pid  := mypid;
    •
csend(10, msg1'address, msg1'size / 8, node, pid, 0);
                                    (Blocking send to Ada task 0 on node 1)
csend(11, msg2'address, msg2'size / 8, node, pid, 0);
                                    (Blocking send to Ada task 0 on node 1)
msg_id := isend(12, msg3'address, msg3'size / 8, node,
               pid, 0);          (Asynchronous send to Ada task 0 on node 1)
    •
msgcancel(msg_id);                                  (Cancel msg3 send)
flushmsg(12, node, pid);                        (Now msg3 will be flushed)
flushmsg(11, node, pid);
flushmsg(10, node, pid);
```

FLUSHMSG *(cont.)*

FLUSHMSG *(cont.)*

Environment

Host, Node

Languages

Ada, C, Fortran

Description of Parameters

type — An integer value that specifies the type of message that you want to flush. The type depends on the type specified in the send. You must select a specific message type to flush.

node — A positive integer that specifies the receiving node (`-1` specifies all nodes and the host). Nodes in a cube are numbered consecutively from 0. If you want to flush messages only from the host, use **myhost()** to determine the host's node number.

pid — An integer that specifies the receiving process id (`-1` specifies all processes). Valid *pids* include any integer value, but negative numbers other than `-1` are reserved for system programs.

Discussion

Use **flushmsg()** to flush all messages of the specified type from system buffers. All messages of the type selected by *type* in the selected nodes(s) that have been sent to the selected *pid*(s) will be eliminated from the system.

Use of **flushmsg()** has no effect on node processes. It also has no effect on messages in transit to processes that do not have the selected *pid*, even if they were sent by a process that does have a matching *pid*. That is, **flushmsg()** flushes messages by the destination *pid*, not the source *pid*. If a particular node is selected, the only messages affected are those that are sent to the selected node, and which have arrived on the node, but have not been received.

To ensure that all messages are flushed, use **flushmsg()** with **msgcancel()**.

FLUSHMSG *(cont.)*

FLUSHMSG *(cont.)*

IPSC Exceptions Raised

`Invalid_node`

Use **numnodes()** to determine cube size, and use **myhost()** to determine host number.

See Also

msgcancel(), killcube()

INFO: INFOCOUNT, INFONODE, INFOPID, INFOTYPE

Return information about a pending or received message.

Synopsis

function **infocount** return integer;

function **infonode** return integer;

function **infopid** return integer;

function **infotaskid** return integer;

function **infotype** return integer;

Return Value

The indicated information. The return value is defined only if the call is made immediately after a **crecv()**, **cprobe()**, or **msgwait()** completes, or after an **iprobe()** or **msgdone()** returns `1`, all of which indicate that a message is ready.

infocount	Returns an integer value specifying the length of the message.
infonode	Returns the node ID of the task that sent the message.
infopid	Returns the process id of the task that sent the message.
infotaskid	Returns the user-defined task id of the task that sent the message.
infotype	Returns the type of the message. Message type is specified in the send operation.

INFO... *(cont.)*

INFO... *(cont.)*

Example

The following example uses all of the **info...()** calls. First, it waits for a message of type 0 and then uses **infotype()** to verify the message type, and **infocount()** to determine the actual message length. Next, it uses that information to receive the waiting message. Finally, the example posts an **irecv()** for a message of type 10, and if it has been received, uses **infonode()**, **infopid()**, and **infotaskid()** to find the node, process, and task id from which the message came.

```
    •
BIGNUM : constant integer;
    •
type intbuf_type is array (1..BIGNUM) of integer;
    •
buf         : intbuf_type;
msgtype     : integer;
msglen      : integer;
msg_id      : integer;
from_node   : integer;
from_pid    : integer;
from_taskid : integer;
status      : integer;
    •
cprobe(0);                                    (Wait for type 0 message to arrive)
msgtype := infotype;                                    (Get message type)
msglen  := infocount;                                   (Get message length)
crecv(msgtype, buf'address, msglen);
    •
msg_id := irecv(10, buf'address, buf'size / 8);          (Post receive
    •                                                     for type 10 message)
status := msgdone(msg_id);
if (status = 1) then
    from_node   := infonode;
    from_pid    := infopid;
    from_taskid := infopid;
end if;
```

Environment

Host, Node

INFO... *(cont.)*

INFO... *(cont.)*

Languages

Ada, C, Fortran

Description of Parameters

None

Discussion

Use **infocount()**, **infonode()**, **infopid()**, **infotaskid()**, and **infotype()** to return information about a pending or received message. They return information about the message detected when **cprobe()** completes or when **iprobe()** returns `1`. They return information about the message received when **crecv()** or **msgwait()** return or when **msgdone()** for a previous **irecv()** returns a `1`.

iPSC Exceptions Raised

None

See Also

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

IPROBE IPROBE

Determine whether a message of a selected type is pending.

Synopsis

function **iprobe**(*type* : in integer) return integer;

Return Value

0	A message of the selected type is not waiting to be received.
1	A message of the selected type is available to be received.

Example

The following example contains a continuous while-loop that uses **iprobe()** to test for the availability of two messages (type 1 and 2) and processes the first message to come in. If neither message is available, **flick()** allows another process on the same node to execute before looping again.

```
    •
i : integer := 0;
    •
while i = 0 loop
    if (iprobe(1) = 1) then
            •
            •                                    (Process type 1 message)
            •
        exit;
    elsif (iprobe(2) = 2) then
            •
            •                                    (Process type 2 message)
            •
        exit;
    else
        flick;
    end if;
end loop;
```

IPROBE *(cont.)*

IPROBE *(cont.)*

Environment

Host, Node

Languages

Ada, C, Fortran

Description of Parameters

type — An integer value that specifies the type of message for which you are waiting. You assign a type to a message when you initiate a send operation. The message type and *type* parameter must match exactly for **iprobe()** to return a `1`.

Discussion

Use **iprobe()** to determine whether a message of a selected type is available for receipt. You specify the message type with the send operation, and **iprobe()** looks for a message whose type matches the type 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, it returns a `0`. When **iprobe()** returns a `1`, the **info...()** calls can be used to get more information about the message.

This is an asynchronous call and, therefore, does not block the task waiting for a message to arrive. Use **cprobe()** to block the task until a message of a selected type is available for receipt.

IPSC Exceptions Raised

`No_pid_defined`

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

`Too_many_requests`

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

`Invalid_type`

Cannot probe for force-type messages.

IPROBE *(cont.)*

IPROBE *(cont.)*

See Also

cprobe(), infocount(), infonode(), infopid(), infotaskid(), infotype()

IRECV IRECV

Initiate the receipt of a message.

Synopsis

```
function irecv(        type    : in integer;
                       recvbuf : in system.address;
                       recvlen : in integer) return integer;
```

Return Value

A non-negative message id to be used in **msgcancel()**, **msgdone()**, or **msgwait()**.

Example

The following example uses **irecv()** (to initiate receipt of a message) and **msgwait()** (to block execution of the process until the receive is complete).

```
    •
BUFLEN : constant integer := 256
    •
buf           : string(1..BUFLEN);
msg_id, type  : integer;
    •
msg_id := irecv(type, buf'address, BUFLEN);     (Initiate message receipt)
    •
    •                 (Perform tasks that do not depend on the message being received)
    •
msgwait(msg_id);                                (Wait to guarantee message receipt)
    •
    •                                  (Now use the contents of "buf" and the info calls)
```

IRECV *(cont.)*

IRECV *(cont.)*

Environment

Host, Node

Languages

Ada, C, Fortran

Description of Parameters

type — A non-negative integer value that specifies the type of message for which you are waiting. You assign a type to a message when you initiate a send operation.

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

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

Discussion

Use **irecv()** to post an asynchronous receive that initiates the receipt of a message from a process. It searches for a message whose type matches the type specified in the *type* parameter, and immediately returns a message id that is used by **msgdone()** and **msgwait()** to determine whether the receive is complete. The **msgwait()** routine also clears the id assigned to the message as soon as the receive completes, as does **msgdone()** when it returns a `1`. You must use **msgwait()**, **msgdone()**, or **msgcancel()** to free this id for future sends and receives because the number of ids available is limited, and an error results if you use all the ids. When the message is received, it is stored in the buffer at address *recvbuf*.

After **msgwait()** completes or **msgdone()** returns `1`, you can use the **info...()** calls to get more information about the message. The **irecv()** call does not affect the values returned by the **info...()** calls; it only initiates the receipt of the message.

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

The **irecv()** routine is asynchronous, allowing the calling task to continue even if the desired message is not yet available. It returns as soon as the receive request has been initiated. Use **crecv()** when you want the calling task to be blocked until the desired message is received.

IPSC Exceptions Raised

`No_pid_defined`

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

`Invalid_length`

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

`Too_many_requests`

Use **msgcancel()**, **msgdone()**, or **msgwait()** for outstanding asynchronous requests.

`Buffer_length_exceeds_allocation`

Make sure the length parameter does not exceed the buffer size.

`Received_message_too_long_for_buffer`

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

`Invalid_node`

Use **numnodes()** to determine the cube size, and use **myhost()** to determine the host number.

`Invalid_buffer_pointer`

Specify the address of a valid data buffer.

See Also

crecv(), **csend()**, **infocount()**, **infonode()**, **infopid()**, **infotaskid()**, **infotype()**, **isend()**

ISEND ISEND

Initiate a message send.

Synopsis

```
function isend(      type      : in integer;
                     sendbuf   : in system.address;
                     sendlen   : in integer;
                     destnode  : in integer;
                     destpid   : in integer;
                     dest_taskid : in integer) return integer;
```

Return Value

A non-negative message id to be used in **msgcancel()**, **msgdone()**, or **msgwait()**.

Example

The following example uses **isend()** to send a message of type 1 to task 1 on node 16. When the send buffer is required, the example uses **msgwait()** to block execution of the process until the send is complete, and to make the id used for the send available for subsequent asynchronous message-passing calls.

```
    •
msg                                  : string(1..132);
type, len, node, pid, task, msg_id : integer;
    •
type := 1;
len  := msg'size / 8;
node := 16;
pid  := 0;
task := 1;
    •
msg_id := isend(type, msg'address,                    (Send type 1 message
                len, node, pid, task);                 to task 1 on node 16)
    •
msgwait(msg_id);                      (Now you can use the message buffer again)
```

ISEND *(cont.)*

ISEND *(cont.)*

Environment

Host, Node

Languages

Ada, C, Fortran

Description of Parameters

type	An integer that specifies the type of the message that you want to send. Type restrictions are described in the "Discussion".
sendbuf	A pointer to the buffer that contains the message you want to send. The buffer may be any legal data type. It is recommended that data types match in send and receive operations.
sendlen	A positive integer that specifies the size (in bytes) of the message that you wish to send. Messages to or from a host process are limited to 256K bytes. There is no limit on message length between nodes. If *sendlen* is greater than the maximum message length, an error message will be issued.
destnode	An integer that specifies the node or nodes to which the message is to be sent.
destpid	An integer that specifies the process id that is to receive the message. Valid *destpids* include any integer value. Negative numbers are reserved for system programs. A node *destpid* is assigned when you **load()** a process and a host *destpid* is assigned with **setpid()**.
dest_taskid	An integer that specifies the id of the Ada task that is to receive the message.

Discussion

Use **isend()** to send a message to a node or host program. Completion of **isend()** does not imply that the message was received by the destination process, only that the send operation was initiated. When the message is sent, the buffer can be used for another message, if desired.

ISEND *(cont.)*

ISEND *(cont.)*

This call is asynchronous, returning as soon as the send request is initiated. Use **csend()** when you want to block the calling process until the message buffer is available for reuse.

It is recommended that you use a type with a value from 0 to 15,257. Types 16,384 to 30,516 are special force types that bypass the normal flow control mechanisms, and are discarded if no receive is posted. Force types are intended to be used with **csendrecv()** and **isendrecv()**, but can be used by other message-passing calls. It is recommended that you not use types in other ranges because they are used by the system and could produce unpredictable results.

The buffer can be of any type. You should not modify the buffer contents until **msgwait()** returns or **msgdone()** indicates that **isend()** is complete. Otherwise, corrupted data might be sent.

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

If *destnode* is a negative number up to a certain value, it causes a global send. If node is `-1`, the message is sent to all nodes. Other negative numbers cause the message to be sent to a cube composed of a set of the nodes surrounding the sender. The dimension of this cube is calculated as:

d - **nodedim()** - 1

where *d* is the dimension of the cube, and *d* is less than or equal to **nodedim()**.

If a global send specifies its own process (**mypid()**), it does not receive its own message. However, if a process other than its own is specified, the sending node also receives the message.

iPSC Exceptions Raised

`No_pid_defined`

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

`Invalid_type`

Use a non-negative number.

`Invalid_length`

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

`Invalid_node`

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

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

`Buffer_length_exceeds_allocation`

Make sure the length parameter does not exceed the buffer size.

`Invalid_buffer_pointer`

Specify the address of a valid data buffer.

`Not_enough_memory`

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

`Too_many_requests`

Use **msgcancel()**, **msgdone()**, or **msgwait()** for outstanding asynchronous requests.

See Also

cprobe(), **crecv()**, **csend()**, **csendrecv()**, **iprobe()**, **irecv()**, **isendrecv()**, **msgcancel()**, **msgdone()**, **msgwait()**

ISENDRECV ISENDRECV

Send a message and receive a reply.

Synopsis

```
function isendrecv(     sendtype  : in integer;
                        sendbuf   : in system.address;
                        sendlen   : in integer;
                        destnode  : in integer;
                        destpid   : in integer;
                        dest_taskid : in integer;
                        recvtype  : in integer;
                        recvbuf   : in system.address;
                        recvlen   : in integer) return integer;
```

Return Value

A non-negative message id to be used in **msgcancel()**, **msgdone()**, or **msgwait()**.

Example

The following example shows how to use the **isendrecv()** call:

- Part 1 is a procedure that uses **isendrecv()** to make a remote procedure call and post a receive for the message containing the result. When the result is required, **msgwait()** is used to block the process until result is received.
- Part 2 is a portion of the server running on the receiving node that receives the message, performs the operation and returns the result to the first procedure.

ISENDRECV *(cont.)*

ISENDRECV *(cont.)*

Part 1:

```
    •
x      : float;
result : float;
msg_id : integer;
    •
msg_id := isendrecv(SIN_REQUEST_TYPE,      (Send value x to remote process,
                    x'address, 8, SERVER_NODE,          post a receive,
                    SERVER_PID, SERVER_TASK,     and put result in buffer)
                    SIN_REPLY_TYPE, result'address, 8);
    •
msgwait(msg_id);
return result;
```

Part 2:

```
    •
x, result                                    : float;
request_node, request_pid, request_task, i : integer;
    •
i := 1;
while (i = 1) loop
    crecv(SIN_REQUEST_TYPE, x'address, 8);
    request_node  := infonode;
    request_pid   := infopid;
    request_task  := infotaskid;
        •
        •                                          (Do processing here)
        •
    csend(SIN_REPLY_TYPE, result'address, 8, request_node,
          request_pid, request_task);                  (Return result)
end loop;
```

Environment

Host, Node

Languages

Ada, C, Fortran

ISENDRECV *(cont.)*

ISENDRECV *(cont.)*

Description of Parameters

sendtype A non-negative integer that specifies the type of the message that you are sending.

sendbuf A pointer to the buffer that contains the message to be sent.

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

destnode An integer that determines the node(s) to which the message is to be sent.

destpid An integer that specifies the id of the process that is to receive the message. Valid *pids* include any integer value. Negative numbers are reserved for system programs. A node *pid* is assigned when you **load()** a process, and a host pid is assigned with **setpid()**.

dest_taskid A positive integer in the range 0 to 32,767 that specifies the id of the task that is to receive the message.

recvtype An integer value that specifies the type of the reply message.

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

recvlen An integer value that specifies the size of the reply message buffer in bytes.

Discussion

Use **isendrecv()** to send a message to a node or host process, and to simultaneously post a receive for the reply. When a message arrives whose type matches the type specified in the *recvtype* parameter, the calling process receives the message, and stores it in the receive buffer whose address is specified by *recvbuf*. The calling process continues executing instructions while the send and receive operation is occurring.

When it is executed, **isendrecv()** returns a message id. Use **msgdone()** or **msgwait()** to free the message id numbers and to determine whether the **isendrecv()** routine has completed. You cannot use the **info...()** calls to get information about the reply message because **isendrecv()** does not set up this information.

ISENDRECV *(cont.)*

ISENDRECV *(cont.)*

The *sendtype* parameter is a user-defined variable that is used to identify the kind of information contained in the message being sent. Types in the range 0 to 15,257 are normal types that can be used by any send or receive call, including this one. Types 16,384 to 30,515 are special force types that are available to the user specifically for the send and receive calls. Force types have these special properties:

- A message with a force type bypasses the normal flow control mechanisms, and is not delayed when normal messages have filled the message buffers.
- If no receive is posted, as when the receiving process has been killed, a message with a force type is discarded. In general, bypassing the normal flow control causes no problem because **isendrecv()** guarantees that a receive is posted for the message.

Types greater than 30,515 are reserved for system use. (These types are translated into the ranges 1,000,000,000 to 1,073,741,823 and 2,000,000,000 and up, once the shifting is done).

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

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

If *destnode* is a negative number up to a certain value, it causes a global send. If *destnode* is `-1`, the message is sent to all nodes. Other negative numbers cause the message to be sent to a cube composed of a set of the nodes surrounding the sender. The dimension of this cube is calculated as:

d - **nodedim()** - 1

where *d* is the dimension of the cube, and *d* is less than or equal to **nodedim()**.

If a global send specifies its own process (**mypid()**), it does not receive its own message. However, if a process other than its own is specified, the sending node also receives the message.

The *recvtype* parameter describes the type of the message expected to be received. The message type must match *recvtype* exactly.

ISENDRECV *(cont.)*

ISENDRECV *(cont.)*

iPSC Exceptions Raised

`No_pid_defined`

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

`Invalid_type`

Use a non-negative number.

`Invalid_length`

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

`Invalid_node`

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

`Buffer_length_exceeds_allocation`

Make sure the length parameter does not exceed the buffer size.

`Invalid_buffer_pointer`

Specify the address of a valid data buffer.

`Not_enough_memory`

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

`Received_message_too_long_for_buffer`

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

See Also

cprobe(), crecv(), csendrecv(), iprobe(), irecv(), isend(), msgcancel(), msgdone(), msgwait()

MY_TASKID MY_TASKID

Obtain the user-defined task id of the calling process.

Synopsis

function **my_taskid** return integer;

Return Value

The task id.

Example

The following example uses **my_taskid()** to return the current process id to be used in a **csend()**.

```
     •
buf   : string(1..100);
stype : integer;
     •
csend(stype, buf'address, buf'size / 8,        (Send message to process
          -1, mypid, my_taskid);      with same pid and taskid on all nodes.)
```

Environment

Host, Node

Languages

Ada, C, Fortran

Description of Parameters

None

MY_TASKID *(cont.)*

MY_TASKID *(cont.)*

Discussion

Use **my_taskid()** to return the user-defined task id for an Ada task. Use **mypid()** to return the id of the process.

IPSC Exceptions Raised

`No_pid_defined`

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

See Also

csend(), csendrecv(), flushmsg(), isend(), isendrecv(), mypid(), setpid(), set_taskid()

SET_TASKID SET_TASKID

Sets the id of an Ada task.

Synopsis

procedure **set_taskid(** *user_taskid* : in integer);

Return Value

None

Example

The following example sets the user-defined task id for this task.

```
set_taskid(10);                    (Set the task id to 10 for this task)
```

Environment

Host, Node

Languages

Ada, C, Fortran

Description of Parameters

user_taskid An integer in the range 0 to 32,767.

Discussion

Use **set_taskid()** to set the id of an Ada task. This is the id to use for the *dest_taskid* parameter in a send operation. If **set_taskid()** is not called, the task id defaults to `0`.

SET_TASKID *(cont.)*

SET_TASKID *(cont.)*

iPSC Exceptions Raised

None

See Also

mypid(), my_taskid()

CROSS-DEBUGGER COMMANDS 5

This chapter documents the iPSC/2 routines that are used with the Ada cross-debugger to debug node programs.

The routines are listed alphabetically and are documented as follows:

- Name and brief description
- Synopsis
- Description of parameters
- Discussion
- Example
- Environment
- Languages
- References to related information

A.DBNODE A.DBNODE

Executes the cross-debugger from the shell.

Synopsis

a.dbnode [*options*] [*executable_file*]

Description of Parameters

-i *file_name*
: (input) read command input from specified file. When the debugger commands in this file have all been read and executed, the debugger switches to reading from standard input (unless the file contains a **quit** or **exit** command). The commands can be any **set** commands or line-mode commands.

-p *VADS_library*
: (program) read program compilation information from the specified VADS library directory.

-sh
: (show) display the name of the cross-debugger executable but do not execute it.

-v
: (visual) invoke the screen-mode debugger.

<file_name
>file_name
>&file_name
: Normally, the debugger reads from the terminal. You may redirect any or all of standard input, standard output, and standard error to a file. The debugger may also be run in background by appending "&" to the invocation line. For example:

```
a.db my_prog < debug.in >& debug.out &
```

There are two restrictions:

1. You cannot use screen mode when debugging input is a file.

2. Your program cannot share the debugger's input file. Use **set input** to set the input file for your program. A sample *debug.in* file might be:

```
set input my_prog.in
r
quit
```

A.DBNODE *(cont.)*

A.DBNODE *(cont.)*

Discussion

In addition to the invocation line, the user can supply parameters to **a.dbnode** using a *.dbrc* file. During the debugger initialization, **a.dbnode** checks for *./.dbrc*. If that does not exist, it checks for *$HOME/.dbrc*. The *.dbrc* file may contain only **set** commands. These commands are executed before any other commands, including those in the input file specified on the command line.

If *executable_file* is not specified, **a.dbnode** searches for **a.vox**. If only a root file name is given (*foo*, as opposed to */usr/bohunk/foo*), **a.dbnode** searches the directories on the *$PATH* environment (exported) variable for an executable file *foo* just as the shell does.

If a *.dbrc* file is present, the debugger will also search directories specified in a **set source** command for the first directory that contains an *ada.lib* file. That directory is used by the debugger to obtain the *DIANA* net files and the line number files produced by the compiler.

The **quit** command is used to leave the debugger and return to the shell.

The debugger establishes the debugging environment when it starts up. Certain key parameters are displayed on the screen to verify what and where it is debugging.

```
Debugging /usr/joe/tst3/nodeprog
VADS_library:  /usr/joe/tst3
library search list:
       /usr/joe/tst3
       /usr/vads/standard
       /usr/ipsc/ada/lib
db_iface process:  /usr/vads/sub/crossdeb

this is db_iface!

initialized ok
```

The first line shows the name of the program being debugged. The *VADS_library* is the name of the VADS library directory being used for this debugging session. The library search list is derived from the *ada.lib* file in the *VADS_library*. It shows the VADS library directories that are searched when the debugger looks for an Ada unit. The search list is printed out in the same order that the debugger searches it. In the above example, */usr/joe/tst3* is searched first.

The *db_iface* process is the name of the cross-debugger being invoked. The lines below it indicate that it has been successfully initiated.

A.DBNODE *(cont.)*

A.DBNODE *(cont.)*

Environment

Node

Languages

Ada

See Also

iPSC®/2 Ada Program Development Guide:
Debugger Reference, *screen mode*;
Library Tools, **a.mklib**;
VADS Debugger;
Invoking the Debugger;
Ada Library Directory, **-p**;
DB_IFACE, *process startup*

CONTEXT CONTEXT

Sets the debug context. The debug context defines the process that is the target of the debug command.

Synopsis

context (*node*:*pid*)

Description of Parameters

node	Specifies the node on which you wish to run the cross-debugger.
pid	Specifies the NX/2 process ID.

Discussion

You *must* set up a default context with the **context** command before you load the file to be debugged. This is called the *current* context. This command is used in conjunction with the **pass** command.

Example

Set the context to process 0 on node 1.

```
pass context(1:0)
```

Environment

Node

Languages

Ada

See Also

iPSC®/2 Ada Program Development Guide: Debugger Reference, **pass**

SUMMARY OF ADA LIBRARY CALLS A

COMMON LIBRARY CALLS

```
procedure attachcube(     cubename : in string);

function cubeinfo(              ct : ct_ptr_type;
                          numslots : integer;
                            global : integer) return integer;

procedure gdhigh(                x : in out float_vector_type;
                                 n : in integer;
                              work : in float_vector_type);

procedure gdlow(                 x : in out float_vector_type;
                                 n : in integer;
                              work : in float_vector_type);

procedure gdprod(                x : in out float_vector_type;
                                 n : in integer;
                              work : in float_vector_type);

procedure gdsum(                 x : in out float_vector_type;
                                 n : in integer;
                              work : in float_vector_type);
```

```
procedure getcube(      cubename : in string;
                        cubetype : in string;
                        hostname : in string;
                            keep : in integer);

procedure giand(               x : in out int_vector_type;
                               n : in integer;
                            work : in int_vector type);

procedure gihigh(              x : in out int_vector_type;
                               n : in integer;
                            work : in int_vector_type);

procedure gilow(               x : in out int_vector_type;
                               n : in integer;
                            work : in int_vector_type);

function ginv(                 j : integer) return integer;

procedure gior(                x : in out int_vector_type;
                               n : in integer;
                            work : in int_vector type);

procedure giprod(              x : in out int_vector_type;
                               n : in integer;
                            work : in int_vector_type);

procedure gisum(               x : in out int_vector_type;
                               n : in integer;
                            work : in int_vector_type);

procedure gixor(               x : in out int_vector_type;
                               n : in integer;
                            work : in int_vector type);

procedure gland(               x : in out bool_vector_type;
                               n : in integer;
                            work : in bool_vector_type);
```

```
procedure glor(          x : in out bool_vector_type;
                         n : in integer;
                      work : in bool_vector_type);

procedure glxor(         x : in out bool_vector_type;
                         n : in integer;
                      work : in bool_vector_type);

function gray(           j : integer) return integer;

procedure gsendx(     type : in integer;
                         x : in system.address;
                      xlen : in integer;
                  nodenums : in int_vector_type;
                      nlen : in integer);

procedure gshigh(        x : in out sfloat_vector_type;
                         n : in integer;
                      work : in sfloat_vector_type);

procedure gslow(         x : in out sfloat_vector_type;
                         n : in integer;
                      work : in sfloat_vector_type);

procedure gsprod(        x : in out sfloat_vector_type;
                         n : in integer;
                      work : in sfloat_vector_type);

procedure gssum(         x : in out sfloat_vector_type;
                         n : in integer;
                      work : in sfloat_vector_type);

procedure gsync;

procedure killcube(   node : in integer;
                       pid : in integer);

procedure killproc(   node : in integer;
                       pid : in integer);

procedure killsyslog;
```

```
procedure led(          lstate : in integer);

procedure load(        filename : in string;
                           node : in integer;
                            pid : in integer);

function mclock return integer;

procedure msgcancel(         id : in integer);

function msgdone(            id : in integer);

procedure msgwait(           id : in integer);

function myhost return integer;

function mynode return integer;

function mypid return integer;

procedure newserver(   cubename : in string);

function nodedim return integer;

function numnodes return integer;

procedure relcube(     cubename : in string);

procedure setpid(            pid : in integer);

procedure setsyslog(     outfile : in integer);

procedure waitall(          node : in integer;
                             pid : in integer);

procedure waitone(          node : in integer;
                             pid : in integer;
                           cnode : in integer_ptr;
                            cpid : in integer_ptr;
                           ccode : in integer_ptr);
```

NONTASKING LIBRARY CALLS

```
procedure cprobe(          typesel : in integer);

procedure crecv(           typesel : in integer;
                           recvbuf : in system.address;
                           recvlen : in integer);

procedure csend(             type : in integer;
                           sendbuf : in system.address;
                           sendlen : in integer;
                          destnode : in integer;
                           destpid : in integer);

function csendrecv(        sendtype : in integer;
                           sendbuf : in system.address;
                           sendlen : in integer;
                          destnode : in integer;
                           destpid : in integer;
                           typesel : in integer;
                           recvbuf : in system.address;
                           recvlen : in integer) return integer;

procedure flick;

procedure flushmsg(        typesel : in integer;
                              node : in integer;
                               pid : in integer);

function infocount return integer;

function infonode return integer;

function infopid return integer;

function infotype return integer;

function iprobe(           typesel : in integer) return integer;
```

```
function irecv(          typesel : in integer;
                         recvbuf : in system.address;
                         recvlen : in integer) return integer;

function isend(            type : in integer;
                         sendbuf : in system.address;
                          msglen : in integer;
                        destnode : in integer;
                          destpid : in integer) return integer;

function isendrecv(     sendtype : in integer;
                         sendbuf : in system.address;
                         sendlen : in integer;
                        destnode : in integer;
                          destpid : in integer;
                         typesel : in integer;
                         recvbuf : in system.address;
                         recvlen : in integer) return integer;
```

TASKING LIBRARY CALLS

```
procedure cprobe(          type : in integer);

procedure crecv(           type : in integer;
                         recvbuf : in system.address;
                         recvlen : in integer);

procedure csend(           type : in integer;
                         sendbuf : in system.address;
                         sendlen : in integer;
                        destnode : in integer;
                         destpid : in integer;
                      dest_taskid : in integer);

function csendrecv(      sendtype : in integer;
                         sendbuf : in system.address;
                         sendlen : in integer;
                        destnode : in integer;
                         destpid : in integer;
                      dest_taskid : in integer;
                         recvtype : in integer;
                         recvbuf : in system.address;
                         recvlen : in integer) return integer;

procedure flick;

procedure flushmsg(        type : in integer;
                           node : in integer;
                            pid : in integer);

function infocount return integer;

function infonode return integer;

function infopid return integer;

function infotaskid return integer;

function infotype return integer;
```

```
function iprobe(          type : in integer) return integer;

function irecv(           type : in integer;
                       recvbuf : in system.address;
                       recvlen : in integer) return integer;

function isend(           type : in integer;
                       sendbuf : in system.address;
                       sendlen : in integer;
                      destnode : in integer;
                       destpid : in integer;
                    dest_taskid : in integer) return integer;

function isendrecv(       sendtype : in integer;
                       sendbuf : in system.address;
                       sendlen : in integer;
                      destnode : in integer;
                       destpid : in integer;
                    dest_taskid : in integer;
                       recvtype : in integer;
                       recvbuf : in system.address;
                       recvlen : in integer) return integer;

function my_taskid return integer;

procedure set_taskid(  user_taskid : in integer);
```