intel® Scientific Computers

iPSC™
TECHNICAL DESCRIPTION

September, 1987
Order Number: 175278-004

iPSC TECHNICAL DESCRIPTION

intel Corporation

iPSC Technical Description

REV.	REVISION HISTORY	DATE
-001	Original issue	10/85
-002	Revision	1/86
-003	Revision	9/86
-004	Revision	9/87

CONTENTS

CHAPTER 4 - DEVELOPING FORTRAN PROGRAMS

CHAPTER 5 - CUBE MANAGER COMMANDS

CHAPTER 6 - C ROUTINES

C ROUTINES (cont.)

CHAPTER 7- FORTRAN ROUTINES

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

iPSC BASIC SYSTEM DESCRIPTION

INTRODUCTION

The iPSC™ is the first family of expandable concurrent *"personal supercomputers."* It is designed to provide the research community with a system upon which to develop parallel programming techniques, tools, and application programs. This manual provides a detailed overview of the iPSC. More information can be found in the iPSC manual set.

The basic system has two main elements:

Cube

The complete ensemble of microcomputers connected in a parallel architecture. Each high-performance microcomputer, along with its own numeric processing unit and local memory, is referred to as a "node." Nodes are connected together by high-speed communication channels to form a self-contained "cube" in a free-standing enclosure.

The individual nodes are connected in a hypercube interconnect topology. This design eliminates the need for global or shared system resources. Each node is completely independent. The resident node operating system, in conjunction with communication coprocessors, allow each node to communicate with its neighbors by queued "message passing." This means that each node can execute its own computational task...on its own data set...without the processor interaction collisions encountered in systems using shared resources. Because a hypercube architecture has multiple independent processors...all working on different segments of the same problem...the iPSC is classified as a multiple instruction, multiple data (MIMD) machine.

Cube Manager

Intel's Multibus-based System 310AP microcomputer connected to each node by an Ethernet communication channel. The System 310AP provides the user interface to the cube as well as hosting the programming tools and system diagnostics. This station allows you to use the iPSC as either a stand-alone concurrent computer system or as a computational server within a distributed processing environment. In the latter case, the iPSC communicates with other systems through an optional Ethernet interface.

System software includes those elements residing in the nodes of the cube; other elements residing in the cube manager.

Node-Resident Software	A message-based operating system which supports message passing, input/output services, and process creation/destruction and a PROM-based node confidence test which initializes the board at system powerup.
Cube Manager	Includes the XENIX operating system, program development tools, and commands to manage the cube and monitor its status.

Various models and options are available (refer to Table 1-1). A system has one, two, or four computational units. In the standard system, each unit contains 32 nodes. In a Large Memory system, each unit contains 16 nodes and 16 LBX memory boards (name derived from the fact that the memory board is accessed through its companion node's LBX port). In a Vector Processing system, each unit contains 16 nodes and 16 vector processor boards.

The following pages provide more detailed information on both the hardware and software components of the iPSC system.

CUBE DESCRIPTION

The following sections present an overview of the hypercube topology, the cube hardware, and the cube software.

Hypercube Topology

The interconnection scheme, or topology, for the iPSC is a *"binary n-cube"* or *"hypercube"*. This is an *n*-dimensional cube where *n* represents the number of directly-connected nodes that establish the cube's dimension.

The Hypercube

A hypercube has 2^d identical nodes in which *d* represents the dimension of the hypercube. Note that the number of nodes is a power of two. Thus, a 2^5 hypercube has 32 nodes and a dimension 5.

In this topology, each node has *d* nearest neighbor nodes. One way of measuring the distance between nodes is the number of "hops" a message must take. The average distance between two arbitrary nodes is *d*/2; and the maximum distance is *d*. Thus, in a 32-node system (2^5), each node would have exactly 5 neighbors; the average distance between two arbitrary nodes would be 2.5 message units, and the maximum distance would be 5.

The hypercube interconnection for a D5 cube is physically implemented via the backplane within each computational unit. Internal cables connect units in larger dimension cubes.

Because of the versatility of the architecture, other topologies ...such as rings, trees, etc...can be mapped onto the system. These topologies can be realized by using certain techniques in the application code.

iPSC Models

The various iPSC models are designated by their dimension and type of boards. There are three types of available models: Standard systems, Large Memory systems, and Vector Processor systems.

Table 1-1
iPSC Models

Model Number	Dimension	Number of Nodes	Number and Type of LBX Boards
iPSC/D5	5	2^5 = 32 nodes	none
iPSC/D6	6	2^6 = 64 nodes	none
iPSC/D7	7	2^7 = 128 nodes	none
iPSC/D4MX	4	2^4 = 16 nodes	16 4-MB memory
iPSC/D5MX	5	2^5 = 32 nodes	32 4-MB memory
iPSC/D6MX	6	2^6 = 64 nodes	64 4-MB memory
iPSC/D4VX	4	2^4 = 16 nodes	16 vector processors
iPSC/D5VX	5	2^5 = 32 nodes	32 vector processors
iPSC/D6VX	6	2^6 = 64 nodes	64 vector processors

Cube Hardware

This section presents an overview of the cube hardware. A more detailed discussion of the various hardware components is covered in Chapter 3, Hardware Overview, of the iPSC System Overview Manual.

Node

Each node in the cube is an independent, single-board computer. The node contains an Intel 80286 central processing unit and its companion 80287 numeric processing unit which support 32-, 64-, and 80-bit floating-point formats. The node also contains 512K bytes of NMOS dynamic RAM, with byte parity. An initialization and self-test monitor is contained in 64K bytes of PROM.

Each node also contains eight bidirectional communication channels managed by dedicated 82586 communication coprocessors. Seven of these channels physically link the nodes together and serve as dedicated point-to-point communication channels. The eighth channel is a global Ethernet channel that provides direct access to and from the cube manager for program loading and message transmission.

Each node board has red and green LED indicators which are used for diagnostic and system purposes.

Node Expansion Port

For enhancement purposes, the local processor memory bus on each node is accessible via a standard MULTIBUS II iLBX high-speed bus interface. The iLBX II interface reaches the backplane via one of the two 96-pin DIN connectors. On the backplane, the iLBX bus is routed from each even-numbered board slot to the adjacent odd-numbered board slot. The odd-numbered slots may be populated by boards that extend the memory or processing capacity of the nodes as in iPSC Large Memory systems and Vector Processor systems.

Manager-to-Cube Communication Channels

There are two cube manager-to-node communication links: global and diagnostic.

The global link enables cube manager processes to communicate with the node processes. It physically links the cube manager to each node from the iSBC 186/51 communications board to the global communications coprocessor chip set on each node board. This link complies with the standard Ethernet specification (IEEE 802.3).

The diagnostic link is a separate path for communicating with the nodes. It links the cube manager to the unit services module (USM) through an RS 422 port from the iSBC 286/12 processor board in the System 310AP. The USM demultiplexes this channel into each node via the monitor bus, which consists of communication, interrupt, and reset lines. In the event of a failure, this alternate path is used to determine if the fault is within a node or within the global communication link to the nodes. It is used to reset the nodes, initiate on-board diagnostics, and monitor the results...a more fundamental level of communication than is possible over the global link.

Cube Enclosure

Each computational unit is a free-standing enclosure which is 49 inches high and 16 inches square with a 26.75" x 26.75" footprint. Each enclosure has 32 board slots, one spare board slot and the USM board in one 34-slot card cage. Communication transceivers, multiplexer, power supplies, air cooling system, and associated cables are also housed in the enclosure.

Power Distribution

A removable key, found in the switch on the front panel (inside the front door), turns on the cooling fans and + 5-volt power. Front panel lights indicate AC on and DC on in that sequence. A power module at the base of the unit houses a filter and circuit breaker which can be externally reset from the rear.

Unit 0 has the only key switch. Thus, in a multi-unit system, Unit 0 enables the power-up sequence for the other units.

Cube Software

Cube software consists of a Node Confidence Test (NCT) and a node operating system residing on each node.

Node Confidence Test

This PROM-based test runs automatically when the system is powered up or reset. It initializes each node by resetting and enabling the node memory, communication controllers, I/O controller, interrupt controller, and CPU. Node identity is also set by reading the slot ID from the backplane.

Node Operating System

A copy of the node operating system is loaded into each node after initialization and confidence testing have been successfully completed. The node operating system performs these basic functions:

Interprocess Communication - The node operating system provides users with a powerful set of communication routines in FORTRAN and C programming languages. The communication interface is consistent whether communicating with other processes in the same node, to remote nodes, or to processes in the cube manager. Sending and receiving can be synchronous or asynchronous. Messages are automatically routed from node-to-node, if necessary, to reach the destination process.

Reliable message delivery service is provided between nearest node neighbors. Data message length can vary from 0 to 16K bytes. Messages larger than 1K byte are automatically fragmented and reassembled at the destination node, transparently to the user process.

Process Management - Processes are executed for a given interval (50 milliseconds) in a round-robin fashion. The number of processes is limited only by available memory.

Physical Memory Management - Provides memory space for each process on the node as well as message buffering. Memory management is not directly visible to the user. Temporary message buffers are managed automatically.

Protected Address Space - Local descriptor tables storing code and data for each process provide a "firewall" between the node operating system and user space and between multiple user process spaces. This is hardware enforced so that the user cannot corrupt the node operating system.

CUBE MANAGER DESCRIPTION

The cube manager is the desktop programming station that provides programming support and system management. The following sections present an overview of the cube manager hardware and software.

Cube Manager Hardware

The cube manager hardware consists of a microcomputer system, related options, and an alphanumeric terminal.

Intel System 310AP Microcomputer

The System 310AP is a Multibus-based microcomputer built with Intel's 80286 CPU and 80287 numeric processing unit. It contains a $5\frac{1}{4}$" 140 megabyte Winchester disk, a 320K byte floppy disk, a 45 megabyte cartridge tape, and 2 megabyte ECC RAM memory. The 310AP also has an integrated Ethernet interface for communicating with the cube.

Internally, the 310AP has a 7-slot card cage powered by a 360-watt power supply with the following Multibus boards integrated into four slots:

iSBC 286/12 CPU board
iSBC 020EX iLBX memory board
iSBC 214 disk controller board/multimodule
iSBC 186/51 communications board

The communication board is connected to an external Ethernet transceiver which is cabled to the cube. An RS422 link runs from the CPU board to the cube to provide the diagnostic link. Another optional Ethernet (Excelan option) is used for communication with other computers.

Terminal

The terminal is an alphanumeric terminal which meets ANSI X3.64 specifications (VT100 compatible). It features high resolution characters (upper/lower case) on a 14" non-glare green phosphor tilt display with detachable keyboard. The 101 keys include a numeric pad and function keys in addition to a standard typewriter keypad.

Expansion Options

Printers and other peripherals can easily be added to the Cube Manager via a Centronics compatible printer port and three spare Multibus expansion slots. One slot is reserved for iLBX boards and the other two are available for standard MULTIBUS boards.

Expansion options include:

Ethernet, TCP/IP network option - Enables access to remote host environment resources such as mass storage, workstations, or printers. It consists of an Ethernet front-end processor and associated software. Provides protocol package conforming to current DOD ARPANET TCP/IP specifications.

2 MB memory expansion option - Expands memory capacity to 4 MB. It consists of a 2048K byte memory board with dual port capability (Multibus and iLBX interface). Includes double-bit error detection and single-bit error correction logic. An error status register provides error logging to the host CPU board.

Serial port option - Provides eight additional serial ports. It is an intelligent communications controller which functions as a single-board controller or as an intelligent slave for multi-terminal communications expansion. The on-board iAPX 188 CPU provides communications control and buffer management for up to eight programmable synchronous/asynchronous channels. The board includes 64K bytes of dual-ported parity RAM buffer space to handle messages at data rates up to 19.2 K Baud.

Over 1500 other board options - available from Intel and other Multibus vendors.

Cube Manager Software

The cube manager software is divided into three basic categories: programming and development software, cube manager commands, and diagnostics.

Programming & Development Software

XENIX Operating System	XENIX/286 R3.0 is a fully-licensed derivative of UNIX System III. It includes enhancements from University of California at Berkeley as well as Microsoft and Intel. It is compatible with UNIX Version 7.
FORTRAN Compiler	Full implementation of ANSI X3.9-1978 FORTRAN Standard. Supports programs and arrays larger than 64K bytes and IEEE floating-point arithmetic.
Macro Assembler	Native 286 assembler ("ASM 286") plus macro extension.
Tools	All the standard XENIX tools are available Additional tools developed at iSC

Cube Manager Commands

These commands are developed by iSC and may be invoked from the terminal or can be executed by the XENIX program "exec". Some of the functions performed by these commands are:

Cube Access	Allows a user to gain exclusive access to the cube for running an application. When finished, another command allows the user to release exclusive access.
Load	Loads both the node operating system and application code into the cube.
System Logfile	Updated with events relating to the operation of the system, for example, when a system error occurs. Each entry is date and time stamped so that the sequence of events is preserved.

Diagnostics

The iPSC diagnostics provide both confidence and diagnostic tests. The confidence tests are used to verify overall system integrity prior to normal usage. The diagnostic tests isolate problems at the individual board or system module level.

System Confidence Testing	At power up, or upon reset, PROM-based confidence tests in the cube manager and cube verify individual module integrity and basic system functionality. Immediately following, during the boot-up sequence, XENIX verifies the system configuration. If an error occurs during initialization confidence testing, you can invoke system diagnostic tests as appropriate.
System Diagnostic Testing	Using CDP (Cube Diagnostic Program) you can isolate problems at the board/system module level. The diagnostic strategy is to verify the functionality of the cube manager first, manager-to-cube communications next, and then the cube.

CHAPTER 2

iPSC PROGRAMMING CONCEPTS

INTRODUCTION

This chapter introduces you to the general programming concepts of the iPSC in order to aid you in writing applications for this concurrent machine. It is assumed that you already understand either C or FORTRAN programming, and also have a working knowledge of XENIX.

Although you may already be familiar with concurrent programming, the purpose of this chapter is to provide you with information that will help you write programs for the iPSC.

Please note that this chapter only covers basic strategies and concepts. Details of available software routines and methods of developing programs are covered in Chapters 3 through 7 of this manual.

USING THE iPSC

When using the iPSC, you are basically implementing an application (the problem to be solved) as a set of sequential processes that operate in parallel on this common problem. These processes may be either "node" processes, "cube manager" processes, or...more typically...a combination of both. A process can be defined as a C or FORTRAN program that includes system routines which cause messages to be sent and received.

Commonly, a parallel processing application requires distributing the input data to the node processes...and collecting the results. This function is performed by a host process...or processes ...running on the cube manager. In a simple application, each node process initializes, then waits for its initial data from the cube manager. When the data is received, the node process begins its computation...exchanging intermediate data with other node processes as necessary. When the computation is complete, each process returns its results to the host. Although applications usually are composed of a combination of host and node processes, there is no rule that excludes having only node processes if that is how your problem can best be solved.

The entire application (consisting of cube manager and node processes) is developed in the cube manager. The actual development steps (compiling, binding, etc.) that you use will vary depending on the programming language (C or FORTRAN) and whether the process is to be executed within the cube manager or on the nodes.

The cube manager can act as a host to the cube, creating node processes, starting the computation, and collecting the results. It lets you start and stop your application with XENIX commands.

Another way to view the cube manager is to consider it just *another node* in the cube. Although it does have access to additional facilities (such as the XENIX file system), it communicates with cube processes with similar message primitives that node processes use.

This chapter discusses how to design an application using:

- the two multitasking operating systems (XENIX and the node operating system)
- a message-based interconnection structure
- the four basic language routines for passing messages: SEND, SENDW, RECV, and RECVW
- the virtual communication links, called channels, that the four language routines require

PROGRAMMING APPROACH

The approach to programming the iPSC consists of decomposing a problem into several parts. Many applications can be broken down into a few or many independent pieces. The approach to programming the iPSC is to identify these pieces and distribute them amongst the processors. These pieces could be identical programs which work on different data sets, or they could be unique programs, each contributing its part to the final solution.

The programming approach does not require that you formulate a computation to fit on nodes and on the physical communication links that exist only between certain pairs of nodes. Instead, you can formulate the problem in terms of "processes" and "virtual" communication channels that connect all processes. This is possible because the message system automatically routes messages from any process to any other process.

In general, the programming approach for the iPSC can be summarized in two steps:

1. At a fundamental level, formulate the problem by determining the set of processes that are required and the intercommunication that is necessary. For example, an application may require a set of nine processes as represented in Figure 2-1. The lines connecting the processes represent the necessary intercommunication paths.

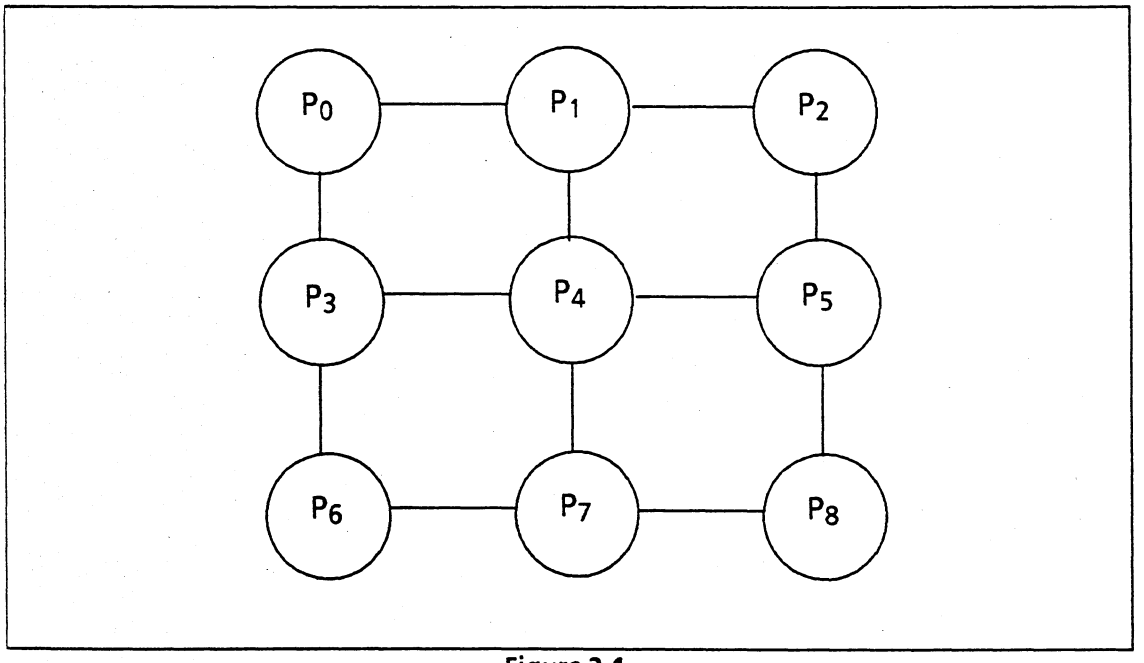

Figure 2-1
Process and Communication Formulation

2. The next step is to determine how to place these processes onto the nodes.

   Continuing the same example, Figures 2-2 and 2-3 show two possibilities for doing this. The dashed boxes represent single nodes. The first shows each process placed onto separate nodes. The second shows all nine processes placed onto a single node.

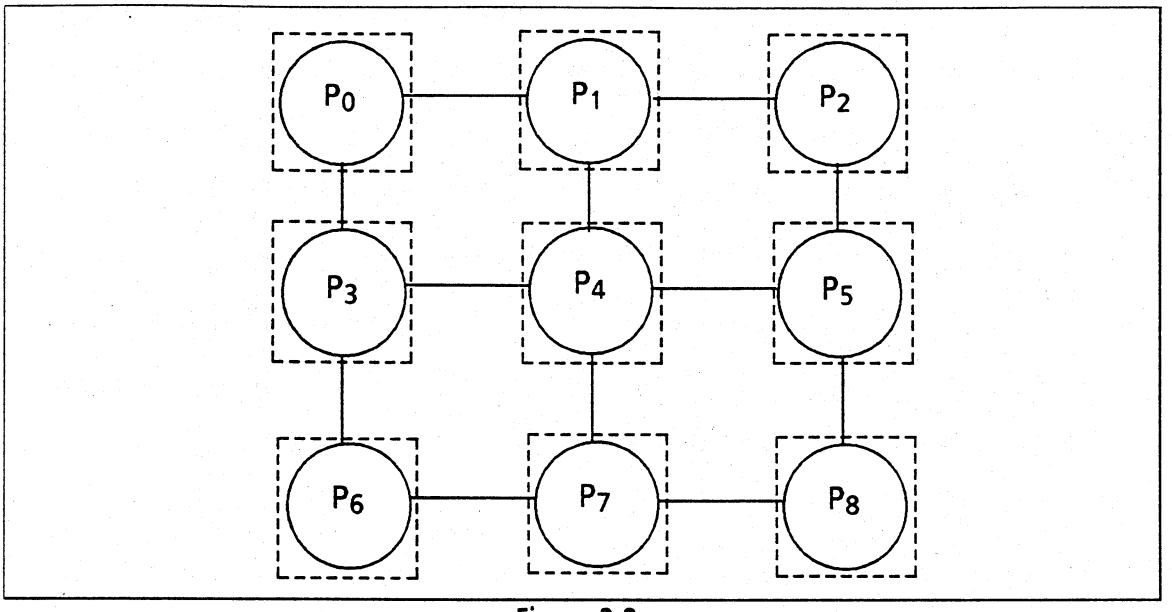

Figure 2-2
Process Placement I

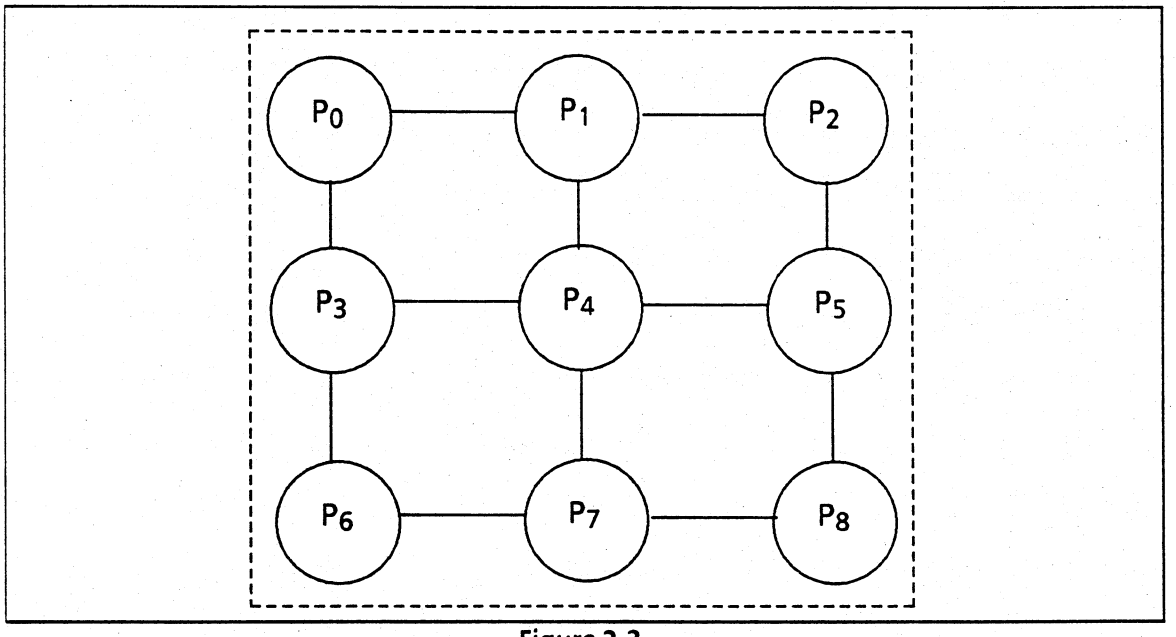

Figure 2-3
Process Placement II

As the example demonstrates, there may be more than one way to distribute the processes on the nodes. You should select a distribution that minimizes the number of messages and the distance these messages must travel. Other points to consider when selecting a distribution are keeping messages infrequent compared with computation and balancing the load so the nodes have approximately the same amount of work to do.

PROCESSES

The code for a process may be written in any combination of FORTRAN, C, or ASM286. The iPSC libraries (containing COPEN, SEND, etc.) are linked appropriately to cube manager and node processes using the XENIX link editor, **ld**. Node processes cannot perform I/O functions. If you do include I/O functions in node processes, you will receive errors when building the load modules. You can find a list of available node process functions for C in Chapter 3. If you are a FORTRAN programmer, simply avoid I/O functions (READ, WRITE, etc.) when writing node processes.

Each node process will need its own copy of every routine used.

Here are a few more points to know about processes:

Multiple Processes Per Node

Each node may contain up to 20 processes; the same number allowed in the cube manager. Therefore, the number of processes involved in a single computation may be more than the number of nodes in the system. The practical limit for the number of processes on each node is the available memory on the node and the size of the processes. User processes can take up to approximately 275 kilobytes per node in standard systems; Large Memory systems have an additional 4 megabytes . The remainder of the node's memory is reserved for the operating system and the message buffers.

Referring to Processes

The two key parameters used in the message-passing routines are process id (pid) and node id (or number). Node process id's are user-defined, and can be any number in the range of 0 to 32767. Although the system does not demand it, it is a good idea to consistently use the same pid when referring to a process.

Node numbers range from 0 to the number of nodes in the cube minus one. For example:

Cube Size	Node Number Range
16-node	0 - 15
32-node	0 - 31
64-node	0 - 63
128-node	0 - 127

The cube manager's node number is -32768. This value was chosen to allow future increases in the number of nodes per system without requiring you to modify portions of your program that refer to the cube manager.

Message Passing

Processes coordinate activities and exchange data by sending and receiving messages to and from other processes. No variables are shared among processes.

CHANNELS

A channel, as used in the iPSC system, is a 64-byte block of memory that contains information about a message to be sent or received. Examples of the type of information a channel contains are:

- source node and process ids
- destination node and process ids
- message length
- message location

A sending process needs to establish a channel to contain this information before a message can be sent. Likewise, a receiving process must also establish a channel to receive this information before the message can be received. Processes use the COPEN routine to do this. Refer to Chapters 6 and 7 for more information about this routine.

Once an operation (send or receive) has been completed, the information is no longer needed and the channel can be used again by another message. If a process needs to send/receive more than one message simultaneously, the process needs to open a channel for each of the simultaneous send/receive operations.

DEADLOCK

The term "deadlock" refers to the permanent failure of a concurrent computation to make progress. For example, a computation is deadlocked if all processes are waiting to receive a message and no one has sent one.

A similar situation can occur if processes allow too many of its messages to pile up without processing them. This will lead to congestion of the message routing and could ultimately result in deadlock.

To avoid these situations, care should be taken when designing algorithms to make sure that not all processes will block without someone sending a message and to make sure that all processes do, in fact, consume messages.

CUBE MANAGER AND NODE SYSTEM INTERFACE LIBRARIES

This section introduces you to the cube manager and node system interface libraries and explains how the two libraries differ. For the specifics about each routine refer to Chapters 6, C Routines and Chapter 7, FORTRAN Routines. Examples of the routines' usage are in Chapter 3, Developing C Programs and Chapter 4, Developing FORTRAN Programs.

The next two tables list the routines in the cube manager and node system interface libraries. The routines are the same for the two languages supported on the iPSC: C and FORTRAN. Assembly language programmers can use either library.

You will notice that the two libraries have a few routines that have the same name. For example, both libraries have a COPEN routine.

Table 2-1 (1 of 2)
Cube Manager System Interface Library

Procedure	Description
COPEN	Creates a communication channel for a cube manager process.
LOAD	Loads the file into the specified node (or all nodes if -1 is specified for the node number) and starts it.
SENDMSG	Initiates the transmission of a message from a cube manager process to a node or another cube manager process.
RECVMSG	Initiates the receipt of a message from a node or another cube manager process.
LKILL	Kills the selected process (or processes) in the selected node (or nodes).
CUBEDIM	Returns a value which indicates the dimension of the cube in which the program is running.

Table 2-1 (2 of 2)
Cube Manager System Interface Library

Procedure	Description
MYPID	Returns the process id of the calling process.
PROBEMSG	Returns a value indicating whether a message is pending for a cube manager process.
SYSLOG	Allows a cube manager process to write a message into the system log file.
LWAIT	Waits for the selected process in the selected node (or nodes) to complete. It then returns the pid, node, and completion code. Enter -1 to return as soon as the first process completes.
LWAITALL	Waits for the selected process in the selected node (or nodes) to complete. Enter -1 to wait for all processes to complete.
CCLOSE	Destroys a specified communication channel created by a previous "copen" call.

Table 2-2 (1 of 2)
Node System Interface Library

Procedure	Description
COPEN	Creates a communication channel for a process.
SEND	Initiates the transmission of a message from a node process to a cube manager or node process. The user message buffer should not be rewritten until a "status" indicates that the send operation is complete.
SENDW	Initiates the transmission of a message from a node process to a cube manager or node process. Execution of this routine causes the calling process to be blocked until the message has been sent.
RECV	Initiates the receipt of a message from another process. The user message buffer is not available for reuse and the return values are not updated until "status" is performed.
RECVW	Initiates the receipt of a message from another process. Execution of this routine causes the calling process to be blocked until the message has been received.
STATUS	Allows a node process to determine the state of the process message buffer designated for sending or receiving a message.
PROBE	Allows a node process to determine if a message of a given type is available for reception.
FLICK	Enables a node process to defer execution to another process on the node.

Table 2-2 (2 of 2)
Node System Interface Library

Procedure	Description
CUBEDIM	Returns a value equal to the dimension of the cube in which the process is running.
MYNODE	Returns a value that is the node id of the requesting process.
MYPID	Returns the process id of the calling process.
GREENLED	Allows the process to turn its node board's green LED on or off.
REDLED	Allows the process to turn its node board's red LED on or off.
SYSLOG	Allows a process to write a message into the system log file.
CLOCK	Enables a node process to determine the number of elapsed millisecond intervals since the node was initialized, modulo 2^{32}.
CCLOSE	Destroys the specified communication channel created during a previous "copen."
HANDLER	Allows a process to assign the execution of a user-written exception handler to the occurrence of a specific exception.

Differences Between the Two Libraries

The primary reason that the cube manager and node libraries are different is because the operating systems, which are the basis of the routines, are different. The cube manager operates under XENIX while the nodes have a unique operating system developed specifically for the iPSC.

The XENIX operating system dictates the way channels are treated for cube manager processes and thus the way certain cube manager, message-handling routines work. In XENIX, a channel is treated like a device. A process can only execute one device-related system routine at a time. Therefore, a cube manager process cannot perform concurrent sends and receives because that would require concurrent device operations.

Node processes, on the other hand, can perform concurrent sends and receives if the node process has multiple channels open.

The cube manager library has no analogous routine to the node library's asynchronous RECV or SEND. Consequently, the cube manager routine RECVMSG is more like the node routine RECVW than RECV. The cube manager routine SENDMSG is functionally identical to the node routine SENDW.

Difference Between SEND and SENDW; RECV and RECVW

SEND and RECV are non-blocking, asynchronous routines. When issued, they return to the calling process as soon as the node operating system records the request. The calling process continues to execute. The user message buffer should not be reused until "status" indicates that the send or receive operation is complete. "Status" indicates that the data can be reliably read. If a second send or receive request is issued before reading the data in the user message buffer, the data may be overwritten, corrupting both messages.

To explain it in terms of the routines and the channels that must be used, if a send or a receive is issued on a channel which already has a request pending, the second request is blocked until the first completes. If you issue a send or receive, and then issue yet another receive using a different channel, but the same buffer, data in the buffer will probably be overwritten. The way to avoid this is to never modify the buffer until "status" indicates that the buffer is ready for reuse.

Note that you can issue successive sends using the same buffer if your program does not change the data in the buffer.

You might use the SEND and RECV routines when the process has additional computations to complete which are not dependent on the information contained in the message. At the end of these computations, you can issue a "status" routine and further message processing can be done.

SENDW and RECVW, on the other hand, are blocking, synchronous routines. They do not return to the calling process until the node operating system has actually finished the operation and the message buffer is available for reuse. You might use these routines when the process has no other tasks it can do while the operation is being performed.

Note, however, that when the system returns from a SENDW, the message has simply been sent. It does not indicate that it has been received at the intended destination.

Using the Message Parameter "Type"

"Type" is one of the parameters in send and receive routines. Its value is user-definable, and its purpose is to identify the message content or to serve as a flag. Valid values are positive numbers in a range from 0 to 32767. The "type" parameter should not be confused with variable typing done in high-level languages.

The cube manager RECVMSG routine and the node RECVW routine receive messages differently.

On the nodes, a message is received based on its type. "Type" is an input parameter to RECV and RECVW which means that a message is received only if its "type" matches the request. That is, a pending receive on type 10 will only be fulfilled by a message sent as type 10.

On the cube manager, the "type" parameter is returned as a result of the routine. Cube manager processes cannot use message "type" to qualify message receipt, because the RECVMSG routine can be fulfilled by a message of any type. It is up to the programmer to look at the type after the message has been received in order to decide what to do with it.

Node processes, therefore, qualify message reception by "type" and cube manager processes do not.

This concept is illustrated on the next page where a common use of "type" is shown.

Suppose process A can send three different messages to process B. The only way process B knows which of the three messages it has received is by type value. Process A has defined type 10 to be an initialization message, type 15 as a floating point number, and type 20 as an error condition. Process B might then use the "probe" routine to determine the type of message that process A has sent, then branch accordingly. Or, Process B might execute three receive calls on separate channels to receive each message. Each receive specifies a different type.

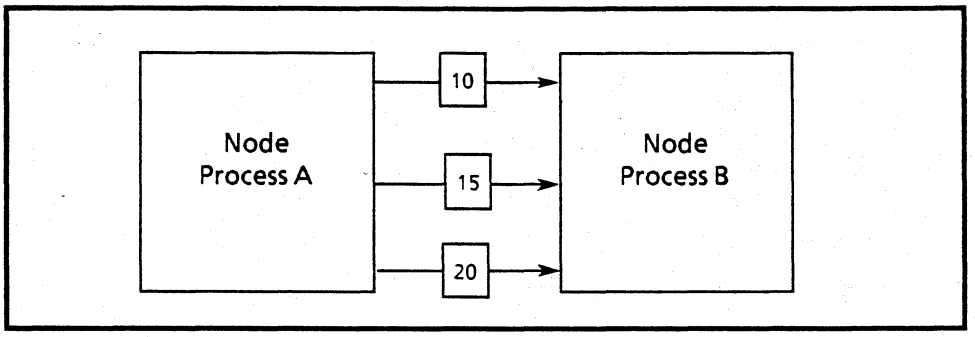

Figure 2-4
Node Processes Using "Type"

The next figure shows a cube manager process receiving messages. The cube manager process executes a RECVMSG and branches on the "type" after the message has been received. In this case, the cube manager process may execute three RECVMSG calls on three channels but cannot predict which message type will arrive at each channel.

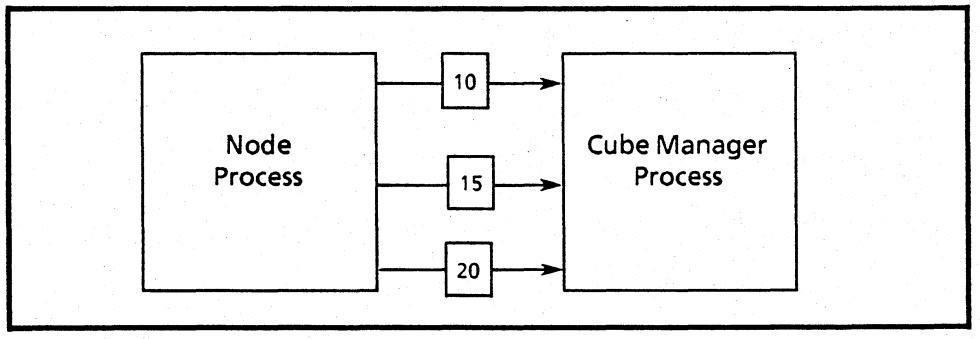

Figure 2-5
Node and Cube Manager Processes
Using "Type"

CHAPTER 3

DEVELOPING C PROGRAMS

INTRODUCTION

This chapter describes the steps for developing cube applications using the C language.

It contains the following sections:

Development Steps	Describes how to develop both cube manager and node processes. Includes a sample makefile.
The "Ring" Application	Shows a simple application composed of both cube manager and node processes demonstrating iPSC routine usage. These routines are documented in the iPSC Programmer's Reference Manual.
Huge Model Arrays	Provides information about compiling huge model arrays.
Comparison of the Math Libraries	Compares the routines in the Microsoft math library with those in the Intel math library.

NOTE

Refer to Appendix D for additional information about C that is not documented in the XENIX manuals.

DEVELOPING PROGRAMS

The following pages describe how to develop both cube manager and node programs. This is followed by a program example.

Cube Manager Processes

Once you have a source code file, preparing cube manager (host) processes for execution involves only one step...compilation. If your program calls math routines, you can use either the Intel or Microsoft math libraries. Refer to the end of this chapter for a comparison of these libraries.

These invocations assume you have a single source file named "host.c". If you have more than one module, compile each separately.

Using Intel Math Libraries:

```
cc -Alfu -o host host.c /usr/ipsc/lib/Llibcel.a \
   /usr/intel/lib/cel287.a /usr/ipsc/lib/chost.a
```

where:

-Alfu	Creates a file to conform to Intel's "large" model. Use huge model (-Alhu) if the source program contains an array that exceeds 64K bytes.
-o host	Names output file
host.c	Name of file to be compiled
/usr/ipsc/lib/Llibcel.a **/usr/intel/lib/cel287.a**	Intel math libraries
/usr/ipsc/lib/chost.a	Name of cube manager system interface library

Using Microsoft math library:

```
cc -Alfu host.c -o host -lm /usr/ipsc/lib/chost.a
```

where:

-Alfu	Creates a file to conform to Intel's "large" model. Use huge model (-Alhu) if the source program contains an array that exceeds 64K bytes.
host.c	Source file name
-o host	Names output file
-lm	Links in Microsoft's math library "Llibm.a"
/usr/ipsc/lib/chost.a	Name of the cube manager system interface library

Either compile produces a file called "host." This file can now be executed. For more information about **cc** refer to the XENIX 286 Programmer's Guide.

Node Processes

Preparing node processes to run on the cube involves two steps. These invocations assume you have a single source file named "node.c". If you have more than one module, compile each separately.

1. Compile by typing:

   cc -Alfu -K -O -c node.c

   where:

-Alfu	Creates a file to conform to Intel's "large" model. Use huge model (-Alhu) if the source program contains an array that exceeds 64K bytes.
-K	Does not include stack probes in output module
-O	Optimizes generated code
-c	Does not invoke **ld**. Stops at a ".o" file without binding.
node.c	Source code file name.

   This produces a "node.o" file that can be linked using **ld**.

Node Processes (cont.)

2. Bind by typing:

```
ld -Ml -o node /lib/Lseg.o /usr/ipsc/lib/Lcrtn0.o node.o \
   /usr/ipsc/lib/Llibcnode.a \
   /usr/ipsc/lib/Llibmnode.a
```

where:

-Ml	Creates a large model C application
-o node	Names the output module "node"
/libl/Lseg.o	Defines large model segments.
/usr/ipsc/lib/Lcrtn0.o	Is the C runtime start-up library
node.o	Name of the file to be linked. To link several modules into your application, add their names in here.
/usr/ipsc/lib/Llibcnode.a	Is the C library containing versions of malloc, free, calloc, realloc, and sbrk which get their memory directly from the node operating system.
/usr/ipsc/lib/Llibmnode.a	The Intel math libraries "Llibcel.a" and "cel287.a" may be substituted for the Microsoft math library "Llibmnode.a".

This produces an executable file named "node" that can be loaded into the cube using the "load" cube manager command or the "load" routine called by a host process. For more information about **cc** and **ld**, refer to the XENIX 286 Programmer's Guide.

Sample Makefile

The process of developing an iPSC application can be simplified and automated through the use of makefiles. Using "make" ensures that the correct build steps are carried out in the correct order. Additionally, whenever a file is modified, "make" determines the minimum number of steps required to rebuild the application.

You should be familiar with "make". It is described in the Xenix 286 Programmer's Guide (Order No. 174391).

The following example can be used to generate both cube manager (or host) and node object modules. You can adapt this makefile to your applications by substituting your program names where "host" and "node" occur.

A disk copy of this file can be found in "/usr/ipsc/examples/nx_c/ring" in a file called "makefile" (an ASCII text file). You can execute this file by typing: **make host, make node, make all,** etc., depending on what you want to do.

```
CFLAGS = -Alfu -K

#
#  NOTE: This makefile uses the default rule for the C compiler for node.c
#

both:   host node

help:
        @echo "make both or make - makes both the host and node processes"
        @echo "make host   - makes the host process"
        @echo "make node   - makes the node process"
        @echo "make clean  - cleans up"

host:   host.c
        cc -Alfu -o host host.c /usr/ipsc/lib/chost.a                    (1)


node:   node.o
        ld -Ml -o node /lib/Lseg.o /usr/ipsc/lib/LcrtnO.o  \
        node.o \
        /usr/ipsc/lib/Llibcnode.a                                        (1)

clean:
        -rm *.o
```

(1) If you are customizing this makefile for your development purposes and your program needs math libraries, you can use either Microsoft's or Intel's. Refer to the Development Steps described in the previous pages.

THE "RING" APPLICATION

This section gives an example of a C application that may be run on the iPSC. This example describes all steps involved in getting the program running. The source code files (both cube manager and node) and makefile are supplied in the "/usr/ipsc/examples/nx_c/ring" directory. This application involves passing messages around a ring as illustrated in Figure 3-1 below.

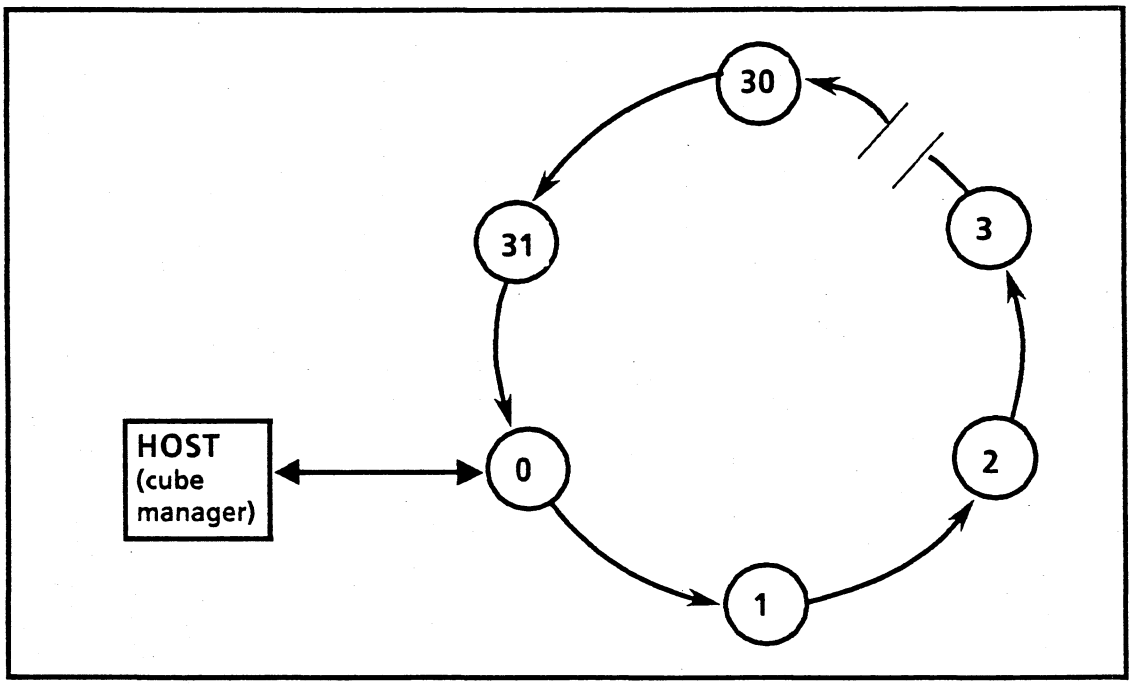

Figure 3-1
Ring Application

Although a D5 is shown, the program will work, without modification, on any cube of dimension greater than zero.

The cube manager prompts for the message length and the number of times the message is to go around the ring. Node 0 receives this information and sends a message of the desired length to the next node (1). As each subsequent node receives the message, it sends it onto the next node in the ring. Every time the message completes a round trip, node 0 reports to the cube manager the current "ring count". At the end, when the desired number of rounds have been completed, it reports the time the message spent "circling" the cube.

This example allows you to inject messages of various sizes and have them circle the cube various times without having to reload or restart the program. When you are done with the example, simply type a negative number for the desired number of ring trips and the cube manager will clean out the ring processes and return you to XENIX.

Code in the node portion of the application is somewhat more involved than that running on the cube manager. Therefore, several elements deserve comment.

Notice first that, although the functions carried out by node 0 and node n ($n \neq 0$) are not symmetric, the code is written in such a way that identical copies can be used on all nodes. However, also note that these functions are reflected in the example program's organization by having distinct pieces of the code execute on node 0 and node n. This approach has been found useful in developing and debugging application code.

Second, each node calculates the "next node" by simply adding 1 to its local node id, or wrapping around to node 0 if it has the largest node id. This method rarely selects next nodes that are nearest neighbors in the hypercube topology. Therefore, to reach them, messages have to traverse a number (< cubedim) of intermediate nodes. If you watch the node LED's when passing long messages, you can see the consequences of this.

Finally, you might be puzzled by the status check on the cube manager channel in the main loop for node 0. The reasons for this are: it protects the "count" variable from being updated (completed) before the previous count value has been received by the cube manager; and it allows node 0 to post its request to "send (...count...)" to the cube manager and not have to wait until the cube manager is ready to receive the message. Thus, node 0 goes on and sends the ring message to the next node and waits for the ring message to return from its trip. Node 0 only has to wait on the cube manager if the ring message completes its round trip before the cube manager can consume the count message.

You can compile and link the ***cube manager*** process, which is assumed to reside in a file called **host.c**, by typing:

```
cc -Alfu -o host host.c /usr/ipsc/lib/chost.a
```

This particular compile needs no math libraries and produces a file, **host**, which can now be executed on the cube manager.

Use the next two commands to prepare the ***node*** process which is assumed to reside in a file called **node.c**.

```
cc -Alfu -K -O -c node.c

ld -Ml -o node /lib/Lseg.o /usr/ipsc/lib/Lcrtn0.o node.o \
    /usr/ipsc/lib/Llibcnode.a
```

This results in a node process, called **node,** which can now be loaded into the cube.

Or, you can generate ***both*** processes using the **make** utility (copy the makefile provided in the "/usr/ipsc/examples/nx_c/ring" directory) by typing: make both or make.

The following sequence of commands should be used to run the application:

getcube	Obtain permission to use the cube.
cubelog -l mylog	Transfers logging to a file called "mylog" in your current directory.
tail -f mylog &	Invoke XENIX "tail" command to display log file entries on the screen.
load -c	Load the node operating system into the cube (the "cubeconf" file is accessed for the dimension). Only needs to be executed once. Use "loadkill" to reset the cube once NX is running.
host	Run the application. The cube manager process loads the node portion of the application using the "load" routine. It starts executing at that time.

When you are finished running the ring example, do the following:

loadkill	Clear processes out the cube.
relcube	Release the cube. This is particularly important in multi-user environments. Automatically returns logging to the default (/usr/ipsc/log/LOGFILE) log file.
kill [process id]	If "tail" is still running in the background, you should kill it using the "kill" command specifying its process id. You can list process ids using the XENIX "ps" command.

Now, if you want to change the ring example, make a copy of the file, then:

edit	Using any editor, correct the source code file(s).
make	Create new application objects.

You can now issue the first sequence of commands (starting with "getcube") to rerun the application.

This sequence of commands is similar to one you can use to run and edit/debug your own program.

The remainder of this section consists of the following:

Cube manager program	host.c
Node program	node.c

The source code files and makefile are supplied in the "/usr/ipsc/examples/nx_c/ring" directory.

At the end of the chapter are sections comparing the math libraries.

Cube Manager Code - /usr/ipsc/examples/nx_c/ring/host.c

```
/*****************************************************************************
*
* This is the Host code (in C) for the Ring demo.
*
* It prompts the user for:
*   a) the length of a message to send around a RING in the cube, and
*   b) the number of times the message is to go around the RING.
*
* It outputs:
*   a) a ring "count" each time the ring message goes past node 0,
*      and
*   b) the time it took the message to go around the ring the
*      specified number of times.
*
*****************************************************************************
*/

/*
* Define some constants:
*/
  #define NODE_PID   1
  #define HOST_PID   1

  #define ALL_NODES -1

  #define INIT_TYPE 10

  #define INIT_MSG_SIZE 4
  #define CNT_MSG_SIZE  2
  #define TIME_MSG_SIZE 4

/*
* iPSC System Functions used:
*/
  int copen();
/*
* Program variables:
*/
  int ci,type,cnt,fr_node,fr_pid;
  int msg_len;
  int i, ring_count;
  int msg_buff[2];

  long  time_buf;
  float ring_time;

  char CARRIAGE_RETURN = 13;
```

```
/*
*******************************************************************************
*/

main() {

/*******************************************************************************
*
* MAIN CODE:
*/
  printf("LOADING RING INTO CUBE ...\n");
/*
* load the cube:
*/
  load("node", ALL_NODES, NODE_PID);
/*
* Open a channel for the host-to-node-0 communications.
*/
  ci = copen(HOST_PID);
/*
* BEGIN MAIN PROGRAM LOOP:
*/
  for(;;){

     printf("************************* READY ****************************\n");
/*
*    get the number of times to go around the ring:
*/
     printf("Number of times to go around the ring (neg. value quits): ");
     scanf ("%d", &ring_count);
/*
*    If ring_count is negative break out of main loop & clean up:
*/
     if (ring_count < 0) break;
/*
*    Include ring_count in the message to the RING:
*/
     msg_buff[0] = ring_count;
/*
*    get the message length:
*/
     printf("Length of Ring message in bytes (0-16384): ");
     scanf ("%d",&msg_len);
/*
*    Include msg_len in the message to the RING:
*/
     msg_buff[1] = msg_len;
/*
*    ship the message buffer off to node 0:
*/
     sendmsg(ci, INIT_TYPE, msg_buff, INIT_MSG_SIZE, 0, NODE_PID);
```

```
/*
*    Get the current ring count from node 0 and report to user:
*/
     for (i=1;i<=ring_count;i++){
       recvmsg(ci, &type, msg_buff, CNT_MSG_SIZE, &cnt, &fr_node, &fr_pid);
       printf("Ring count: %d %c", msg_buff[0], CARRIAGE_RETURN);
     }
/*
*    Get the RING time from node 0 & report to user:
*/
     recvmsg(ci, &type, &time_buf, TIME_MSG_SIZE, &cnt, &fr_node, &fr_pid);
     ring_time = (float)time_buf/1000.00;
     printf("\nRing time : %0.2f secs.\n", ring_time);
  }

/*  END OF MAIN PROGRAM LOOP.
*********************************************************************************
/

/*********************************************************************************
*
*
* CLEAN UP TIME!
*/
  printf("CLEARING THE CUBE ...\n");
/*
* Kill RING processes in cube:
*/
  lkill(-1,-1);
  lwaitall(-1,-1);
  printf("************************** DONE ******************************\n");
}
```

Node Process Code - /usr/ipsc/examples/nx_c/ring/node.c

```
/******************************************************************************
*
* This is the NODE part of the RING example program.
*
* Node 0 will play the role of "controller" node.
*
* It waits for a message from the host telling it:
*   a) the number of times to go around the RING, and
*   b) the length of the message to send around.
*
* It then sends a message of the desired length to node 1 and
* "controls" how many times the message goes around the RING.
*
* At the end, Node 0 reports back to the Host the time it took
* the message to go around the RING.
*
*
* All the other nodes patiently wait for a message and then dutifuly
* pass it on to the next node in the RING.
*
******************************************************************************/

/*
*  Define some constants
*/
#define HOST_NID  0x8000
#define HOST_PID  1

#define INIT_TYPE  10
#define NODE_TYPE  20
#define TIME_TYPE  30
#define COUNT_TYPE 40

#define INIT_SIZE  4
#define TIME_SIZE  4
#define COUNT_SIZE 2
#define MAX_MSG_SIZE 16384

/*
*  iPSC System Calls used:
*/
   int copen(), status(), mynode(), mypid(), cubedim();
   long clock();
/*
*  Program variables:
*/
   int host_chan, node_chan;
   int i, count, ring_count;
   int msg_len;
```

```
   int msg_buff[8192];
   int my_node, my_pid, next_node, next_pid;
   int num_nodes;
   int rcnt, rnode, rpid;

   long start_time, ring_time;

/*************************************************************************
/

main() {

/*
*  Each process identifies the node its running on and its pid:
*/
   my_node = mynode();
   my_pid  = mypid();
/*
*  Each process determines the node id & and the pid of the node following
*  itself in the RING:
*/
   num_nodes = 1<<cubedim();
   next_node = (my_node + 1)% num_nodes;
   next_pid  = my_pid;

   if (my_node == 0) {
/*
*     BEGIN NODE 0 CODE:
*
*     Open channels for communicating with both the next node in the
*     RING (node 1) & the host:
*/
      node_chan = copen(my_pid);
      host_chan = copen(my_pid);
/*
*     NODE 0 MAIN LOOP:
*/
      for (;;) {

         recvw(host_chan, INIT_TYPE, msg_buff, INIT_SIZE, &rcnt, &rnode,
&rpid);

         ring_count = msg_buff[0];
         msg_len    = msg_buff[1];

         start_time = clock();

         for(i=1;i<=ring_count;i++) {

           sendw(node_chan, NODE_TYPE, msg_buff, msg_len, next_node,
next_pid);
```

```
            recvw(node_chan, NODE_TYPE, msg_buff, msg_len, &rcnt, &rnode,
&rpid);

            while (status(host_chan)) flick();

            count = i;

            send (host_chan, COUNT_TYPE, &count, COUNT_SIZE, HOST_NID,
HOST_PID);

          }

          ring_time = clock() - start_time;

          sendw(host_chan, TIME_TYPE, &ring_time, TIME_SIZE, HOST_NID,
HOST_PID);

       }

    }
    else
    {
/*
*      BEGIN OTHER NODES' CODE:
*
*      All other nodes wait for a value from their left hand neighbor,
*      and pass it to their right hand neighbor.
*
*      They only have to open one channel for communication:
*/
       node_chan = copen(my_pid);

/*
*      BEGIN OTHER NODES' MAIN LOOP:
*/
       for (;;) {

         recvw(node_chan, NODE_TYPE, msg_buff, MAX_MSG_SIZE, &rcnt, &rnode,
               &rpid);
         sendw(node_chan, NODE_TYPE, msg_buff, rcnt, next_node, next_pid);
       }
    }
}
```

HUGE MODEL ARRAYS

"Huge model" implies that data objects can be greater than 64K bytes. This section describes what you should know when using huge model arrays.

Huge Array Support

The compiler supports arrays larger than 64K bytes. The elements of the array can be either scalar objects (C data types) or composite objects (structures and unions). In fact, the term "element" may be defined to be any of the following data constructs:

- characters
- integers
- floating-point numbers
- structures
- unions

In order to avoid the high run-time cost of executing code that references items across data segments, note the following characteristic:

- Each array element must be contained within a single 64K byte segment. The only way to do this is to make the individual elements that comprise the array a power-of-2 bytes in length (see exception below). All of the C data types conform to this restriction, so it is an issue only when the array contains structures or unions. You can pad structure or union elements, as needed, to be a power of 2 in length. No individual array or union element can exceed 64K bytes in length.

  Exception: If you have a huge array whose size is larger than one segment (64K bytes), but less than two segments (128K bytes), it is not necessary that the array elements be a power-of-2 bytes in length. If they are not, the compiler will align the array in memory so that the 64K segment boundary falls between array elements.

It may be useful to know how the compiler allocates memory for arrays as described in the following two points:

- Each individual memory segment making up a huge array will be a full 64K bytes in length, with the exception of the highest-address segment, which may not need 64K bytes allocated to it because of the array size.

- All of the segments making up a huge array have contiguous LDT entries, making the array appear to be in a contiguous memory space to the executing code. In the iPSC nodes, the segments are also continguous in physical memory.

Compiling In Huge Model

Except for array handling, huge model programs are the same as large model programs. If the source program (either cube manager or node) contains an array that exceeds 64K bytes, compile it in the huge model using the **-Alhu** option (instead of the usual -Alfu).

An example for node processes is:

```
cc -Alhu -K -O -c xxx.c
```

Invoke ld for node processes the usual way using **-Ml** option as described earlier in this chapter. Using the huge model may cause undesirable inefficiencies that are addressed in the next section.

Using The "Huge" Keyword

You do not necessarily have to use huge model when compiling a program that contains one or more arrays whose size exceeds 64K bytes. Huge model compilation generates extra code to access arrays, resulting in a larger program and slower execution. Arrays that exceed 64K require this extra code, smaller arrays do not. When you compile a program in the huge model, the extra code is generated for all accesses to *all* arrays, regardless of their size. To eliminate the extra code and execution overhead in the cases where it is not needed, use the ***huge*** keyword in the declaration of arrays that exceed 64K and compile in ***large*** model.

An example of using the huge keyword is:

```
long huge user_array[20000];
```

Note that the keyword "huge" must come ***before*** the variable name in the code statement. Another use of the "huge" keyword is when you are declaring a pointer to a huge array. For example:

```
int huge *abc
```

declares pointer abc as a pointer to a huge integer array.

You must then direct the compiler to process keywords using the -Me command line option. To compile in the large model and enable the huge keyword, use both the -Alfu and -Me option strings. For example:

```
cc -Alfu -Me -K -o -c xxx.c
```

"SIZEOF" Operator

The "sizeof" operator returns an *int* value, which is not large enough to contain the size of some huge arrays. To work around this problem, the result of a "sizeof" can be cast into a *long int* to produce a correct result, for example:

```
array_size = (long)sizeof(huge_array);
```

The variable array_size must be a long integer in order to contain the result.

Difference of Pointers

When two pointers to a huge array are subtracted the result may not fit into an *int*, which is the default type for this operation. The solution is similar to that for the "sizeof" operator, in that the result can be cast into a *long int*:

```
diff = (long)(huge_ptr_1 - huge_ptr_2);
```

COMPARISON OF THE MATH LIBRARIES

Table 3-1 lists the mapping between /lib/Llibm.a (Microsoft's math library) and /usr/intel/lib/cel287.a (the Intel math library). Use the C-style name when calling a procedure in cel287.a. For example, call **sin** instead of **mqersin**. The /usr/ipsc/lib/Llibcel.a library makes the mapping between the two.

NOTE

If you are using the cel287.a library, use the header file, /usr/include/cel287.h instead of math.h for external declarations.

Only those calls in the Llibm.a library that have a corresponding call in the cel287.a are supported. For example, **atof** cannot be used. Refer to the XENIX 286 C Library Guide for calling sequences.

Table 3-1
Comparison of Llibm.a With cel287.a

Llibm.a	cel287.a	Description
acos	mqeracs	arc cosine
asin	mqerasn	arc sine
atan	mqeratn	arc tangent
atan2	mqerat2	arc tangent (y/x)
atof		convert ascii to float
ceil		smallest integer not <x
cos	mqercos	cosine
cosh	mqercsh	hyperbolic cosine
exp	mqerexp	exponential function
fabs*		absolute value of float
floor		smallest integer not >x
fmod	mqermod	modulus (remainder)
gamma		log gamma function
hypot		euclidean distance
j0		bessel function
j1		bessel function
jn		bessel function
ldexp		x times 2 to the y
log	mqerlge	natural log
log10	mqerlgd	common log
pow	mqery2x	y raised to the power x
sin	mqersin	sine
sinh	mqersnh	hyperbolic sine
sqrt*		square root
tan	mqertan	tangent
tanh	mqertnh	hyperbolic tangent
y0		bessel function
y1		bessel function
yn		bessel function

*These calls are not in the cel287.a library, but ***are*** supported directly by the Llibcel.a interface library. Refer to Appendix B for the calling sequence and description.

Table 3-2
Library Functions in cel287.a But Not in Llibm.a

For the C calling convention for these calls, refer to Appendix B. For more information about these functions, refer to Intel 80287 Support Library Reference Manual. You call the procedure using the name in /usr/ipsc/lib/Llibcel.a which in turn calls the corresponding procedure in /usr/intel/lib/cel287.a.

Llibcel.a	cel287.a	Description
dim	mqerdim	positive difference
ia2	mqeria2	round to nearest integer
ia4	mqeria4	round to nearest integer
iax	mqeriax	round to nearest integer
ic2	mqeric2	truncate to integer
ic4	mqeric4	truncate to integer
icx	mqericx	truncate to integer
ie2	mqerie2	round to even integer
ie4	mqerie4	round to even integer
iex	mqeriex	round to even integer
max	mqermax	largest value
min	mqermin	smallest value
rmd	mqerrmd	remainder, close to 0
sgn	mqersgn	sign transfer
yis	mqeryis	y to an integer power

CONTENTS OF Llibcnode.a

The library **/usr/ipsc/lib/Llibcnode.a** is intended for use by node processes, and is a functional subset of the standard XENIX library **/lib/Llibc.a**. Those procedures contained in **/usr/ipsc/lib/Llibcnode.a** are listed below. Procedures that perform disk or terminal I/O, such as *fopen* and *printf*, and those that make XENIX system calls, such as *system*, are not supported by this library.

abs	integer absolute value
atoi	convert ascii to integer
atol	convert ascii to long
bsearch	binary search
calloc	memory management
ecvt	numeric output conversion
exit	terminate a process
fcvt	numeric output conversion
free	memory management
gcvt	numeric output conversion
isalnum	classify characters
isalpha	classify characters
isascii	classify characters
iscntrl	classify characters
isdigit	classify characters
isgraph	classify characters
islower	classify characters
isprint	classify characters
ispunct	classify characters
isspace	classify characters
isupper	classify characters
isxdigit	classify characters
lsearch	linear search
malloc	memory management
qsort	sort
rand	generate a random number
realloc	memory management
sbrk	memory management
sprintf	formatted write to a string
srand	seed random number generator
sscanf	convert and format input from a string
strcat	string operations
strchr	string operations
strcmp	string operations
strcpy	string operations
strcspn	string operations
strlen	string operations
strncat	string operations
strncmp	string operations
strncpy	string operations
strpbrk	string operations
strrchr	string operations
strspn	string operations
strtok	string operations
toascii	translate characters
tolower	translate characters
toupper	translate characters

CHAPTER 4

DEVELOPING
FORTRAN PROGRAMS

INTRODUCTION

This chapter describes the steps for developing cube applications using the FORTRAN language.

It contains the following sections:

Development Steps	Describes how to develop both cube manager and node processes. Includes a sample makefile.
The "Ring" Application	Shows a simple application composed of both cube manager and node processes demonstrating iPSC routine usage. These routines are documented in the iPSC Programmer's Reference Manual.

DEVELOPING PROGRAMS

The following pages describe how to develop both cube manager and node programs. This is followed by a program example.

Cube Manager Processes

Once you have a source code file, preparing cube manager (host) processes for execution involves two steps. These invocations assume you have a single source file named "host.for". If you have more than one module, compile each separately.

1. Compile by typing:

   rmfort host.for

   where:

host.for	Name of source file

   The rmfort compiler has several switches available. None are mandatory, but several are useful. Refer to the Ryan-McFarland FORTRAN manual for more information.

   This produces a "host.o" file.

2. Link by typing:

   ```
   cc -Ml -o host host.o /lib/Llibf.a \
       /usr/ipsc/lib/rmfhost.a
   ```

   where:

-Ml	Creates a file to conform to Intel's "large" model
-o host	Names executable file "host"
host.o	Name of file to be linked. To link several modules into your application, add their names in here.
/lib/Llibf.a	Runtime library
/usr/ipsc/lib/rmfhost.a	Ryan-McFarland cube manager library

   This produces a file called "host" which can now be executed. For more information about cc, refer to the XENIX 286 Programmer's Guide.

Node Processes

Preparing node processes to run on the cube involves two steps. These invocations assume you have a single source file named "node.for". If you have more than one module, compile each separately.

1. Compile by typing:

   rmfort node.for

   where:

   node.for Source code file name

   The rmfort compiler has several switches available. None are mandatory, but several are useful. Refer to the Ryan-McFarland FORTRAN manual for more information.

   This produces a "node.o" file that can be linked using ld.

Node Processes (cont.)

2. Bind by typing:

```
ld -Ml -o node \
   /lib/Lseg.o \
   /usr/ipsc/lib/Lcrtn0.o \
   node.o \
   /usr/ipsc/lib/Llibfnode.a \
   /usr/ipsc/lib/Llibcnode.a
```

where:

-Ml	Creates a large model C application
-o node	Names the executable module "node"
/lib/Lseg.o	Defines large model segments
/usr/ipsc/lib/Lcrtn0.o	Start-up library containing runtime initialization routines for node processes
node.o	Name of the file to be linked. To link several modules into your application, add their names in here.
/usr/ipsc/lib/Llibfnode.a	Ryan-McFarland node operating system runtime library
/usr/ipsc/lib/Llibcnode.a	C library for dynamic loading. Contains modules used by Llibfnode.a

This produces an executable file named "node" that can be loaded into the cube using the "load" cube manager command or the "load" routine called by a host process. For more information about **ld**, refer to the XENIX 286 Programmer's Guide.

Sample Makefile

The process of developing an iPSC application can be simplified and automated through the use of makefiles. Using "make" ensures that the correct build steps are carried out in the correct order. Additionally, whenever a file is modified, "make" determines the minimum number of steps required to rebuild the application.

You should be familiar with "make". It is described the Xenix 286 Programmer's Guide (Order No. 174391).

The following example can be used to generate both cube manager (or host) and node object modules. You can adapt this makefile to your applications by substituting your program names where "host" and "node" occur.

RM FORTRAN requires a filename extension of either ".for" or ".f". You cannot use ".ftn".

A disk copy of this file can be found in "/usr/ipsc/examples/nx_rmf/ring" in a file called "makefile" (an ASCII text file). You can execute this file by typing: **make host, make node, make both**, etc., depending on what you want to do.

```
.SUFFIXES: .for .o

.for.o :
     @echo rmfort -bhe $<
     @sh -c "rmfort -bhe $< 2>&1 | tee /dev/tty | grep '0 Errors' > /dev/null
#
#  note:  'b and h' switches are for contiguous huge array support
#         'e' is to list errors
#
#  These are commonly used switches. See the RM/FORTRAN User's Guide
#  for more details on all compiler switches.
#
#  The rule to go from .for to .o files is designed to allow make to
#  continue when compiler warnings are issued, but stop on compiler errors.
#  RM/FORTRAN exits with a non-zero status when warnings or errors are
#  issued, and this non-zero status causes make to stop.  It is not always
#  feasible to modify an RM/FORTRAN source program to eliminate the warning
#  messages.  The second line of the rule invokes the compiler, piping its
#  screen output to the grep utility, and to the terminal screen.  The grep
#  utility looks for the string '0 Errors', and it is grep's exit status that
#  causes make to stop or continue on.  grep's output is sent to the null
#  device.  The first line of the rule simply echoes the compile command to
#  the terminal screen in the same display format as if the compiler had been
#  invoked in a more straightforward way.

both : host node

help :
        @echo "make both or make  - makes both the host and node processes"
        @echo "make host    - makes the host process"
        @echo "make node    - makes the node process"
        @echo "make clean   - cleans up"

host:           host.o
                cc -Ml -o host  host.o \
                /lib/Llibf.a \
                /usr/ipsc/lib/rmfhost.a

node:           node.o
                ld -Ml -o node \
                /lib/Lseg.o \
                /usr/ipsc/lib/LcrtnO.o \
                node.o \
                /usr/ipsc/lib/Llibfnode.a \
                /usr/ipsc/lib/Llibcnode.a

clean:
                -rm *.o
```

THE "RING" APPLICATION

This section gives an example of a FORTRAN application that may be run on the iPSC. This example describes all steps involved in getting the program running. The source code files (both cube manager and node) and makefile are supplied in the "/usr/ipsc/examples/nx_rmf/ring" directory. This application involves passing messages around a ring as illustrated in Figure 4-1 below.

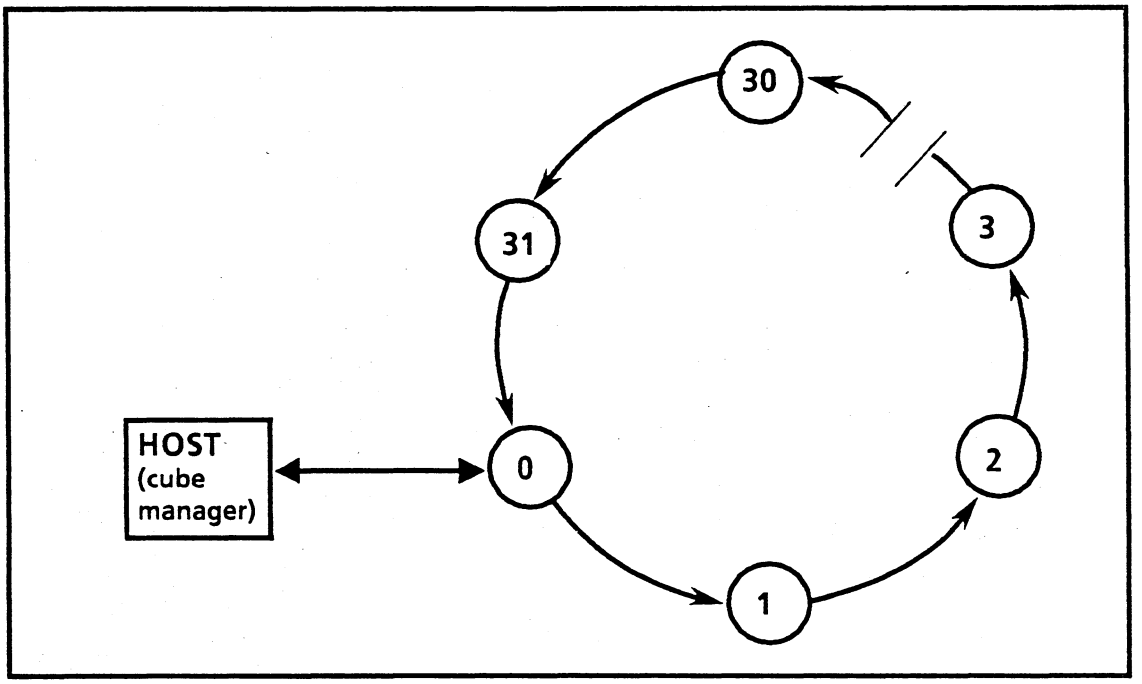

Figure 4-1
Ring Application

Although a D5 is shown, the program will work, without modification, on any cube of dimension greater than zero.

The cube manager prompts for the message length and the number of times the message is to go around the ring. Node 0 receives this information and sends a message of the desired length to the next node (1). As each subsequent node receives the message, it sends it onto the next node in the ring. Every time the message completes a round trip, node 0 reports to the cube manager the current "ring count". At the end, when the desired number of rounds have been completed, it reports the time the message spent "circling" the cube.

This example allows you to inject messages of various sizes and have them circle the cube various times without having to reload or restart the program. When you are done with the example, simply type a negative number for the desired number of ring trips and the cube manager will clean out the ring processes and return you to XENIX.

Code in the node portion of the application is somewhat more involved than that running on the cube manager. Therefore, several elements deserve comment.

Notice first that, although the functions carried out by node 0 and node n ($n \neq 0$) are not symmetric, the code is written in such a way that identical copies can be used on all nodes. However, also note that these functions are reflected in the example program's organization by having distinct pieces of the code execute on node 0 and node n. This approach has been found useful in developing and debugging application code.

Second, each node calculates the "next node" by simply adding 1 to its local node id, or wrapping around to node 0 if it has the largest node id. This method rarely selects next nodes that are nearest neighbors in the hypercube topology. Therefore, to reach them, messages have to traverse a number (< cubedim) of intermediate nodes. If you watch the node LED's when passing long messages, you can see the consequences of this.

Finally, you might be puzzled by the status check on the cube manager channel in the main loop for node 0. There are two reasons for this. First, it protects the "count" variable from being updated (corrupted) before the previous count value has been received by the cube manager. Second, it allows node 0 to post its request to "send (...count...)" and not have to wait until the cube manager is ready to receive the message. Thus, node 0 goes on and sends the ring message to the next node and waits for the ring message to return from its trip. Node 0 only has to wait on the cube manager if the ring message completes its round trip before the cube manager can consume the count message.

You can compile and link the ***cube manager*** process, which is assumed to reside in a file called **host.for**, by typing:

```
rmfort  host.for

cc -Ml  -o  host  host.o  \
    /lib/Llibf.a  \
    /usr/ipsc/lib/rmfhost.a
```

This produces a file, **host**, which can be executed on the cube manager.

The next two commands prepare the ***node*** process, which is assumed to reside in a file called **node.for**.

```
rmfort   node.for

ld -Ml -o node \
   /lib/Lseg.o \
   /usr/ipsc/lib/Lcrtn0.o  \
   node.o \
   /usr/ipsc/lib/Llibfnode.a  \
   /usr/ipsc/lib/Llibcnode.a
```

This results in a node process, called **node,** which can be loaded into the cube.

Or, you can generate ***both*** processes using the **make** utility (copy the makefile provided in the "/usr/ipsc/examples/nx_rmf/ring" directory) by typing: make both or make.

The following sequence of commands should be used to run the application:

getcube	Obtain permission to use the cube.
cubelog -l mylog	Transfer logging to a file called "mylog" in your current directory.
tail -f mylog &	Invoke XENIX "tail" command to display log file entries on the screen.
load -c	Load NX into the cube (the "cubeconf" file is accessed for the dimension). Only needs to be executed once. Use "loadkill" to reset the cube once NX is running.
host	Run the application. The cube manager process loads the node portion of the application using the "load" routine. It starts executing at that time.

When you are finished running the ring example, do the following:

loadkill	Clear processes out of the cube.
relcube	Release the cube. This is particularly important in multi-user environments. Automatically returns logging to the default (/usr/ipsc/log/LOGFILE) log file.
kill [process id]	If "tail" is still running in the background, you should kill it using the "kill" command specifying its process id. You can list process ids using the XENIX "ps" command.

Now, if you want to change the ring example, make a copy of the file, then:

edit	Using any editor, correct the source code file(s).
make	Create new application objects.

You can now issue the first sequence of commands (starting with "getcube") to rerun the application.

This sequence of commands is similar to one you can use to run and edit/debug your own program.

The remainder of this section consists of the following:

Cube manager program	host.for
Node program	node.for

The source code files and makefile are supplied in the "/usr/ipsc/examples/nx_rmf/ring" directory.

At the end of the chapter are instructions for converting FTN286 to RM FORTRAN.

Cube Manager Code - /usr/ipsc/examples/nx_rmf/ring/host.for

```
c*******************************************************************************

        program host

c*******************************************************************************
c
c       This is the Host code (in Fortran) for the Ring demo.
c
c       It prompts the user for:
c         a) the length of a message to send around a RING in the cube, and
c         b) the number of times the message is to go around the RING.
c
c       It outputs:
c         a) a ring "count" each time the ring message goes past node 0,
c            and
c         b) the time it took the message to go around the ring the
c            specified number of times.
c
c*******************************************************************************

c*******************************************************************************
c
c       DECLARATIONS:
c
c
c       Declare & initialize CONSTANTS:
*
        integer*4 NODEPID
        integer*4 HOSTPID
        integer*4 ALLNODES
        integer*4 INITTYPE
        integer*4 TIMEMSGSIZE
        integer*4 CNTMSGSIZE
        integer*4 INITMSGSIZE

c
c       Declare iPSC System Functions used:
c
        integer*4 copen

c
c       Declare program variables:
c
        integer*4 ci,type,cnt,frnode,frpid
        integer*4 msglen
        integer*4 ringcount
        integer*4 msgbuff(2)
```

```
c       Declare time variable:
c
        real*4    ringtime

        DATA NODEPID            /1/
        DATA HOSTPID            /1/
        DATA ALLNODES           /-1/
        DATA INITTYPE           /10/
        DATA TIMEMSGSIZE        /4/
        DATA CNTMSGSIZE /4/
        DATA INITMSGSIZE        /8/

c
c*******************************************************************************

c*******************************************************************************
c
c       MAIN CODE:
c
c
        write (6,51)
51      format(' LOADING RING INTO CUBE ...')

c       load the cube:

        call load('node', ALLNODES, NODEPID)

c       Open a channel for the host-to-node-0 communications.

        ci = copen(HOSTPID)

c*******************************************************************************
c
c       BEGIN MAIN PROGRAM LOOP:
c

10        write (6,100)
100       format(' ************************* READY ********************
     1********')

c         get the number of times to go around the ring:

          write(6,101)
101       format(' Number of times to go around the ring (neg. value quit
     1s): ')
          read(5,102) ringcount
102       format(i7)
c       ringcount = 1

c         If ringcount is negative exit HOST program:
c
```

```
          if (ringcount .lt. 0) goto 600

c         Include ringcount in the message to the RING:

          msgbuff(1) = ringcount

c         get the message length:

          write (6,201)
201       format(' Length of Ring message in bytes (0-16384): ')
          read (5, 202) msglen
202       format(i5)
c       msglen = 2

c         Include msglen in the message to the RING:

          msgbuff(2) = msglen

c         ship the message buffer off to node 0:

          call sendmsg(ci, INITTYPE, msgbuff, INITMSGSIZE, 0, NODEPID)

c         Get the current ring count from node 0 and report to user:

          do 400 i=1, ringcount

      call recvmsg(ci, type, msgbuff, CNTMSGSIZE, cnt, frnode, frpid)

            write (6,310) msgbuff(1)
310         format('+Ring count: ',i5)

400     continue

c         Get the RING time from node 0 & report to user:

      call recvmsg(ci, type, msgbuff, TIMEMSGSIZE, cnt, frnode, frpid)

          ringtime = real(msgbuff(1))/1000.00

          write (6,306) ringtime
306       format(/,' Ring time :',F9.2,' secs.')

          goto 10

c
c         END OF MAIN PROGRAM LOOP.
c
c*******************************************************************************

c*******************************************************************************
c
```

```
c       CLEAN UP TIME!
c

600     write (6,601)
601     format(' CLEARING THE CUBE ...')

c       Kill RING processes in cube

        call lkill(-1,-1)
        call lwaitall(-1,-1)

        write (6,701)
701     format(' ************************** DONE ************************
     1******')

        end
c
c*******************************************************************************
```

Node Process Code - /usr/ipsc/examples/nx_rmf/ring/node.for

```
*******************************************************************************

        program node

*******************************************************************************
*
*       This is the NODE part of the RING demo Program.
*
*       Node 0 will play the role of "controller" node.
*
*       It waits for a message from the host telling it:
*         a) the number of times to go around the RING, and
*         b) the length of the message to send around.
*
*       It then sends a message of the desired length to node 1 and
*       "controls" how many times the message goes around the RING.
*
*       At the end, Node 0 reports back to the Host the time it took
*       the message to go around the RING.
*
*
*       All the other nodes wait for a message and then
*       pass it on to the next node in the RING.
*
*******************************************************************************

*******************************************************************************
*
*       DECLARATIONS:
*
*
c       Program CONSTANTS:

        integer*4 HOSTNID
        integer*4 HOSTPID

        integer*4 INITTYPE
        integer*4 NODETYPE
        integer*4 TIMETYPE
        integer*4 COUNTTYPE

        integer*4 INITSIZE
        integer*4 TIMESIZE
        integer*4 COUNTSIZE
        integer*4 MAXMSGSIZE

        integer*4 NOTBUSY
```

```
c       iPSC System Calls used:

        integer*4 copen, status, mynode, mypid, cubedim
        integer*4 clock

c       Program variables:

        integer*4 hostchan, nodechan
        integer*4 i, count, ringcount
        integer*4 msglen
        integer*4 nextnode, nextpid
        integer*4 msgbuff(4096)
        integer*4 ownnode, ownpid
        integer*4 numnodes
        integer*4 rtype, rcnt, rnode, rpid

c       Timing variables:

        integer*4 starttime, ringtime

        data HOSTNID   /-32768/
        data HOSTPID    /1/

        data INITTYPE   /10/
        data NODETYPE   /20/
        data TIMETYPE   /30/
        data COUNTTYPE /40/

        data INITSIZE     /8/
        data TIMESIZE     /4/
        data COUNTSIZE    /4/
        data MAXMSGSIZE /16384/

        data NOTBUSY    /0/

*******************************************************************************
*
*       MAIN CODE:
*
*
c       Each process identifies the node its running on and its pid:

        ownnode = mynode()
        ownpid  = mypid()

c       Each process determines the node id & and the pid of the next node
c       in the RING:

        numnodes = 2**cubedim()
        nextnode = mod(ownnode + 1, numnodes)
```

```
        nextpid  = ownpid

        if(ownnode.eq.0) then

*****************************************************************************
*
*         BEGIN NODE 0 CODE:
*
c         Open channels for communicating with both the next node in the
c         RING (node 1) & the host:

          nodechan = copen(ownpid)
          hostchan = copen(ownpid)

*****************************************************************************
*
*         NODE 0 MAIN LOOP:
*

10        call recvw(hostchan, INITTYPE, msgbuff, INITSIZE, rcnt, rnode,
     >       rpid)

          ringcount = msgbuff(1)
          msglen    = msgbuff(2)

          starttime = clock()

          do 400 i=1,ringcount
               call sendw(nodechan, NODETYPE, msgbuff, msglen, nextnode,
     >              nextpid)

               call recvw(nodechan, NODETYPE, msgbuff, msglen, rcnt, rnode,
     >              rpid)

c              As soon as the host channel is not busy report the current
c              count to the HOST:

200            if (status(hostchan).eq.NOTBUSY) goto 300
                  call flick()
                  goto 200
300            continue

               count = i

               call send (hostchan, COUNTTYPE, count, COUNTSIZE, HOSTNID,
     >              HOSTPID)
```

```
400       continue

          ringtime = clock() - starttime

          call sendw(hostchan, TIMETYPE, ringtime, TIMESIZE, HOSTNID,
     >             HOSTPID)

          goto 10
*
*         END NODE 0 MAIN LOOP.
*
*******************************************************************************
*
*       END OF NODE 0 CODE.
*
*******************************************************************************

        else

*******************************************************************************
*
*       BEGIN OTHER NODES' CODE:
*
c       All other nodes wait for a value from their left hand neighbor,
c       and pass it to their right hand neighbor.
c
c       They only have to open one channel for communication:

        nodechan = copen(ownpid)

*******************************************************************************
*
*
*       BEGIN OTHER NODES' MAIN LOOP:
*

20        call recvw(nodechan, NODETYPE, msgbuff, MAXMSGSIZE, rcnt,
     >      rnode, rpid)
          call sendw(nodechan, NODETYPE, msgbuff, rcnt, nextnode,
     >      nextpid)

          goto 20
```

```
*
*       END OTHERS' MAIN LOOP.
*
*******************************************************************************
*
*       END OTHERS' CODE.
*
*******************************************************************************

      endif

      end
*
*     ... OF PROGRAM CODE.
*
*******************************************************************************
```

CHAPTER 5

CUBE MANAGER COMMANDS

INTRODUCTION

Cube manager commands are invoked from a terminal and allow you to perform such tasks as accessing the cube, loading an application, and executing an application.

This chapter serves as a reference section for the cube manager commands. A summary of how they are normally used is given at the end of the chapter, as well as in Chapter 3, "Developing C Programs" or Chapter 4, "Developing FORTRAN Programs" in the iPSC Program Development Guide.

NOTE

Some commands use parameters from the cube configuration file. Refer to the iPSC System Admintrator's Guide for complete details about the cube configuration file.

The cube manager commands are XENIX programs that may be invoked from a standard XENIX shell or executed from a file along with other commands using the "exec" system function. Refer to XENIX 286 C Library Guide for more information about "exec".

Each command's explanation is divided into four parts:

- a description of what the command does
- the invocation sequence
- a description of each argument. Arguments are shown in italic print and must be entered in the given order. Optional arguments are in brackets. As with any XENIX command, you must press RETURN after typing the command from the terminal.
- errors that can occur through the use of this command

USING CUBE MANAGER COMMANDS IN SHELL SCRIPTS

If you are writing shell scripts using cube manager commands, one of the following values is returned to the shell variable:

0 = successful execution

1 = an error has occurred (errors are documented for each command in this chapter)

2 = you do not have access to the cube (you must execute "getcube")

The shell variable name varies depending on which shell you use. For the C shell, it is "$status". For the Bourne shell, it is "$?". To display the contents of these variables, use the XENIX command "echo". You would type:

echo $status	(if you are using the C shell)
echo $?	(if you are using the Bourne shell)

CUBEINIT

This command initializes the cube without loading the node operating system. Since the "load" command always uses cubeinit to initialize the cube before loading the operating system, you will rarely need to use this command. However, in the event that "load -c -R" does not work, use "cubeinit". This command is useful for quickly isolating a problem with the cube, such as a disconnected cable.

Invocation Sequence

cubeinit ***[-I] [-c] [-n node] [-s 1] [-Q file] [-R] dim***

Arguments

-I	Reset the nodes using the RS422/USM link rather than using the global link. Use this option only when you have difficulty invoking "cubeinit" with the **-c** or **-R** switch.
-c	Specifies a coldstart be performed. This puts the cube in a condition the same as having just been powered on.
-n node	Specifies one node, by number, to initialize. The valid range is 0 to $2^{cubedim}$ -1.
-s 1	Use only if initializing a Large Memory or Vector Processor system. Specifies the number of slots to skip when initializing. Initialization should only be done to node boards. Currently, only the value "1" is used to skip memory boards or vector processor boards in the LBX (odd-numbered) slots. If you do not use this switch, all boards are initialized.
-Q file	Specifies an alternative pathname for the cube configuration file. The default filename is "/usr/ipsc/conf/cubeconf".
-R	Performs the same type of coldstart on the cube as **-c**, however, it also resets the 310's i186/51 board and its associated software driver.
dim	Required switch. Specifies the dimension of the cube you wish to initialize. "dim" is the total number of boards, including any optional boards such as memory or vector boards. For example, a D4MX has 32 boards, so dimension is 5. This value is not automatically read from the cube configuration (cubeconf) file.

CUBEINIT (cont.)

Errors

Cubeinit: timeout on serial channel	The cube is off, or the RS422 connection is broken, or a USM is not working.
Initialization receive timeout, node: *n*	If all nodes are listed (in separate messages), the global Ethernet channel is not working. If only one or a few nodes are listed, those nodes specified are not working.

CUBELOG

Allows you to perform various manipulations on the system log file. For example, you may want to empty the contents of the system log file or change the name of the log file.

Invocation Sequence

cubelog *[{-n | -q | -l filename}]*

Arguments

-n	Specifies that you do not want the system to maintain the log any longer. If you want to start logging significant events again, you can type "cubelog" with no arguments. The default log file name will be used.
-q	Queries the system to find out the pathname of the file currently being logged to. The pathname is displayed on the terminal that issued the command. Note that you cannot assign "cubelog -q" to a variable in a shell script and have the returned pathname be assigned to that variable.
-l filename	Specifies that you want the system to maintain the log named "filename". If the log file name is omitted, the system reopens the log using the default name of /usr/ipsc/log/LOGFILE. When you are finished running the cube and execute "relcube", "relcube" automatically invokes "cubelog" so that logging is returned to the default log file, /usr/ipsc/log/LOGFILE.

NOTE

Before you remove (rm) a log file or move (mv) a log file, you must stop logging messages to it. You can stop logging completely by typing "cubelog -n", or you can start logging to another file by typing "cubelog -l newfile". For example, you would issue this sequence:

```
cubelog -n
rm LOGFILE
cubelog
```

to stop logging, remove LOGFILE, and then start a new LOGFILE.

Errors

Usage: cubelog [{-n | -q | -l filename}]	Switches on "cubelog" were used incorrectly.
Pathname of logfile is too long	The log file name is longer than 128 or the entire pathname is longer than 256 characters.
Can't open file: <filename>	File does not exist or permissions not set correctly.

GETCUBE

Allows you to gain exclusive access to the cube. Invoking this command guarantees that other cube users cannot access the cube until the "exclusive" user executes the "relcube" command.

After invoking this command, the message "assigning cube to <user ID>" is displayed if you are gaining access to the cube and an entry is made in the system log file. If you or someone else already has access, the message "cube assigned to <user ID>" is displayed.

The only way to release the cube is to invoke "relcube". Neither logging out, nor rebooting XENIX will clear the ownership of the cube. "relcube" can be executed by the current owner of the cube or by the super user (root). Super user's instructions are in the iPSC System Administrator's Guide.

Invocation Sequence

getcube

Arguments

None.

Errors

Usage: getcube	Try invoking the command again. If problem persists, execute "shutdown" and reboot XENIX.
Please wait	The program is trying to access /etc/cubetab. If it cannot, the program aborts. Try running /etc/fixperm. If problem persists, remove the "/dev/asglock" and "/etc/cubetab" files.
getcube: Can't set lock	Try running /etc/fixperm.
getcube: Can't create control file	Try running /etc/fixperm.
getcube: Can't read /dev	Try running /etc/fixperm.
getcube: No tty	Execute "shutdown" and reboot XENIX.
getcube: name too long	Someone has tampered with the "getcube" code. Recopy it from floppy or tape.

LOAD

Loads a program into the specified node or nodes and starts it. Use this command to load the node operating system (using **-c**) and/or application process files.

Invocation Sequence

load *[-c [dim]] [-R] [-b buffers] [[-p pid] [-H] [-S stack] [-F free] [-T time] [-M[c][d]] [-Q file] [-K file] [-L file] [-V file] [-X file] [node [node...]] file]*

Arguments

-c dim	cold start the cube (a condition the same as having just powered it on). The cube will be reset and reloaded with a new copy of NX. "dim" is the cube dimension. If "dim" is not specified, the value assigned to the keyword "dimension" in the cube configuration file ("cubeconf") is used.
-R	does a hard reset of the i186/51 board in the 310. It performs the same reset as would occur if you pressed the RESET button on the 310, however, this switch allows you to do the reset from a remote location. Use this switch if you get "cNA960 Unresponsive" errors as a result of invoking "load -c". This reset takes longer than **-c**, but is more effective.
-b buffers	Buffers are used to hold messages destined for the node or being routed to another node. If this switch is not used, 100 buffers are allocated. You can only change this number when no user processes are running in the node. The **-b** switch deletes any loaded processes and restarts the system (similar to **-c**). The maximum number of buffers is 300. If you allocate the maximum number and your application still requires more, you must modify your program and reduce the numbers of message transmissions occurring at any one time.
-p pid	"pid" is the process id to be assigned to the new process. If **-p** is not used, the pid will be zero.
-H	Do not start the process immediately. The process will not be started until a "loadstart" command is given for it.
-S stack	NX enforces a minimum stack size of 1000 bytes. If the process requires more stack space, enter an absolute value using this argument. For example, if you specify "-S 1000", the stack size would be what NX already provides. If you specify "-S 2000", the stack size would be twice as big as what NX already provides.
-F free	For C applications only, specify the number of free LDT slots for the process. Free LDT slots are needed if a process allocates memory using "malloc". One slot is used for each 64K of memory. NX normally supplies 20 free slots.

LOAD (cont.)

Arguments (cont.)

-T time Specify the time slice (in milliseconds) for the process to execute. NX defaults to a time slice of 50 milliseconds.

-M[c][d] The "load" command always checks the "/usr/ipsc/conf/cubeconf" file to determine the type of system you have and the default method of loading code and data. If your system has node boards in all slots, code and data are loaded into node board memory and you do not need to use this switch. The default loading methods for other types of systems are:

Large Memory system - code and data are loaded into the LBX memory.
Vector Processor system - code is loaded into node memory and data is loaded into vector (LBX) memory.
Hybrid system - Default is explained on the next page under "Description".

If you have a memory, vector, or hybrid system, you can change the default method of loading using this switch.

"-M" loads both code and data into the node board memory.
"-Mc" loads code into the LBX memory and data into node board memory.
"-Md" loads code into node board memory and data into LBX memory.
"-Mcd" loads both code and data into LBX memory.

-Q file Specifies an alternative pathname for the cube configuration file. The default is "/usr/ipsc/conf/cubeconf".

-K file Loads an alternative node operating system with the pathname specified. The default is "/usr/ipsc/lib/nex".

-L file Loads an alternative LBX memory initialization program with the pathname specified. The default is "/usr/ipsc/lib/lbxmem".

-V file Loads an alternative vector board initialization program with the pathname specified. The default is "/usr/ipsc/lib/lbxvx".

-X file Loads an alternative vector driver program with the pathname specified. The default is "/usr/ipsc/lib/vxd".

node The number of a node into which the process is to be loaded. You can enter multiple node numbers, separated by a space. If no node numbers are given, all nodes will be loaded.

file The pathname of the application process file to be loaded.

LOAD (cont.)

Description

When you load NX with the "load" command, a message is displayed giving the number of system (message) buffers and the exact amount of available memory. When you load process(es) with the "load" command, the number of bytes the process consumed of node board memory and of LBX memory (in extended systems) separated by a "/" is logged in the system log file.

Multi-process Loading

For multi-process applications, invoke the "load" command for each process separately. Use the "-c" argument only on the *first* invocation. Assign each a unique "pid". You can vary the values of the other arguments for each load. For example, use the "-H" on one load and not on another.

Automatic Spill Capability

If you have a vector or large memory system and the program you are loading is too large to fit into the memory specified in the **load** invocation, **load** automatically "spills" the code or data segments into the companion slot's memory.

For example, a vector system has 1M byte of vector board memory and 1/2M byte of node board memory. By default, code segments are loaded into the node and data segments are loaded into the vector board. **Load** loads data segments into the vector board and code segments into the node board until it reads a segment too large to fit in the remaining memory. If this is a data segment, it then loads this large segment into the node board (if enough memory is available there) and issues a message "Warning: Spilling data segment from LBX to Node board" (the reverse occurs for code segments). Subsequent data segments will be loaded on the vector board or spilled to the node board depending on their size. A similar operation is done for large memory systems.

In vector systems, the spill capability may cause your program to terminate with an exception if any data used in a vector operation is spilled to the node board. Therefore, it is advisable to declare all vector data early in the program before declaring non-vector data so that only non-vector data and linked library data is spilled to the node board.

Loading Programs into Hybrid Systems

Default loading for hybrid systems is determined as follows. The nodes specified in the load command are referenced in the cube configuration file to see what type they are. From this, the system determines the one, default method to use to load them all.

If all the nodes are of the same type, the default method for loading is the one used for that type of node. For example, if all nodes have 4M byte memory boards in the LBX slot, code and data are loaded into LBX memory.

However, if the types of nodes are different, the most restrictive default is used. For example, if the load invocation included both memory and vector nodes, the limiting factor is the memory size of the vector board. Thus, the default method for loading is the one used for Vector Processor systems (code on the node, data on the LBX board).

If the default does not provide the desired result, you can specify the numbers of the precise nodes

LOAD (cont.)

you want loaded. Assuming you specify nodes of the same type, "load" will select the "correct" default. You can also change the default using the "-M" switch.

Errors

configuration file: <XENIX message>	The configuration file cannot be accessed (see **-Q**).
Invalid dimension <n>	The dimension given is negative or too large.
Invalid -M ldr_parameters	Letters other than "c" and "d" were used with **-M**.
Invalid node <n>	The node number is not in the valid range.
Invalid PID <n>	The pid is negative.
Loader: cannot open <file> Loader: cannot read <file>	Cannot access the file specified to be loaded.
Loader: Cube response timed out	NX has crashed or the cube manager/node communication is not working. Execute "load -c".
Loader: duplicate PID on node <n>	An active process with the given pid already exists on that node. Specify another pid with **-p**.
Loader: file too short: <filename> Loader: invalid file <filename>	The file specified to be loaded does not contain a valid loadable program.
Loader: invalid node address <n>	The node number is not in the valid range.
Loader: not enough extended memory on node <n>	LBX memory is not large enough for the program.
Loader: not enough memory on node <n>	Node memory is not large enough for the program.
Loader: too many free slots <n>	The value given with the **-F** switch is too large (maximum is 8175).
Loader: too many processes on node <n>	The process table on that node does not have room for another process.

Informational Messages

Loader: load: <filename> PID: <n> into <n> nodes

Loader: start PID: <n> in <n> nodes

Warning: cube will be operated in contention mode	The cube configuration "cubeconf" file is set to polling mode and you invoked "load -cc" to operate this run in contention mode.

LOADKILL

Kills node processes. If you use "loadkill" with no arguments, it will clean out every process in the cube, leaving only the operating system. "Loadkill" also waits for the processes to complete, unlike the cube manager library "lkill" routine. If the copy of the operating system is no longer good, a new copy is automatically reloaded.

Invocation Sequence

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

Arguments

-p pid	"pid" is the process id of the processes to be killed. If **-p** is not used, all processes will be killed.
node	The number of a node in which processes are to be killed. If no node numbers are given, all processes are killed.

Errors

Invalid node <n>	The node number is not in the valid range.
Invalid PID <n>	The pid is negative.
Kill failed LOADKILL FAILED Wait failed	The loadkill failed. A previous message tells why.
Loader: Cube response timed out	NX has crashed or the cube manager/node communication is not working. Execute "load -c".
loadkill: Cannot reload cube	"Loadkill" failed to reload cube after second attempt. Execute "load -R".
Unknown switch <s>	Invalid argument <s>.

Informational Messages

Loader: kill PID: <n> in <n> nodes

loadkill: complete

Reloading

LOADSTART

Starts processes which have been loaded with the **-H** flag and are not already started. If you use "loadstart" with no arguments, it will start every process in the cube. It has no effect on processes that are already started.

Invocation Sequence

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

Arguments

-p pid	"pid" is the process id of the processes to be started. If **-p** is not used, all processes will be started.
node	The number of a node in which processes are to be started. If no node numbers are given, processes on all nodes will be started.

Errors

Invalid node <n>	The node number is not in the valid range.
Loader: Cube response timed out	NX has crashed or the cube manager/node communication is not working. Execute "load -c".
Start failed	The loadstart failed. A previous message tells why.
Unknown switch <s>	Invalid argument <s>.

LOADWAIT

Waits for a process to complete. If you use "loadwait" with no arguments, it will wait for the first process to complete anywhere in the cube.

"Loadwait" prints a message for the process when it completes. The message format is:

Node 99 PID 99 completed, code 99 99

The first code is the result code which indicates whether the process completed normally or not. The second code is the exit code if the result code is zero. Otherwise, it may contain additional information. All numbers are in decimal.

Completion codes are described on the next page.

Invocation Sequence

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

Arguments

-p pid	"pid" is a specific process id to wait for. If **-p** is not used, "loadwait" waits for any process.
node	The number of a node in which processes are to be found. If no node numbers are given, "loadwait" waits for any process in the cube.

Errors

Invalid node <n>	The node number is not in the valid range.
Invalid PID <n>	The pid is negative.
Loader: Cube response timed out	NX has crashed or the cube manager/node communication is not working. Execute "load -c".
Unknown switch <s>	Invalid argument <s>.
Wait failed	The loadwait failed. A previous message tells why.

Informational Messages

Loader: <n> processes exited PID: <n> in <n> nodes

Loader: process exit NODE: <n> PID: <n> CODE: <n> <n>

node <n> pid <n> completed, code <n> <n> <message>

COMPLETION CODES

Each process returns two, 16-bit values when it exits or is killed.

First Value

The first value (least significant 16 bits) is the result code which indicates how the process ended:

Result Code	Reason	Result Code	Reason
0	Normal exit	10	Not used
1	Divide error exception	11	Not used
2	Integer overflow exception	12	Not used
3	Bound range exceeded exception	13	Invalid opcode exception
4	General protection fault	14	Processor extension segment overrun interrupt
5	Segment not present fault		
6	Stack segment overrun or not present	15-31	Not used
		32	Single step interrupt
7	Floating point address exception	33	Breakpoint interrupt
8	Not used	34	NMI interrupt
9	Not used	35	Not used
		36	Parameter error

Second Value

The second value (most significant 16 bits) is the exit code which contains additional information, depending on the first value.

If 1st value is 0	Second value contains the exit code (value supplied to exit() in C, or value on STOP statement in FORTRAN).
If 1st value is 7	Second value contains the floating point status word.

RELCUBE

Allows the user, who currently has access rights, to remove exclusive access rights to the cube (obtained by invoking "getcube") so that another user may have access. Also, the superuser (root) can issue "relcube" to release the current user's access to the cube (refer to the iPSC System Administrator's Guide).

After invoking this command, three events occur:

- a message is displayed that log file entries will automatically be recorded in the default log file (/usr/ipsc/log/LOGFILE), equivalent to invoking the "cubelog" command.
- the message "relinquishing cube from <user ID>" is displayed.
- an entry is made in the system log file

If you do not have access rights currently, the message "cube assigned to <user ID>" is displayed.

Invocation Sequence

relcube

Arguments

None.

Errors

Usage: relcube	Try invoking the command again. If problem persists, execute "shutdown" and reboot XENIX.
Please wait	The program is trying to access /etc/cubetab. If it cannot, the program aborts. Try running /etc/fixperms. If problem persists, remove the "/dev/asglock" and "/etc/cubetab" files.
relcube: Can't set lock	Try running /etc/fixperms.
relcube: Can't create control file	Try running /etc/fixperms.
relcube: Can't read /dev	Try running /etc/fixperms.
relcube: No tty	Execute "shutdown" and reboot XENIX.
relcube: name too long	Someone has tampered with the "relcube" code. Recopy it from floppy or tape.

SUMMARY OF NORMAL USE

This section shows a sequence of cube manager commands demonstrating their usage for several situations encountered when operating the cube: loading the first application and subsequent ones, what to do when you are finished using the cube, and what to do if you have to make code corrections. Be sure to invoke commands in the order in which they are presented.

In this sample sequence of commands, you should be aware of the following:

- The node processes can be loaded either by a call from a cube manager process using the "load" routine or from the terminal using the "load" cube manager command.
- "Host" and "host2" are names of cube manager processes; "node" is the name of a node process.

To load the first application:

getcube	Acquire exclusive use of the cube.
cubelog -l mylog	Start logging to a file, called "mylog", in current directory.
tail -f mylog &	Invoke XENIX "tail" command with "-f" option which displays all the log file entries on the screen, then initiates an infinite loop polling the log file for new input.
load -c	Initially load NX into the cube (the cube's dimension is obtained from the "cubeconf" file). Executed only once if "loadkill" is executed subsequently.
host	Start the execution of the cube manager process. In this case, the cube manager process loads the node process using the "load" routine. If your cube manager process does not require input during execution, you may want to invoke it with an "&" so it will run in the background.

If you are doing several runs, you can load the next application by invoking:

loadkill	Clear processes out of the cube. If necessary, NX will be reloaded.
cubelog -l newlog	To keep separate log files for each run, change the message logging to a new file.
kill [process id]	Kill the original "tail" command. To find out the process id, use the XENIX "ps" command.
tail -f newlog &	Start displaying the contents of the new log file.

(continued on next page)

SUMMARY OF NORMAL USE (cont.)

load -p 1 node	Load the node process into all nodes of the cube, assigning "1" as the process id, and start execution.
host2	Start the execution of another cube manager process.

When you are finished using the cube, you should:

loadkill	Clear out the cube. If necessary, NX will be reloaded.
relcube	Release exclusive use of the cube. This is particularly important in multi-user environments. This command automatically executes "cubelog" to return the system logging to the default log file (/usr/ipsc/log/LOGFILE).
kill [process id]	Kill the "tail" command if it is still running in the background. To find out the process id, use the XENIX "ps" command.

If you need to make corrections to code (be sure to release access to the cube prior to doing this):

edit	Using any editor, correct the source code file(s).
make	Create new application objects.

Refer to the iPSC Program Development Guide for more information.

CHAPTER 6

C ROUTINES

INTRODUCTION

This chapter is a reference section describing each of the routines in the two C system interface libraries: cube manager and node. Note that the name of the library is in bold letters on the right hand side of the page; the name of the routine on the left.

To obtain a more general understanding of the routines and how to use them, refer to Chapter 3, Developing C Programs, in the iPSC Program Development Guide.

The routines are documented in six parts:

- name of the routine and a brief description
- the calling sequence
- the input parameters. These parameters are required and have no default values. In addition, you must specify them in the exact order presented in the calling sequence.
- the return values or results of executing the routine
- a description (occasionally omitted when not necessary)
- errors that can occur through the use of this routine. All errors cause the process that originated them to stop running and an error message to be written in the system log file. For more information about errors, refer to Chapter 5 in the iPSC Program Development Guide. Errors are displayed and logged in the following format:

  9/9/86 16:59 NODE: xxx PID: zzzz error message

The first four items in an error message are: date, time, node number, and process id. The fifth item, *error message*, has the following format:

Type of error: *xxxx* in *call* at *yyyy:zzzz*

where:

Type of error is the message text specifying the error or exception

xxxx is the value of the invalid parameter, or exception number

call is the name of the system call in which the error occurred

yyyy:zzzz is the memory location, or address, of the error in hexadecimal. It consists of two parts: the code segment (*yyyy*) and the instruction pointer (*zzzz*)

Only *Type of error* is documented for each routine in this chapter.

SUMMARY OF CALLING SEQUENCES

The following tables summarize the calling sequence for each routine in the cube manager and node interface libraries for C.

Table 6-1
Calling Summary for C
Cube Manager Routines

Procedure	Calling Sequence
CCLOSE	`cclose(ci);`
COPEN	`ci = copen(pid);`
CUBEDIM	`result = cubedim();`
LKILL	`lkill(node, pid);`
LOAD	`load("filename", node, pid);`
LWAIT	`lwait(node, pid, &cnode, &cpid, &ccode);`
LWAITALL	`lwaitall(node, pid);`
MYPID	`pid = mypid();`
PROBEMSG	`result = probemsg(pid);`
RECVMSG	`recvmsg(ci, &type, buf, len, &cnt, &node, &pid);`
SENDMSG	`sendmsg(ci, type, buf, len, node, pid);`
SYSLOG	`syslog(pid, "desired message");`

Table 6-2
Calling Summary for C Node Routines

Procedure	Calling Sequence
CCLOSE	`cclose(ci);`
CLOCK	`result = clock();`
COPEN	`ci = copen(pid);`
CUBEDIM	`result = cubedim();`
FLICK	`flick();`
GREENLED	`greenled(state);`
HANDLER	`handler(type, new_handler);`
MYNODE	`result = mynode();`
MYPID	`pid = mypid();`
PROBE	`result = probe(ci, type);`
RECV	`recv(ci, type, buf, len, &cnt, &node, &pid);`
RECVW	`recvw(ci, type, buf, len, &cnt, &node, &pid);`
REDLED	`redled(state);`
SEND	`send(ci, type, buf, len, node, pid);`
SENDW	`sendw(ci, type, buf, len, node, pid);`
STATUS	`result = status(ci);`
SYSLOG	`syslog(pid, "desired message");`

CUBE MANAGER SYSTEM INTERFACE LIBRARY

This section describes the C language interface to the cube manager system library. Refer to Chapter 3, Developing C Programs, in the iPSC Program Development Guide for examples of sequences of these routines.

All processes that use this interface library must conform to the 80286 "large" or "huge" model of segmentation. Refer to the XENIX 286 Programmer's Guide for more information.

In order for your C application program to call any of the procedures from the cube manager system interface library, you must define the external declarations for the functions and routines you are using in your program. As an aid, the "chost.def" file is provided which contains the external declarations for the cube manager library. If you want to include the entire file in the source code, use the statement:

```
#include  </usr/ipsc/lib/chost.def>
```

Otherwise, you can declare only those particular routines or functions that are actually used in your source code.

CCLOSE — Cube Manager

CCLOSE destroys the specified communication channel created by a previous "copen".

Calling Sequence

```
int ci;
  .
  .
  .
cclose(ci);
```

Input Parameters

`ci`	is the channel identifier of the channel you want to close. `ci` was returned by XENIX as a result of a previous "copen".

Return Values

None.

Description

Once a process has closed a channel, messages that are addressed to the process will be stored by XENIX until a channel is reopened. The only exception to this occurs when parent and child processes are sharing the same channel. If either the parent or the child process closes its channel, messages are delivered to the process that still has the channel open.

This routine allows you to explicitly close a channel. Any channels not explicitly closed are automatically closed when the process terminates.

Errors

Invalid channel

COPEN — Cube Manager

COPEN creates a communication channel for a cube manager process.

Calling Sequence

```
int  ci;
int  pid;
 .
 .
 .
ci = copen(pid);
```

Input Parameters

`pid`	is the user-defined process id of the process opening the channel. The system uses the pid as a reference in all messages sent via the channel. Valid range is 0 - 32767. It is recommended that all the channels opened by a single process be assigned the same process id.

Return Values

`ci`	is the channel identifier used by XENIX to manage the transmission and reception of messages.

Description

Once a channel has been established, it may be used to send and receive messages. Messages cannot be sent or received until a channel is opened.

Unlike node processes which can perform simultaneous send and receive operations over multiple channels, a cube manager process can perform only one operation over one channel at a time. This is due to certain characteristics of the XENIX operating system.

The maximum number of channels a cube manager process can have is 18. If you attempt to open more than 18 channels, XENIX will log the "Too many open files" error in the log file.

Errors

Out of memory	The system has insufficient memory to create the structures necessary to manage the communication channel.
Invalid pid	The valid range is 0 - 32767.
Too many open files	A process attempted to open more than the maximum of 18 channels.

CUBEDIM — Cube Manager

CUBEDIM returns a value that is the dimension of the cube in which the program is running.

Calling Sequence

```
int  result;
 .
 .
 .
result = cubedim();
```

Input Parameters

None.

Return Values

`result`	is a value from 0 to 7 for applications running in 1 to 128 nodes. For example, "cubedim" returns 6 for a 64-node cube (or a 2^6-node cube). This is the dimension you specified in the "dim" argument of the "load" command (if "dim" was not specified, it is the value of *dimension* in the "cubeconf" file).

Description

You can use "cubedim" to find out how many nodes an application is running on in order to have the cube manager process send a message to all nodes. A sample statement that would return the number of nodes is:

```
numnodes=1<<cubedim();
```

Errors

None.

LKILL — Cube Manager

LKILL kills the selected process (or processes) in the selected node (or nodes).

Calling Sequence

```
int node;
int pid;
  .
  .
lkill(node, pid);
```

Input Parameters

node	the number of the node in which the process (or processes) is to be killed. Use -1 to select all nodes.
pid	the id of the process to be killed. Use -1 to select all processes. No error results if you specify a nonexistent process.

Return Values

None.

Description

This routine kills the node process specified by "pid". The pid may match a pid given in a previous load, or it may be -1 to kill all processes. "Node" is the node number. Node numbers may range from 0 to (2**cubedim)-1, or -1 for the special case of all nodes. The cube manager number, -32768, is invalid.

Errors

Loader: invalid node address <n>

Loader: Cube Response Timed Out — Execute "load -c".

LOAD — Cube Manager

LOAD loads the file into the specified node (or all nodes if -1 is specified for the node number) and starts it.

Calling Sequence

```
int node;
int pid;
  .
  .
load("filename", node, pid);
```

Input Parameters

`"filename"`	the pathname of the file to be loaded.
`node`	the number of the node into which the process is to be loaded. Use -1 to select all nodes.
`pid`	the process id to be assigned to the process once it has been loaded. An error results if you specify a pid that is already in use by a previously loaded process. A pid may be reused as soon as the process completes. Up to 20 processes per node are allowed.

Return Values

None.

Description

Loads the file (a character string containing the pathname of the file) into a node (specified by its number). Node numbers may range from 0 to (2**cubedim)-1, or -1 for the special case of all nodes. The cube manager number, -32768, is invalid, because this routine does not load processes into the cube manager.

The pid is saved by NX for use in identifying the process in other routines. By convention, this should be the same as the pid used for the process in SEND and RECV routines. Use the MYPID routine in your node program to obtain the process id.

Note that, unlike the "load" cube manager command, the LOAD routine does not provide as many options, such as changing stack size and execution time slice.

LOAD (cont.) Cube Manager

Errors

Invalid file name pointer

Loader: cannot open <file>

Loader: cannot read <file>

Loader: duplicate PID on node <n>

Loader: file too short: <filename>

Loader: invalid file <filename>

Loader: invalid node address <n>

Loader: not enough extended memory on node <n>

Loader: not enough memory on node <n>

Loader: too many processes on node <n>

Loader: Cube Response Timed Out	Execute "load -c".

LWAIT — Cube Manager

LWAIT waits for the selected process in the selected node (or nodes) to complete. It then returns the pid, node, and completion code. If you enter -1 for all processes, LWAIT returns as soon as the ***first*** one completes.

Calling Sequence

```
int node;
int pid;
int cnode;
int cpid;
long ccode;
  .
  .
lwait(node, pid, &cnode, &cpid, &ccode);
```

Input Parameters

node	the number of the node in which the process (or processes) resides. Use -1 to select all nodes.
pid	the process id of the process to wait for. Use -1 to select all processes. An error results if you wait for a process with a specific pid and that pid does not exist.

Return Values

cnode	a pointer to the location where the node number of the completed process is stored.
cpid	a pointer to the location where the pid of the completed process is stored.
ccode	a pointer to the location where the completion code of the completed process is stored. The least significant 16 bits of the code give the reason the process finished. The most significant 16 bits give the exit code or additional information. Refer to the next page for a description of completion codes.

Description

Waits for a process to complete on any one of the nodes. The pid is a process id, or -1 for the special case of all processes. Node numbers may range from 0 to (2**cubedim)-1, or -1 for the special case of all nodes. The cube manager number, -32768, is invalid.

When a process with a matching pid and node completes, the node, pid, and completion code of the process are returned in cnode, cpid, and ccode, respectively.

LWAIT (cont.) Cube Manager

Errors

Invalid ccode pointer	
Invalid cnode pointer	
Invalid cpid pointer	
Loader: invalid node address <n>	
Loader: Cube Response Timed Out	Execute "load -c".

COMPLETION CODES

Each process returns two, 16-bit values when it exits or is killed.

First Value

The first value (least significant 16 bits) is a result code which indicates how the process ended:

Result Code	Reason	Result Code	Reason
0	Normal exit	10	Not used
1	Divide error exception	11	Not used
2	Integer overflow exception	12	Not used
3	Bound range exceeded exception	13	Invalid opcode exception
4	General protection fault	14	Processor extension segment overrun interrupt
5	Segment not present fault		
6	Stack segment overrun or not present	15-31	Not used
		32	Single step interrupt
7	Floating point address exception	33	Breakpoint interrupt
8	Not used	34	NMI interrupt
9	Not used	35	Not used
		36	Parameter error

Second Value

The second value (most significant 16 bits) is the exit code which contains additional information, depending on the first value.

If 1st value is 0	Second value contains the exit code (value supplied to exit() in C, or value on STOP statement in FORTRAN).
If 1st value is 7	Second value contains the floating point status word.

LWAITALL — Cube Manager

LWAITALL waits for one or all processes in one or all nodes to complete. If you enter -1 for all processes, LWAITALL, unlike LWAIT, waits for all processes to complete.

Calling Sequence

```
int node;
int pid;
  .
  .
lwaitall(node, pid);
```

Input Parameters

node	the number of the node in which the process (or processes) resides. Use -1 to select all nodes.
pid	the process id of the process to wait for. Use -1 to select all processes. Unlike LWAIT it is not an error to wait for a nonexistent process. The call will simply return immediately.

Return Values

None.

Description

Waits for processes specified by "pid" on "node" to complete. The pid is a process id, or -1 for the special case of all processes. Node numbers may range from 0 to (2**cubedim)-1, or -1 for the special case of all nodes. The cube manager number, -32768, is invalid.

The LWAITALL routine does not return until all specified processes on all specified nodes are completed. The completion codes for the processes are discarded.

Errors

Loader: invalid node address <n>

Loader: Cube Response Timed Out — Execute "load -c".

MYPID — Cube Manager

MYPID returns the XENIX process id of the calling process.

Calling Sequence

```
int pid;
  .
  .
pid = mypid();
```

Input Parameters

None.

Return Values

`pid`	value of the XENIX process id.

Description

This routine provides the XENIX process id of a cube manager process. Do not confuse this routine with the node MYPID routine which returns the process id given in the "load" command or routine.

Errors

None.

PROBEMSG — Cube Manager

PROBEMSG returns a value indicating whether a message is pending for a cube manager process.

Calling Sequence

```
int status;
int pid;
  .
  .
result = probemsg(pid);
```

Input Parameters

`pid`	the process id of the cube manager process for which to check outstanding messages. This is the same process id used when opening a channel. Valid pid range is 0 - 32767. Use pid "-1" to check messages for all cube manager processes.

Return Values

`result`	one of two values: 0 - if there are no messages for any process (if pid is -1) or for the specified cube manager process. 1 - if there is one or more messages pending for any process (if pid is -1) or for a specified process.

Description

This routine allows you to check on outstanding (waiting to be received) messages for one or all cube manager processes. Although there is some correlation between this routine and the node PROBE routine, their parameters are completely different. For example, "type" is only an issue in node processes, not cube manager processes.

An example of its usage is:

```
if (probemsg(99))
  recvmsg(ci, &type, buf, len, &cnt, &node, &pid);
```

Error Messages

Invalid pid	The valid range is -1 through 32767.

RECVMSG — Cube Manager

RECVMSG initiates the receipt of a message from a node process or another cube manager process.

Calling Sequence

```
int  ci;
int  type;
char buf[n];
int  len;
int  cnt;
int  node;
int  pid;
  .
  .
  .
recvmsg(ci, &type, buf, len, &cnt, &node, &pid);
```

Input Parameters

`ci`	is the channel identifier that was returned to the process as a result of a previous "copen".
`len`	is the number of bytes in the message buffer pointed to by `buf`. You should probably specify `len` to be as large as the biggest message you expect to receive.

Return Values

For each of the parameters described below, a null pointer may be handed to the function, in which case, no value is returned. The parameters, "cnt" or "len", must be null if "buf" is null.

`&type`	is a pointer to the location where the type of the message received will be stored at the completion of the "recvmsg" routine.
`buf`	is a pointer to the process message buffer where the received message is to be stored.
`&cnt`	is a pointer to the location where the number of message bytes received is stored. If the value of `cnt` is less than or equal to the value of `len`, the entire message was received. If the value of `cnt` is greater than `len`, a message whose length was greater than the size of the message buffer was received. Only the first `len` bytes were stored in the message buffer. The remainder of the message is truncated and cannot be recovered.
`&node`	is a pointer to the location where the node id of the process that sent the message is stored.

RECVMSG (cont.) Cube Manager

`&pid`	is a pointer to the location where the process id of the process that sent the message is stored.

Description

The RECVMSG routine receives any message whose destination node and process id's match the cube manager process that originated the RECVMSG request.

Unlike node processes, the "type" of message is a returned value and is not used as a reception qualifier.

This routine causes the calling process to be blocked until a message has been received. When the calling process resumes execution, the message that was requested has been received and stored in the user message buffer.

This routine uses nearly the same parameters as the node system routine RECVW. The only difference is that the type field is returned rather than used for message selection.

Errors

Allocated buffer space less than length	The message buffer, as defined by the buffer pointer (buf) and the buffer length (len) parameters, extends beyond the end of the data segment in which it starts. This means that the entire message buffer cannot be accessed. This error is usually caused by the length parameter being too large.
Invalid buffer length	The length of the message is negative.
Invalid buffer pointer	
Invalid channel	
Invalid count pointer	
Invalid node pointer	
Invalid pid pointer	
Invalid type pointer	

SENDMSG — Cube Manager

SENDMSG initiates the transmission of a message from a cube manager process to a process on the cube manager, on a node, or on all nodes if -1 is specified as the node parameter.

Calling Sequence

```
int  ci;
int  type;
char buf[n];
int  len;
int  node;
int  pid;
  .
  .
  .
sendmsg(ci, type, buf, len, node, pid);
```

Input Parameters

`ci`	is the channel identifier that was returned to the process as a result of a previous "copen".
`type`	is an integer value referring to the type of message to be sent. This can be any value in the range of 0 to 32767. Negative values are reserved for use by the operating system.
`buf`	is a pointer to the buffer that contains the message to be sent. `buf` can be null if `len` is zero.
`len`	is the number of bytes to be transmitted from the buffer. The maximum number of bytes that may be sent in a single message is 16384 (16 kilobytes).
`node`	is the id of the node being sent the message. Valid node id's range from 0 to 127, but may not exceed the value implied by the "dim" parameter in the "load" command. The cube manager's address (-32768) is also valid, but specify it using the hex value of 0x8000. Specify this parameter as -1 to send the message to all processes currently executing in the cube whose `pid` matches the one given in this call.
`pid`	is the process id of the process being sent the message. Process id's range from 0 to 32767. Negative values are reserved for the operating system.

Return Values

None.

SENDMSG (cont.) Cube Manager

Description

The return to the calling process from the SENDMSG routine implies that the user message buffer is available for reuse. This does not indicate that the message has been received by the destination process.

This routine performs similarly to the node system routine SENDW.

Errors

Allocated buffer space less than length	The message buffer, as defined by the buffer pointer (buf) and the buffer length (len) parameters, extends beyond the end of the data segment in which it starts. This means that the entire message buffer cannot be accessed. This error is usually caused by the length parameter being too large.
Invalid buffer length	The length of the message is negative or is greater than 16,384.
Invalid buffer pointer	
Invalid channel	
Invalid node	The node specified is greater than 127, negative, or not equal to -32768.
Invalid pid	The valid range is 0 - 32767.
Invalid type	The valid range is 0 - 32767.

SYSLOG — Cube Manager

SYSLOG enables a process to write a message into the system log file.

Calling Sequence

```
int  pid;
 .
 .
 .
syslog(pid, "desired message");
```

Input Parameters

`pid`	is the process id of the process writing the message into the log file.
`"desired message"`	Maximum message length is 120 characters.

Return Values

None.

Description

The argument string is prefixed with a timestamp, the node id, and process id of the originating process when the entry is written into the file. For example:

```
syslog(3, "hello");
```

would produce an entry similar to

```
11/15/86 14:32  NODE: -32768 PID: 3 : hello
```

in the system log file.

For more information about the log file, refer to the "cubelog" command in Chapter 2.

Errors

Invalid message pointer	The message to be logged cannot be accessed using the supplied pointer.
Invalid pid	The valid range is 0 - 32767.

NODE SYSTEM INTERFACE LIBRARY

The following section describes the C language interface to the node system routines. Refer to Chapter 3, Developing C Programs, in the iPSC Program Development Guide for examples of sequences of these routines.

All processes that use this interface library must conform to the 80286 "large" or "huge" model of segmentation. Refer to the XENIX 286 Programmer's Guide for more information.

In order for your C application program to call any of the procedures from the node system interface library, you must define the external declarations for the functions and routines you are using in your program. As an aid, the "cnode.def" file is provided which contains the external declarations for the node library. If you want to include the entire file in the source code, use the statement:

```
#include  </usr/ipsc/lib/cnode.def>
```

Otherwise, you can declare only those particular routines or functions that are actually used in your source code.

CCLOSE

Node

CCLOSE destroys a specified communication channel created by a previous "copen".

Calling Sequence

```
int  ci;
 .
 .
 .
cclose(ci);
```

Input Parameters

`ci`	is the channel identifier of the channel you want to close. It was returned by the node operating system as a result of a previous "copen".

Return Values

None.

Description

The CCLOSE routine should only be invoked when the process no longer desires to communicate via the channel. This routine allows you to explicitly close a channel. Any channels not explicitly closed will be closed when the process terminates.

The CCLOSE routine maintains the integrity of any messages awaiting transmission on the channel. That is, if CCLOSE is invoked following a SEND routine, the message is still transmitted.

There are three potential situations that can occur by invoking CCLOSE following a RECV request:

- the message has not yet been received. The channel is closed without waiting for the receipt of the message. If another channel were opened, the message could still be received.
- the message is being received when CCLOSE is invoked. The channel is closed and the message is lost.
- The message has been received when CCLOSE is invoked. The message is received intact with the return values updated and the channel is closed.

It is recommended that all receive operations on a channel be completed prior to closing the channel. This can be done using STATUS.

Errors

Invalid channel

CLOCK — Node

CLOCK returns a value that is the number of elapsed millisecond intervals since the node was initialized, modulo 2^{32}.

Calling Sequence

```
long clock();
long result;
 .
 .
 .
result = clock();
```

Input Parameters

None.

Return Values

`result`	is a value that is the number of millisecond intervals since the node was initialized, modulo 2^{32} (49 days).

Description

The CLOCK routine provides a simple timing mechanism to show how long the node has been running. It is not recommended that the CLOCK routine be used to synchronize different aspects of process execution, because each node has its own "clock" value.

The returned value is updated every 5 milliseconds. That is, every 5 milliseconds, the value is incremented by 5. The clock starts keeping time when the "load -c" command is invoked.

Errors

None.

COPEN

Node

COPEN creates a communication channel for a node process.

Calling Sequence

```
int  ci;
int  pid;
 .
 .
 .
ci = copen(pid);
```

Input Parameters

`pid`	is the process id of the process opening the channel. The pid given is used as a reference in all messages sent via the channel. The valid range is 0 to 32767. Negative process id's are reserved for the operating system. It is recommended that all the channels opened by a single process be assigned the same process id and that this pid agrees with the pid used in the "load" command or routine.

Return Values

`ci`	is the channel identifier used by the operating system to manage the transmission and reception of messages.

Description

Once a channel has been established, it may be used to send and receive messages. Messages cannot be sent or received until a channel is opened.

Message sending and receiving cannot be performed on the same channel concurrently. If you want to do concurrent sends and receives, open multiple channels. The number of channels you will need is based on the number of concurrent sends and receives you want to perform.

The maximum number of channels a node can have (regardless of how many processes are executing) is approximately 250.

Errors

Invalid pid	The valid range is 0 - 32767.
No more channels available in copen	The application has exceeded the approximate limit of 250.

CUBEDIM — Node

CUBEDIM returns the dimension of the cube in which the process is running.

Calling Sequence

```
int  result;
 .
 .
 .
result = cubedim();
```

Input Parameters

None.

Return Values

`result`	is a value from 0 to 7 for applications running in 1 to 128 nodes. For example, "cubedim" returns 6 for a 64-node cube (or a 2^6- node cube). This is the dimension you specified in the "dim" argument of the "load" command (if "dim" was not specified, it is the value of *dimension* in the "cubeconf" file).

Description

You can use CUBEDIM to find out how many nodes an application is running on. One example of its usage is when a controlling node is going to receive results from all nodes and needs to know how many nodes that will be. A sample statement that would return the number of nodes is:

```
numnodes=1<<cubedim();
```

Errors

None.

FLICK

Node

FLICK allows a node process to defer execution to another process on the same node.

Calling Sequence

```
flick();
```

Input Parameters

None.

Return Values

None.

Description

The scheduler runs processes on any node in round-robin fashion, running each process for approximately 50 milliseconds (this can be changed using the **-T** switch on the "load" command), or until the process explicitly relinquishes the processor with the FLICK routine. The execution of the process is not deferred indefinitely. Instead, the process will resume execution during its next scheduled time frame. FLICK simply delays execution to allow other node processes to execute.

You can use the FLICK routine when a process is waiting for the change in status on one or more channels, and has no other work to do. A node process should never "busy loop" looking at the status of a channel that has a pending receive request. The process sending the desired message may be running on the same node, but be suspended waiting for the processor.

FLICK is a delay mechanism, not a synchronization mechanism. The scheduler does not know what conditions a process is waiting for. For example, a process may be waiting for the completion of several concurrent message operations.

The following examples show a correct and incorrect usage of FLICK.

FLICK (cont.) Node

Example 1 - Correct usage

```
while (status(ci))
      flick();
```

Example 2 - Incorrect usage

```
if (status(ci))
   flick();
```

In Example 2, control may be passed to the statements following the "if" statement before the status of the channel has changed.

Errors

None.

GREENLED — Node

Allows the process to turn its node board's green LED on or off.

Calling Sequence

```
int state;
  .
  .
greenled(state);
```

Input Parameters

state	is a value: either 0 (for off) or 1 (for on).

Return Values

None.

Description

The node board's green LED can be turned either on or off depending on the value of "state". Results are not predicted when "state" is not 0 or 1.

Error Messages

None.

HANDLER — Node

HANDLER allows a process to assign the execution of a user-written exception handler to the occurrence of a specific exception. A process can link exception handlers that are effective only for exceptions caused by the calling process.

Calling Sequence

```
int  type;
int  (*new_handler) ();
 .
 .
 .
handler(type, new_handler);
```

Input Parameters

type	is the type code of the hardware exception. Values in the range 0 to 16, except for 7, 8, 10, 12, 14 and 15, are valid. Valid type codes are described in Table 3-3.
new_handler	is a pointer to the starting address of the user-written exception handler.

Return Values

None.

Description

The interface between NX and user-written exception handlers is the same as the interface for exception handlers that are called through trap gates and execute on bare hardware (as defined in the iAPX 286 Programmer's Reference manual). The state of the interrupted procedure is saved on the stack. Certain exceptions also cause an error code to be pushed on the stack. The handler must return with an IRET (Return from Interrupt or Nested Task) instruction which returns control to an interrupted routine. (IRET is an assembly language instruction for the 286.)

Refer to the iAPX 286 Programmer's Reference Manual for further information regarding exception handling on the 80286.

Errors

Invalid type	Incorrectly specified the exception type parameter.

HANDLER (cont.) Node

Table 6-3
Hardware Exception Type Codes

Type Code	Exception Name	Description
0	Divide error	Occurs if the quotient is too large or an attempt is made to divide by zero.
1	Single step	Allows programs to execute one instruction at a time. If the TF bit in the flag word is set, an interrupt occurs after the next instruction has been executed.
2	NMI interrupt	Occurs upon receipt of an external signal on the NMI pin.
3	Breakpoint interrupt	Occurs when the 1-byte breakpoint instruction (INT 3) is executed.
4	INTO detected overflow	Occurs when the INTO instruction is executed if the overflow bit of the FLAGS register is set.
5	Bounds range exceeded	Occurs when the BOUND instruction is executed if the specified array index is found to be invalid with respect to the given array bounds.
6	Invalid opcode	Occurs if execution of an invalid opcode is attempted. Can also occur if the effective address given by certain instructions, notably BOUND, LDS, LES, and LIDT, specifies a register rather than a memory location.
9	Processor extension segment overrun	Occurs if a processor extension memory operand does not fit in a segment.
11	Segment or call gate not present	Occurs when an attempt is made to load a not-present segment or to use a control descriptor that is marked not-present.
13	General protection fault	Occurs when a violation occurs which is not covered by another interrupt.
16	Floating point	Occurs after the numeric instruction that caused the error. It can only occur while executing a subsequent WAIT or ESC.

MYNODE

MYNODE returns the node ID of the calling process.

Calling Sequence

```
int result;
 .
 .
 .
result = mynode();
```

Input Parameters

None.

Return Values

`result`	is the node id of the process that initiated the routine.

Errors

None.

MYPID — Node

MYPID returns the process id of the calling process.

Calling Sequence

```
int pid;
  .
  .
pid = mypid();
```

Input Parameters

None.

Return Values

pid	value of pid supplied in the cube manager "load" command or routine.

Description

Some examples of this routine's usage are:

```
syslog(mypid(), "desired message");

ci = copen(mypid());
```

Errors

None.

PROBE — Node

PROBE allows a node process to determine whether a message of a given type is available for reception.

Calling Sequence

```
int  result;
int  ci;
int  type;
 .
 .
 .
result = probe(ci, type);
```

Input Parameters

`ci`	is a channel identifier returned to the process as a result of a previous "copen". ***Any*** open channel can be used. For example, if you have opened four channels, use any one of them to probe all four.
`type`	is an integer value referring to the type of message. This can be any value in the range of 0 to 32767. Negative values are reserved for the operating system.

Return Values

`result` is one of two sets of values:

- If a message of the specific type is found, the value returned equals the length (in bytes) of the pending message. This can be any value in the range 0 to 16384. If there are multiple messages of the same type, "probe" returns the length of the message received first.
- If there are no messages, a **negative** value is returned.

Description

This routine "probes" all pending messages for a process and returns a positive or negative value based on whether or not any messages of a given type are available for reception by a process. The "ci" in this routine is not used the same as the "ci" in send and receive operations. "Probe" tests messages based on type only, regardless of the channel the messages will be received on. "Probe" does not block or wait for a previous operation to complete on the specified channel.

PROBE (cont.)

Node

Errors

Invalid channel	
Invalid type	The valid range is 0 - 32767.

RECV — Node

RECV initiates the receipt of a message from another process. You must do a "status" call to determine when the message is received. When "status" returns 0, the message has been received and the return values have been updated. You should not use the buffer for anything else until "status" returns 0.

Calling Sequence

```
int  ci;
int  type;
char buf[n];
int  len;
int  cnt;
int  node;
int  pid;
 .
 .
 .
recv(ci, type, buf, len, &cnt, &node, &pid);
```

Input Parameters

`ci`	is the channel identifier that was returned to the process as a result of a previous "copen".
`type`	is an integer value referring to the type of message to be received. The type field is the only qualifier you can use to select a message to be received. You cannot select a message to be received based on the originator's process or node id. Type can be any value in the range of 0 to 32767. Negative values are reserved for use by the operating system.
`len`	is the length (in bytes) of the message buffer pointed to by `buf`.

Return Values

For each of the parameters described below, a null pointer may be handed to the function, in which case, no value is returned. The `len` parameter must be zero if `buf` is null.

`buf`	is a pointer to the buffer where the received message is to be stored.
`&cnt`	is a pointer to the location where the number of message bytes received is stored. If the value of `cnt` is less than or equal to the value of `len`, the entire message was received. A value of `cnt` greater than `len` indicates a message was received whose length was greater than the size of the message buffer. Only the first `len` bytes were stored in the message buffer. The remainder of the message was truncated and cannot be recovered.

RECV (cont.) Node

&node	is a pointer to the location where the node id of the process that sent the message is stored.
&pid	is a pointer to the location where the process id of the process that sent the message is stored.

Description

The return to the calling process from the RECV routine does not indicate that the message has been received by the requesting process. It does, however, mean that the request to receive the message has been logged by the operating system.

The calling process can determine when the message has been received by the execution of the STATUS procedure. Until the STATUS routine indicates that the request has been completed, the process cannot rely on the validity of the contents of the specified user message buffer.

If the calling process invokes RECV prior to the completion of a previous send or receive request on the channel, execution of the routine is delayed until the previous request is completed.

Errors

Allocated buffer space less than length	The message buffer, as defined by the buffer pointer (buf) and the buffer length (len) parameters, extends beyond the end of the data segment in which it starts. This means that the entire message buffer cannot be accessed. This error is usually caused by the length parameter being too large.
Invalid buffer length	
Invalid buffer pointer	
Invalid channel	
Invalid count pointer	
Invalid node pointer	
Invalid pid pointer	
Invalid type	

RECVW — Node

RECVW initiates the receipt of a message from another process. Execution of this routine causes the calling process to be blocked until the message has been received.

Calling Sequence

```
int  ci;
int  type;
char buf[n];
int  len;
int  cnt;
int  node;
int  pid;
 .
 .
 .
recvw(ci, type, buf, len, &cnt, &node, &pid);
```

Input Parameters

`ci`	is the channel identifier that was returned to the process as a result of a previous "copen".
`type`	is an integer value referring to the type of message to be received. The type field is the only qualifier you can use to select a message to be received. You cannot select a message to be received based on the originator's process or node id. Type can be any value in the range of 0 to 32767. Negative values are reserved for use by the operating system.
`len`	is the length (in bytes) of the message buffer pointed to by `buf`.

Return Values

For each of the parameters described below, a null pointer may be handed to the function, in which case, no value is returned. The `len` parameters must be zero if `buf` is null.

`buf`	is a pointer to the buffer where the received message is to be stored.
`&cnt`	is a pointer to the location where the number of message bytes received is stored. If the value of `cnt` is less than or equal to the value of `len`, the entire message was received. If the value of `cnt` is greater than `len`, a message whose length is greater than the size of the message buffer was received. Only `len` bytes were stored in the message buffer. The remainder of the message was truncated and cannot be recovered.
`&node`	is a pointer to the location where the node id of the process that sent the message is stored.

RECVW (cont.) Node

`&pid` is a pointer to the location where the process id of the process that sent the message is stored.

Description

The execution of the RECVW routine causes the calling process to be blocked until the message has been received. When the calling process resumes execution, the message that was requested has been received and stored in the user message buffer and all return values are updated.

If the calling process invokes RECVW prior to the completion of a previous send or receive request on the channel, execution of the routine is delayed until the previous request is completed.

Errors

Allocated buffer space less than length	The message buffer, as defined by the buffer pointer (buf) and the buffer length (len) parameters, extends beyond the end of the data segment in which it starts. This means that the entire message buffer cannot be accessed. This error is usually caused by the length parameter being too large.
Invalid buffer length	
Invalid buffer pointer	
Invalid channel	
Invalid count pointer	
Invalid node pointer	
Invalid pid pointer	
Invalid type	
Out of system buffers	See **-b** switch on "load" command.

REDLED — Node

Allows the process to turn its node board's red LED on or off.

Calling Sequence

```
int state;
  .
  .
redled(state);
```

Input Parameters

state	is a value: either 0 (for off) or 1 (for on).

Return Values

None.

Description

The node board's red LED can be turned either on or off depending on the value of "state". Results are not predicted when "state" is not 0 or 1.

Error Messages

None.

SEND — Node

SEND initiates the transmission of a message to a process on the same node, on another node, or on all other nodes if -1 is specified as the node parameter. The user message buffer should not be rewritten until a "status" indicates that the send operation is complete.

Calling Sequence

```
int  ci;
int  type;
char buf[n];
int  len;
int  node;
int  pid;
 .
 .
 .
send(ci, type, buf, len, node, pid);
```

Input Parameters

`ci`	is the channel identifier that was returned to the process as a result of a previous "copen".
`type`	is an integer value referring to the type of message to be sent. This can be any value in the range of 0 to 32767. Negative values are reserved for use by the operating system.
`buf`	is a pointer to the buffer that contains the message to be sent. `buf` can be null if `len` is zero.
`len`	is the number of bytes to be transmitted from the buffer. The maximum number of bytes that may be sent in a single message is 16384 (16 kilobytes).
`node`	is the id of the node being sent the message. Valid node ids range from 0 to 127, but may not exceed the value implied by the "dim" parameter in the "load" command. The cube manager's id (-32768) is also valid, but specify it using the hex value 0x8000. Specify this parameter as -1 to send the message to all processes currently executing in the cube whose `pid` matches the one given in this call.
`pid`	is the process id of the process being sent the message. Process id's range from 0 to 32767. Negative values are reserved for operating system functions.

SEND (cont.) Node

Return Values

None.

Description

The return to the calling process from the "send" routine does not imply that the message has been sent, only that the system has been notified that a message is ready for transmission. Therefore, **the contents of the user message buffer should not be modified until a subsequent "status" operation indicates that the send is complete.** A successful send does not imply reception of the message by the destination process only that the message has been successfully sent.

If the calling process invokes "send" prior to the completion of a previous message request on the channel, execution of the call is delayed until the previous request is completed.

The "global" send capability can be used to send messages to a surrounding subcube of dimension *d* (less than or equal to cubedim()) by specifying (d - cubedim() - 1) as the node parameter. For example, in a D5 cube, to have node 10 send a message to a surrounding subcube of dimension 2, the node parameter would be calculated as:

2 - 5 - 1 = -4

The call would be:

```
send(ci, type, buf, len, -4, pid);
```

This would send the message to the surrounding 3 nodes (since d = 2, 4 nodes are involved). The message would be sent to nodes 8, 9, and 11.

In the node parameter calculation, *d* could be set up as a variable that is decremented, in which case, *d* could be 0 making the subcube dimension 0. If this occurs, no messages are sent to any nodes, and the process continues executing.

Subcubes are groups of contiguously numbered nodes. For example, a subcube of dimension 3 has eight nodes; nodes 0, 8, 16, 32, etc. are the "boundary" nodes. A subcube of dimension 2 has four nodes; nodes 0, 4, 8, 12, 16, etc. are the boundary nodes. If node 5 sends a message to a dimension 2 subcube, nodes 4 through 7 will receive the message since node 5 is in that group of contiguous nodes. For more information, refer to Tables 5-5 through 5-8 in the iPSC System Overview Manual.

SEND (cont.) Node

Errors

Allocated buffer space less than length	The message buffer, as defined by the buffer pointer (buf) and the buffer length (len) parameters, extends beyond the end of the data segment in which it starts. This means that the entire message buffer cannot be accessed. This error is usually caused by the length parameter being too large.
Invalid buffer length	The message length is greater than 16,384.
Invalid buffer pointer	
Invalid channel	
Invalid node	The specified node is greater than 127, negative, or not equal to -32768.
Invalid pid	The valid range is 0 - 32767.
Invalid type	The valid range is 0 - 32767.

SENDW — Node

SENDW initiates the transmission of a message to a process on the same node, on another node, or on all other nodes if -1 is specified as the node parameter. Execution of this routine causes the calling process to be blocked until the message has been sent.

Calling Sequence

```
int  ci;
int  type;
char buf[n];
int  len;
int  node;
int  pid;
 .
 .
 .
sendw(ci, type, buf, len, node, pid);
```

Input Parameters

`ci`	is the channel identifier that was returned to the process as a result of a previous "copen".
`type`	is an integer value referring to the type of message to be sent. This can be any value in the range of 0 to 32767. Negative values are reserved for use by the operating system.
`buf`	is a pointer to the buffer that contains the message to be sent. `buf` can be null if `len` is zero.
`len`	is the number of bytes to be transmitted from the buffer. The maximum number of bytes that may be sent in a single message is 16384 (16 kilobytes).
`node`	is the id of the node being sent the message. Valid node ids range from 0 to 127, but may not exceed the value implied by the "dim" parameter in the "load" command. The cube manager's id (-32768) is also valid, but specify it using the hex value 0x8000. Specify this parameter as -1 to send the message to all processes currently executing in the cube whose `pid` matches the one given in this call.
`pid`	is the process id of the process being sent the message. Process id's range from 0 to 32767. Negative values are reserved for operating system functions.

SENDW (cont.) Node

Return Values

None.

Description

The return to the calling process from the "sendw" routine does not imply that the message has been received by the destination process. It does, however, mean that the user message buffer is available for reuse.

If the calling process invokes "sendw" prior to the completion of a previous message request on the channel, execution of the routine is delayed until the previous request is completed.

The "global" send capability can be used to send messages to a surrounding subcube of dimension *d* (less than or equal to cubedim()) by specifying (d - cubedim() - 1) as the node parameter. For example, in a D5 cube, to have node 10 send a message to a surrounding subcube of dimension 2, the node parameter would be calculated as:

2 - 5 - 1 = -4

The call would be:

```
sendw(ci, type, buf, len, -4, pid);
```

This would send the message to the surrounding 3 nodes (since d = 2, 4 nodes are involved). The message would be sent to nodes 8, 9, and 11.

In the node parameter calculation, *d* could be set up as a variable that is decremented, in which case, *d* could be 0 making the subcube dimension 0. If this occurs, no messages are sent to any nodes, and the process continues executing.

Subcubes are groups of contiguously numbered nodes. For example, a subcube of dimension 3 has eight nodes; nodes 0, 8, 16, 32, etc. are the "boundary" nodes. A subcube of dimension 2 has four nodes; nodes 0, 4, 8, 12, 16, etc. are the boundary nodes. If node 5 sends a message to a dimension 2 subcube, nodes 4 through 7 will receive the message since node 5 is in that group of contiguous nodes. For more information, refer to Tables 5-5 through 5-8 in the iPSC System Overview Manual.

SENDW (cont.) Node

Errors

Allocated buffer space less than length	The message buffer, as defined by the buffer pointer (buf) and the buffer length (len) parameters, extends beyond the end of the data segment in which it starts. This means that the entire message buffer cannot be accessed. This error is usually caused by the length parameter being too large.
Invalid buffer length	The message length is greater than 16,384.
Invalid buffer pointer	
Invalid channel	
Invalid node	The specified node is greater than 127, negative, or not equal to -32768.
Invalid pid	The valid range is 0 - 32767.
Invalid type	The valid range is 0 - 32767.

STATUS — Node

STATUS allows a process to determine the state of the user message buffer associated with sending or receiving a message.

Calling Sequence

```
int  result;
int  ci;
 .
 .
 .
result = status(ci);
```

Input Parameters

`ci` — is the channel identifier that was returned to the process as a result of a previous "copen".

Return Values

`result` — returns either a 0 or a 1 and their meanings are as follows:

- Following a "send" request, a 0 indicates that the user message buffer is available for reuse. Following a "recv" request, a 0 indicates that the buffer contains the message.
- 1 indicates that the user message buffer is currently being read from or written to and is **not** available for reuse.

STATUS (cont.) Node

Description

The following program fragment returns the channel identifier of the first channel to contain valid data. There are three channels open, all of which have pending receives.

```
triselect(ca,cb,cc)
{
     for(;;){
          if (!status(ca)) return(ca);
          if (!status(cb)) return(cb);
          if (!status(cc)) return(cc);
          flick();
     }
}
```

Errors

Invalid channel

SYSLOG — Node

SYSLOG enables a process to write a message into the system log file.

Calling Sequence

```
syslog(pid, "desired message");
```

Input Parameters

`pid`	is the process id of the process writing the message into the log file.
`"desired message"`	Maximum message length is 120 characters.

Return Values

None.

Description

The desired message is prefixed with a timestamp and the process id and node id of the originating process when the entry is written into the file. For example:

```
syslog(3,"hello");
```

would have an entry similar to

```
11/15/86 14:32 NODE: 5 PID: 3 : hello
```

written in the system log file.

For more information about the log file, refer to the "cubelog" command in Chapter 2.

Errors

Invalid message pointer	The message to be logged cannot be accessed using the supplied pointer.
Invalid pid	The valid range is 0 - 32767.

CHAPTER 7

FORTRAN ROUTINES

INTRODUCTION

This chapter is a reference section describing each of the routines in the two FORTRAN system interface libraries: cube manager and node. Note that the name of the library is in bold letters on the right hand side of the page; the name of the routine on the left.

To obtain a more general understanding of the routines and how to use them, refer to Chapter 4, Developing FORTRAN Programs in the iPSC Program Development Guide.

The routines are documented in six parts:

- name of the routine and a brief description
- the calling sequence
- the input parameters. Parameters are required, that is, they have no default values. In addition, you must specify them in the exact order presented in the calling sequence.
- the return values or results of executing the routine
- a description (occasionally omitted when not necessary)
- errors that can occur through the use of this routine. All errors cause the process that originated them to stop running and an error message to be written in the system log file. For more information about errors, refer to Chapter 5 in the iPSC Program Development Guide. Errors are displayed and logged in the following format:

  9/9/86 16:59 NODE: xxx PID: zzzz error message

The first four items in an error message are: date, time, node number, and process id. The fifth item, *error message*, has the following format:

Type of error: *xxxx* in *call* at *yyyy:zzzz*

where:

Type of error is the message text specifying the error or exception

xxxx is the value of the invalid parameter, or exception number

call is the name of the system call in which the error occurred

yyyy:zzzz is the memory location, or address, of the error in hexadecimal. It consists of two parts: the code segment (*yyyy*) and the instruction pointer (*zzzz*)

Only *Type of error* is documented for each routine in this chapter.

SUMMARY OF CALLING SEQUENCES

The following tables summarize the calling sequence for each routine in the cube manager and node interface libraries for FORTRAN.

Table 7-1
Calling Summary for FORTRAN
Cube Manager Routines

Procedure	Calling Sequence
CCLOSE	CALL CCLOSE(CI)
COPEN	CI = COPEN(PID)
CUBEDIM	RESULT = CUBEDIM()
LKILL	CALL LKILL(NODE, PID)
LOAD	CALL LOAD('FILENAME', NODE, PID)
LWAIT	CALL LWAIT(NODE, PID, CNODE, CPID, CCODE)
LWAITALL	CALL LWAITALL(NODE, PID)
MYPID	PID = MYPID()
PROBEMSG	RESULT = PROBEMSG(PID)
RECVMSG	CALL RECVMSG(CI, TYPE, BUF, LEN, CNT, NODE, PID)
SENDMSG	CALL SENDMSG(CI, TYPE, BUF, LEN, NODE, PID)
SYSLOG	CALL SYSLOG(PID, 'DESIRED MESSAGE')

Table 7-2
Calling Summary for Node FORTRAN Routines

Procedure	Calling Sequence
CCLOSE	CALL CCLOSE(CI)
CLOCK	RESULT = CLOCK()
COPEN	CI = COPEN(PID)
CUBEDIM	RESULT = CUBEDIM()
FLICK	CALL FLICK
GREENLED	CALL GREENLED(ISTATE)
HANDLER	CALL HANDLER(TYPE, NEWHAND)
MYNODE	RESULT = MYNODE()
MYPID	PID = MYPID()
PROBE	RESULT = PROBE(CI, TYPE)
RECV	CALL RECV(CI, TYPE, BUF, LEN, CNT, NODE, PID)
RECVW	CALL RECVW(CI, TYPE, BUF, LEN, CNT, NODE, PID)
REDLED	CALL REDLED(ISTATE)
SEND	CALL SEND(CI, TYPE, BUF, LEN, NODE, PID)
SENDW	CALL SENDW(CI, TYPE, BUF, LEN, NODE, PID)
STATUS	RESULT = STATUS(CI)
SYSLOG	CALL SYSLOG(PID, 'DESIRED MESSAGE')

CUBE MANAGER SYSTEM INTERFACE LIBRARY

This section describes the FORTRAN language interface to the cube manager library. Refer to Chapter 4, Developing FORTRAN Programs in the iPSC Program Development Guide for examples of sequences of these routines.

FORTRAN processes must conform to the 80286 "large" model of segmentation (which the compiler automatically creates).

NOTE

The FORTRAN compiler assigns INTEGER*4 to any declaration typed as simply INTEGER.

The COPEN, CUBEDIM, and PROBEMSG routines provided in the cube manager interface library are typed INTEGER but have names that usually imply type REAL.

They should, therefore, be explicitly typed in each subroutine which references them.

As an example:

```
PROGRAM MAIN
INTEGER CI, PID
INTEGER COPEN
  .
  .
  .
CI = COPEN(PID)
```

For convenience, these routines are explicitly typed in a file, called "rmfhost.def", which can be accessed with an "include" compiler directive embedded in the source file. If used, you do not have to explicitly type each function each time it is used. "rmfhost.def" file is located in the "/usr/ipsc/lib" directory.

The following program fragment is an example of using the "include" compiler directive.

```
PROGRAM ALPHA
DIMENSION A(100), B(100)
INCLUDE '/usr/ipsc/lib/rmfhost.def'
INTEGER CI, PID
.
.
.
CI = COPEN(PID)
END
SUBROUTINE BETA(CI)
INTEGER X,Y,Z,CI
INCLUDE '/usr/ipsc/lib/rmfhost.def'
.
.
.
X=STATUS(CI)
.
.
.
END
```

CCLOSE — Cube Manager

CCLOSE destroys the specified communication channel created by a previous "copen".

Calling Sequence

```
INTEGER  CI
   .
   .
   .
CALL CCLOSE(CI)
```

Input Parameters

CI	is the channel identifier of the channel you want to close. CI was returned by XENIX as a result of the execution of "copen".

Return Values

None.

Description

Once a process has closed a channel, messages that are addressed to the process will be stored by XENIX until a channel is reopened. The only exception to this occurs when parent and child processes are sharing the same channel. If either the parent or the child process closes its channel, messages are delivered to the process that still has the channel open.

This routine allows you to explicitly close a channel. Any channels not explicitly closed are automatically closed when the process terminates.

Errors

Invalid channel

Invalid cid pointer

COPEN — Cube Manager

COPEN is an integer function which creates a communication channel for a cube manager process.

Calling Sequence

```
INTEGER  CI, PID
INTEGER  COPEN
  .
  .
  .
CI = COPEN(PID)
```

Input Parameters

PID	is the user-defined process id of the process opening the channel. The system uses the pid as a reference in all messages sent via the channel. Valid range is 0 - 32767. It is recommended that all the channels opened by a single process be assigned the same process id.

Return Values

CI	is the channel identifier used by XENIX to manage the transmission and reception of messages.

Description

Once a channel has been established, it may be used to send and receive messages. Messages cannot be sent or received until a channel is opened.

Unlike node processes which can perform simultaneous send and receive operations over multiple channels, a cube manager process can perform only one operation over one channel at a time. This is due to certain characteristics of the XENIX operating system.

The maximum number of channels a cube manager process can have is 18. If you attempt to open more than 18 channels, XENIX will log a "Too many open files" error in the log file.

Errors

Invalid pid	The valid range is 0 - 32767.
Invalid pid pointer	
Out of memory	The necessary structures cannot be created to manage the communication channel.
Too many open files	A process attempted to open more than the maximum of 18 channels.

CUBEDIM — Cube Manager

CUBEDIM is an integer function which returns a value that is the dimension of the cube in which the program is running.

Calling Sequence

```
INTEGER  RESULT
INTEGER  CUBEDIM
.
.
.
RESULT = CUBEDIM()
```

Input Parameters

None.

Return Values

RESULT	is a value from 0 to 7 for applications running in 1 to 128 nodes. For example, "cubedim" returns 6 for a 64-node cube (or 2^6-node cube). This is the dimension you specified in the "dim" argument of the "load" command.

Description

You can use "cubedim" to find out how many nodes an application is running on in order to have the cube manager process send a message to all nodes. A sample statement that would return the number of nodes is:

```
NNODES = 2**CUBEDIM()
```

Errors

None.

LKILL — Cube Manager

LKILL kills the selected process (or processes) in the selected node (or nodes).

Calling Sequence

```
INTEGER NODE
INTEGER PID
  .
  .
CALL LKILL(NODE, PID)
```

Input Parameters

NODE	the number of the node in which the process (or processes) is to be killed. Use -1 to select all nodes.
PID	the id of the process to be killed. Use -1 to select all processes. No error results if you specify a nonexistent process.

Return Values

None.

Description

This routine kills the process specified by "pid" on the node. The pid may match a pid given in a previous load, or it may be -1 to kill all processes. "Node" is the node number. Node numbers may range from 0 to (2**cubedim)-1, or -1 for the special case of all nodes. The cube manager number, -32768, is invalid.

Errors

Invalid node pointer

Invalid pid pointer

Loader: Cube Response Timed Out — Execute "load -c".

Loader: invalid node address <n>

LOAD — Cube Manager

LOAD loads the file into the specified node (or all nodes if -1 is specified for the node number) and starts it.

Calling Sequence

```
INTEGER NODE
INTEGER PID
  .
  .
CALL LOAD('FILENAME', NODE, PID)
```

Input Parameters

'FILENAME'	the pathname of the file to be loaded.
NODE	the number of the node into which the process is to be loaded. Use -1 to select all nodes.
PID	the process id to be assigned to the process once it has been loaded. An error results if you specify a pid that is already in use by a previously loaded process. A pid may be reused as soon as the process completes. Up to 20 processes per node are allowed.

Return Values

None.

Description

Loads the file (a character string containing the pathname of the file) into a node (as specified by its number). Node numbers may range from 0 to (2**cubedim)-1, or -1 for the special case of all nodes. The cube manager number, -32768, is invalid because this routine does not load processes into the cube manager.

The pid is saved by NX for use in identifying the process in other routines. By convention, this should be the same as the pid used for the process in SEND and RECV routines. Use the MYPID routine in your node program to obtain the process id.

Note that, unlike the "load" cube manager command, the LOAD routine does not provide as many options, such as changing stack size and execution timeslice.

LOAD (cont.) — Cube Manager

Errors

Invalid node pointer	
Invalid pid pointer	
Loader: cannot open <file>	
Loader: cannot read <file>	
Loader: Cube Response Timed Out	Execute "load -c".
Loader: duplicate PID on node <n>	
Loader: file too short: <filename>	
Loader: invalid file <filename>	
Loader: invalid node address <n>	
Loader: not enough extended memory on node <n>	
Loader: not enough memory on node <n>	
Loader: too many processes on node <n>	

LWAIT — Cube Manager

LWAIT waits for the selected process in the selected node (or nodes) to complete. It then returns the pid, node, and completion code. If you enter -1 for all processes, LWAIT returns as soon as the ***first*** one completes.

Calling Sequence

```
INTEGER NODE
INTEGER PID
INTEGER CNODE
INTEGER CPID
INTEGER CCODE
  .
  .
CALL LWAIT(NODE, PID, CNODE, CPID, CCODE)
```

Input Parameters

NODE	the number of the node in which the process (or processes) resides. Use -1 to select all nodes.
PID	the process id of the process to wait for. Use -1 to select all processes. An error results if you wait for a process with a specific pid and that pid does not exist.

Return Values

CNODE	is the location where the node number of the completed process is stored.
CPID	is the location where the pid of the completed process is stored.
CCODE	is the location where the completion code of the completed process is stored. The least significant 16 bits of the code give the reason the process finished. The most significant 16 bits give the exit code or additional information. Refer to the next page for a description of completion codes.

Description

Waits for a process to complete on any one of the nodes. The pid is a process id, or -1 for the special case of all processes. Node numbers may range from 0 to (2**cubedim)-1, or -1 for the special case of all nodes. The cube manager number, -32768, is invalid.

When a process with a matching pid and node completes, the node, pid, and completion code of the process are returned in cnode, cpid, and ccode, respectively.

LWAIT (cont.) Cube Manager

Errors

Invalid ccode pointer	
Invalid cnode pointer	
Invalid cpid pointer	
Invalid node pointer	
Invalid pid pointer	
Loader: Cube Response Timed Out	Execute "load -c".
Loader: invalid node address <n>	

COMPLETION CODES

Each process returns two, 16-bit values when it exits or is killed.

First Value

The first value (least significant 16 bits) is a result code which indicates how the process ended:

Result Code	Reason	Result Code	Reason
0	Normal exit	10	Not used
1	Divide error exception	11	Not used
2	Integer overflow exception	12	Not used
3	Bound range exceeded exception	13	Invalid opcode exception
4	General protection fault	14	Processor extension segment overrun interrupt
5	Segment not present fault		
6	Stack segment overrun or not present	15-31	Not used
		32	Single step interrupt
7	Floating point address exception	33	Breakpoint interrupt
8	Not used	34	NMI interrupt
9	Not used	35	Not used
		36	Parameter error

Second Value

The second value (most significant 16 bits) is the exit code which contains additional information, depending on the first value.

If 1st value is 0	Second value contains the exit code (value supplied to exit() in C, or value on STOP statement in FORTRAN).
If 1st value is 7	Second value contains the floating point status word.

LWAITALL — Cube Manager

LWAITALL waits for one or all processes in one or all nodes to complete. If you enter -1 for all processes, LWAITALL, unlike LWAIT, waits for all the processes to complete.

Calling Sequence

```
INTEGER NODE
INTEGER PID
  .
  .
LWAITALL(NODE, PID)
```

Input Parameters

NODE	the number of the node in which the process (or processes) resides. Use -1 to select all nodes.
PID	the process id of the process to wait for. Use -1 to select all processes. Unlike LWAIT it is not an error to wait for a nonexistent process. The call will simply return immediately.

Return Values

None.

Description

Waits for processes specified by "pid" on "node" to complete. The pid is a process id, or -1 for the special case of all processes. Node numbers may range from 0 to (2**cubedim)-1, or -1 for the special case of all nodes. The cube manager number, -32768, is invalid.

The LWAITALL routine does not return until all specified processes on all specified nodes are completed. The completion codes for the processes are discarded.

Errors

Invalid node pointer

Invalid pid pointer

Loader: Cube Response Timed Out — Execute "load -c".

Loader: invalid node address <n>

MYPID

Cube Manager

MYPID returns the process id of the calling process.

Calling Sequence

```
INTEGER PID
  .
  .
PID = MYPID()
```

Input Parameters

None.

Return Values

PID	value of the XENIX process id.

Description

This routine provides the XENIX process id of a cube manager process. Do not confuse this routine with the node MYPID routine which returns the process id given in the "load" command or routine.

Errors

None.

PROBEMSG — Cube Manager

PROBEMSG returns a value indicating whether a message is pending for a cube manager process.

Calling Sequence

```
INTEGER STATUS
INTEGER PID
INTEGER PROBEMSG
  .
  .
RESULT = PROBEMSG(PID)
```

Input Parameters

PID	the process id of the cube manager process for which to check outstanding messages. This is the same process id used when opening a channel. Valid pid range is 0 - 32767. Use pid "-1" to check messages for all cube manager processes.

Return Values

RESULT	one of two values: 0 - if there are no messages for any process (if pid is -1) or for the specified cube manager process. 1 - if there is one or more messages pending for any process (if pid is -1) or for a specified process.

Description

This routine allows you to check on outstanding (waiting to be received) messages for one or all cube manager processes. Although there is some correlation between this routine and the node PROBE routine, their parameters are completely different. For example, "type" is only an issue in node processes, not cube manager processes.

An example of its usage is:

```
IF (PROBEMSG(99) .EQ. 1) THEN
  CALL RECVMSG(CI, TYPE, BUF, LEN, CNT, NODE, PID)
ENDIF
```

Error Messages

Invalid pid	The valid range is -1 through 32767.

RECVMSG — Cube Manager

RECVMSG initiates the receipt of a message from a node process or another cube manager process.

Calling Sequence

```
INTEGER  CI, TYPE, LEN, CNT, NODE, PID
DIMENSION  BUF(*)
  .
  .
  .
CALL RECVMSG(CI, TYPE, BUF, LEN, CNT, NODE, PID)
```

Input Parameters

CI	is the channel identifier that was returned to the process as a result of a previous "copen".
LEN	is the number of bytes in the message buffer pointed to by BUF. You should probably specify LEN to be as large as the biggest message you expect to receive.

Return Values

TYPE	is the type of the message received.
BUF	is the buffer where the received message is to be stored. Can be any data type except character. For information about receiving character strings, refer to Appendix C in the iPSC Program Development Guide.
CNT	is the number of message bytes received. If the value of CNT is less than or equal to the value of LEN, the entire message was received. If the value of CNT is greater than LEN, a message whose length was greater than the size of the message buffer was received. Only the first LEN bytes were stored in the message buffer. The remainder of the message is truncated and cannot be recovered.
NODE	is the node id of the process that sent the message.
PID	is the process id of the process that sent the message.

RECVMSG (cont.) Cube Manager

Description

The RECVMSG routine receives any message whose destination node and process id's match the cube manager process that originated the RECVMSG request.

Unlike node processes, the "type" of message is a returned value and is not used as a reception qualifier.

This routine causes the calling process to be blocked until the message has been received. When the calling process resumes execution, the message that was requested has been received and stored in the user message buffer.

This routine uses nearly the same parameters as the node system routine RECVW. The only difference is that the "type" is returned rather than used for message selection.

Errors

Allocated buffer space less than length	The message buffer, as defined by the buffer pointer (buf) and the buffer length (len) parameters, extends beyond the end of the data segment in which it starts. This means that the entire message buffer cannot be accessed. This error is usually caused by the length parameter being too large.
Invalid buffer length	The length of the message is negative.
Invalid buffer pointer	
Invalid cid pointer	
Invalid channel	
Invalid count pointer	
Invalid length pointer	
Invalid node pointer	
Invalid pid pointer	
Invalid type pointer	

SENDMSG — Cube Manager

SENDMSG initiates the transmission of a message from a cube manager process to a process on the cube manager, on a node, or on all nodes if -1 is specified as the node parameter.

Calling Sequence

```
INTEGER CI, TYPE, LEN, NODE, PID
DIMENSION BUF(*)
  .
  .
  .
CALL SENDMSG(CI, TYPE, BUF, LEN, NODE, PID)
```

Input Parameters

CI	is the channel identifier that was returned to the process as a result of a previous "copen".
TYPE	is an integer value referring to the type of message to be sent. This can be any value in the range of 0 to 32767. Negative values are reserved for use by the operating system.
BUF	is the buffer that contains the message to be sent. Can be any data type except character. For information about sending character strings, refer to Appendix C in the iPSC Program Development Guide.
LEN	is the number of bytes to be transmitted from the buffer. The maximum number of bytes that may be sent in a single message is 16383 (16 kilobytes).
NODE	is the id of the node being sent the message. Valid node id's range from 0 to 127, but may not exceed the value implied by the "dim" parameter in the "load" command. The cube manager's address (-32768) is also valid. Specify this parameter as -1 to send the message to all processes currently executing on the cube whose PID matches the one given in this call.
PID	is the process id of the process being sent the message. Process id's range from 0 to 32767. Negative values are reserved for the operating system.

Return Values

None.

SENDMSG (cont.) Cube Manager

Description

The return to the calling process from the SENDMSG routine implies that the user message buffer is available for reuse. This does not indicate that the message has been received by the destination process.

This routine performs similarly to the node system routine SENDW.

Errors

Allocated buffer space less than length	The message buffer, as defined by the buffer pointer (buf) and the buffer length (len) parameters, extends beyond the end of the data segment in which it starts. This means that the entire message buffer cannot be accessed. This error is usually caused by the length parameter being too large.
Invalid buffer length	the length of the message is negative or is greater than 16,384.
Invalid buffer pointer	
Invalid cid pointer	
Invalid channel	
Invalid length pointer	
Invalid node	the node specified is greater than 127, negative, or not equal to -32768.
Invalid node pointer	
Invalid pid	the valid range is 0 - 32767.
Invalid pid pointer	
Invalid type	the valid range is 0 - 32767.
Invalid type pointer	

SYSLOG — Cube Manager

SYSLOG enables a process to write a message into the system log file.

Calling Sequence

```
INTEGER  PID
  .
  .
  .
CALL SYSLOG(PID, 'DESIRED MESSAGE')
```

Input Parameters

PID	is the process id of the process writing the message into the log file.
'DESIRED MESSAGE'	Maximum message length is 120 characters. Use string constant or character variable.

Return Values

None.

Description

The argument string is prefixed with a timestamp, the node id, and process id of the originating process when the entry is written into the file. For example:

```
CALL SYSLOG(3, 'hello')
```

would produce an entry similar to

```
11/15/86 14:32 NODE: -32768 PID: 3 : hello
```

in the system log file.

For more information about the log file, refer to the "cubelog" command in Chapter 2.

Errors

Invalid message pointer	The message to be logged cannot be accessed using the supplied pointer.
Invalid pid	The valid range is 0 - 32767.
Invalid pid pointer	

NODE SYSTEM INTERFACE LIBRARY

This section describes the FORTRAN language interface to the node system routines. Refer to Chapter 4, Developing FORTRAN Programs in the iPSC Program Development Guide for examples of sequences of these routines.

FORTRAN processes must conform to the 80286 "large" model of segmentation (which the compiler automatically creates).

The following functions provided in the FORTRAN system interface library are typed INTEGER but have names that usually imply type REAL:

CLOCK
COPEN
CUBEDIM
PROBE
STATUS

They should, therefore, be explicitly typed in each subroutine which references them.

As an example:

```
PROGRAM MAIN
INTEGER CI, PID
INTEGER COPEN
.
.
.
CI = COPEN(PID)
```

For convenience, these routines are explicitly typed in a file, called "rmfnode.def", which can be accessed with an "include" compiler directive embedded in the source file. If used, you do not have to explicitly type each function each time it is used. "rmfnode.def" file is located in the "/usr/ipsc/lib" directory.

The following program fragment is an example of using the "include" compiler directive.

```
PROGRAM ALPHA
DIMENSION A(100), B(100)
INCLUDE '/usr/ipsc/lib/rmfnode.def'
INTEGER CI, PID
.
.
.
CI = COPEN(PID)
END
SUBROUTINE BETA(CI)
INTEGER X,Y,Z,CI
INCLUDE '/usr/ipsc/lib/rmfnode.def'
.
.
.
X=STATUS(CI)
.
.
.
END
```

CCLOSE

Node

CCLOSE destroys a specified communication channel created by a previous "copen".

Calling Sequence

```
INTEGER CI
  .
  .
  .
CALL CCLOSE(CI)
```

Input Parameters

CI	is the channel identifier of the channel you want to close. It was returned by the node operating system as a result of the execution of "copen".

Return Values

None.

Description

The CCLOSE routine should only be invoked when the process no longer desires to communicate via the channel. This routine allows you to explicitly close a channel. Any channels not explicitly closed will be closed when the process terminates.

The CCLOSE routine maintains the integrity of any messages awaiting transmission on the channel. That is, if CCLOSE is invoked following a SEND routine, the message is still transmitted.

There are three potential situations that can occur by invoking CCLOSE following a RECV request:

- the message has not yet been received. The channel is closed without waiting for the receipt of the message. If another channel were opened, the message could still be received.
- the message is being received when CCLOSE is invoked. The channel is closed and the message is lost.
- The message has been received when CCLOSE is invoked. The message is received intact with the return values updated and the channel is closed.

It is recommended that all receive operations on a channel be completed prior to closing the channel. This can be done using STATUS.

Errors

Invalid channel

CLOCK — Node

CLOCK is an integer function which returns a value that is the number of elapsed millisecond intervals since the node was initialized, modulo 2^{32}.

Calling Sequence

```
INTEGER RESULT
INTEGER CLOCK
 .
 .
 .
RESULT = CLOCK()
```

Input Parameters

None.

Return Values

RESULT	is a value that is the number of millisecond intervals since the node was initialized, modulo 2^{32}.

Description

The CLOCK routine provides a simple timing mechanism to show how long the node has been running. It is not recommended that the CLOCK routine be used to synchronize different aspects of process execution, because each node has its own "clock" value.

The returned value is updated every 5 milliseconds. That is, every 5 milliseconds, the value is incremented by 5. The clock starts keeping time when the "load -c" command is invoked.

Errors

None.

COPEN — Node

COPEN is an integer function which creates a communication channel for a node process.

Calling Sequence

```
INTEGER CI, PID
INTEGER COPEN
 .
 .
 .
CI = COPEN(PID)
```

Input Parameters

PID	is the process id of the process opening the channel. The pid given is used as a reference in all messages sent via the channel. The valid range is 0 to 32767. Negative process id's are reserved for the operating system. It is recommended that all the channels opened by a single process be assigned the same process id and that this pid agrees with the pid used in the "load" command or routine.

Return Values

CI	is the channel identifier used by the operating system to manage the transmission and reception of messages.

Description

Once a channel has been established, it may be used to send and receive messages. Messages cannot be sent or received until a channel is opened.

Message sending and receiving cannot be performed on the same channel concurrently. If you want to do concurrent sends and receives, open multiple channels. The number of channels you will need is based on the number of concurrent sends and receives you want to perform.

The maximum number of channels a node can have (regardless of how many processes are executing) is approximately 250.

Errors

Invalid pid	The valid range is 0 - 32767.
No more channels available in copen	The application has exceeded the approximate limit of 250 channels.

CUBEDIM — Node

CUBEDIM is an integer function which returns the dimension of the cube in which the process is running.

Calling Sequence

```
INTEGER RESULT
INTEGER CUBEDIM
  .
  .
  .
RESULT = CUBEDIM()
```

Input Parameters

None.

Return Values

RESULT	is a value from 0 to 7 for applications running in 1 to 128 nodes. For example, CUBEDIM returns 6 for a 64-node cube (or a 2^6-node cube). This is the dimension you specified in the "dim" argument of the "load" command.

Description

You can use CUBEDIM to find out how many nodes an application is running on. One example of its usage is when a controlling node is going to receive results from all nodes and needs to know how many nodes that will be. A sample statement that would return the number of nodes is:

```
NNODES = 2**CUBEDIM()
```

Errors

None.

FLICK — Node

FLICK allows a node process to defer execution to another process on the same node.

Calling Sequence

```
CALL FLICK
```

Input Parameters

None.

Return Values

None.

Description

The scheduler runs processes on any node in round-robin fashion, running each process for approximately 50 milliseconds (this can be changed using the -T switch on the "load" command), or until the process explicitly relinquishes the processor with a FLICK routine. The execution of the process is not deferred indefinitely. Instead, the process will resume execution during its next scheduled time frame. FLICK simply delays execution to allow other node processes to execute.

You can use the FLICK routine when a process is waiting for the change in status on one or more channels, and has no other work to do. A node process should never "busy loop" looking at the status of a channel that has a pending RECV request. The process sending the desired message may be running on the same node, but be suspended waiting for the processor.

FLICK is a delay mechanism, not a synchronization mechanism. The scheduler does not know what conditions a process is waiting for. For example, a process may be waiting for the completion of several concurrent message operations.

The following examples show a correct and incorrect usage of FLICK.

FLICK (cont.) Node

Example (1) - Correct Usage

```
10 IF (STATUS(CI).EQ.1) THEN
       CALL FLICK
       GO TO 10
   ENDIF
```

Example (2) - Incorrect Usage

```
IF (STATUS(CI).EQ.1) CALL FLICK
```

In Example 2, control may be passed to the statements following the "if" statement before the status of the channel has changed.

Errors

None.

GREENLED

Node

Allows the process to turn its node board's green LED on or off.

Calling Sequence

```
INTEGER ISTATE
   .
   .
CALL GREENLED(ISTATE)
```

Input Parameters

ISTATE	is a value: either 0 (for off) or 1 (for on).

Return Values

None.

Description

The node board's green LED can be turned either on or off depending on the value of "ISTATE". Results are not predicted when "ISTATE" is not 0 or 1.

Error Messages

None.

HANDLER — Node

HANDLER allows a process to assign the execution of a user-written exception handling routine on the occurrence of a specific exception. A process can link exception handlers that are effective only for exceptions caused by the calling process.

Calling Sequence

```
INTEGER TYPE
EXTERNAL NEWHAND
  .
  .
  .
CALL HANDLER(TYPE, NEWHAND)
```

Input Parameters

TYPE	is the type code of the hardware exception. Values in the range 0 to 16, except for 7, 8, 10, 12, 14, and 15, are valid. Valid type codes are described in Table 4-3.
NEWHAND	is the name of the user-written exception handler.

Return Values

None.

Description

The interface between NX and user-written exception handlers is the same as the interface for exception handlers that are called through trap gates and execute on bare hardware (as defined in the iAPX 286 Programmer's Reference manual). The state of the interrupted procedure is saved on the stack. Certain exceptions also cause an error code to be pushed on the stack. The handler must return with an IRET (Return from Interrupt or Nested Task) instruction which returns control to an interrupted routine. (IRET is an assembly language instruction for the 286.)

Refer to the iAPX 286 Programmer's Reference Manual for further information regarding exception handling on the 80286.

Errors

Invalid type	Incorrectly specified the exception type parameter.

HANDLER (cont.) Node

Table 7-3
Hardware Exception Type Codes

Type Code	Exception Name	Description
0	Divide error	Occurs if the quotient is too large or an attempt is made to divide by zero.
1	Single step	Allows programs to execute one instruction at a time. If the TF bit in the flag word is set, an interrupt occurs after the next instruction has been executed.
2	NMI interrupt	Occurs upon receipt of an external signal on the NMI pin.
3	Breakpoint interrupt	Occurs when the 1-byte breakpoint instruction (INT 3) is executed.
4	INTO detected overflow	Occurs when the INTO instruction is executed if the overflow bit of the FLAGS register is set.
5	Bounds range exceeded	Occurs when the BOUND instruction is executed if the specified array index is found to be invalid with respect to the given array bounds.
6	Invalid opcode	Occurs if execution of an invalid opcode is attempted. Can also occur if the effective address given by certain instructions, notably BOUND, LDS, LES, and LIDT, specifies a register rather than a memory location.
9	Processor extension segment overrun	Occurs if a processor extension memory operand does not fit in a segment.
11	Segment or call gate not present	Occurs when an attempt is made to load a not-present segment or to use a control descriptor that is marked not-present.
13	General protection fault	Occurs when a violation occurs which is not covered by another interrupt.
16	Floating point	Occurs after the numeric instruction that caused the error. It can only occur while executing a subsequent WAIT or ESC.

MYNODE

Node

MYNODE is an integer function which returns the node ID of the calling process.

Calling Sequence

```
INTEGER RESULT
  .
  .
  .
RESULT = MYNODE()
```

Input Parameters

None.

Return Values

RESULT	is the node id of the process that initiated the routine.

Errors

None.

MYPID — Node

MYPID returns the process id of the calling process.

Calling Sequence

```
INTEGER PID
  .
  .
PID = MYPID()
```

Input Parameters

None.

Return Values

PID	value of pid supplied in the cube manager "load" command or routine.

Description

Some examples of this routine's usage are:

```
CALL SYSLOG(MYPID(), 'DESIRED MESSAGE')

CI = COPEN(MYPID())
```

Errors

None.

PROBE — Node

PROBE is an integer function which allows a node process to determine whether a message of a given type is available for reception.

Calling Sequence

```
INTEGER RESULT, CI, TYPE
INTEGER PROBE
 .
 .
 .
RESULT = PROBE(CI, TYPE)
```

Input Parameters

CI	is a channel identifier returned to the process as a result of a previous "copen". ***Any*** open channel can be used. For example, if you have opened four channels, use any one of them to probe all four.
TYPE	is an integer value referring to the type of message. This can be any value in the range of 0 to 32767. Negative values are reserved for use by the operating system.

Return Values

RESULT is one of two sets of values:

- If a message of the specific type is found, the value returned equals the length (in bytes) of the pending message. This can be any value in the range 0 to 16383. If there are multiple messages of the same type, "probe" returns the length of the message received first.
- If there are no messages, a negative value is returned.

Description

This routine "probes" all pending messages for a process and returns a positive or negative value based on whether or not any messages of a given type are available for reception by a process. The "ci" in this routine is not used the same as the "ci" in send and receive operations. "Probe" tests messages based on type only, regardless of the channel the messages will be received on. "Probe" does not block or wait for a previous operation to complete on the specified channel.

PROBE (cont.)

Errors

Invalid channel	
Invalid type	The valid range is 0 - 32767.

RECV — Node

RECV initiates the receipt of a message from another process. You must do a "status" call to determine when the message is received. When "status" returns 0, the message has been received and the return values have been updated. You should not use the buffer for anything else until "status" returns 0.

Calling Sequence

```
INTEGER CI, TYPE, LEN, CNT, NODE, PID
DIMENSION BUF(*)
 .
 .
 .
CALL RECV(CI, TYPE, BUF, LEN, CNT, NODE, PID)
```

Input Parameters

CI	is the channel identifier that was returned to the process as a result of a previous "copen".
TYPE	is an integer value referring to the type of message to be received. The type field is the only qualifier you can use to select a message to be received. You cannot select a message to be received based on the originator's process or node id. TYPE can be any value in the range of 0 to 32767. Negative values are reserved for use by the operating system.
LEN	is the number of bytes in (length of) the message buffer pointed to by BUF.

Return Values

BUF	is the buffer where the received message is to be stored. Can be any data type except character. For information about receiving character strings, refer to Appendix C in the iPSC Program Development Guide.
CNT	is the location where the number of message bytes received is stored. If the value of CNT is less than or equal to the value of LEN, the entire message was received. A value of CNT greater than LEN indicates a message was received whose length was greater than the size of the message buffer. Only the first LEN bytes were stored in the message buffer. The remainder of the message was truncated and cannot be recovered.
NODE	is the location where the node id of the process that sent the message is stored.
PID	is the location where the process id of the process that sent the message is stored.

RECV (cont.) Node

Description

The return to the calling process from the RECV routine does not indicate that the message has been received by the requesting process. It does, however, mean that the request to receive the message has been logged by the operating system.

The calling process can determine when the message has been received by the execution of the STATUS function. Until STATUS indicates that the request has been completed, the process cannot rely on the validity of the contents of the message buffer specified to receive the message.

If the calling process invokes RECV prior to the completion of a previous send or receive request on the channel, execution of the routine is delayed until the previous request is completed.

Errors

Allocated buffer space less than length	The message buffer, as defined by the buffer pointer (buf) and the buffer length (len) parameters, extends beyond the end of the data segment in which it starts. This means that the entire message buffer cannot be accessed. This error is usually caused by the length parameter being too large.
Invalid buffer length	
Invalid buffer pointer	
Invalid channel	
Invalid count pointer	
Invalid node pointer	
Invalid pid pointer	
Invalid type	

RECVW — Node

RECVW initiates the receipt of a message from another process. Execution of this routine causes the calling process to be blocked until the message has been received.

Calling Sequence

```
INTEGER CI, TYPE, LEN, CNT, NODE, PID
DIMENSION BUF(*)
  .
  .
  .
CALL RECVW(CI, TYPE, BUF, LEN, CNT, NODE, PID)
```

Input Parameters

CI	is the channel identifier that was returned to the process as a result of a previous "copen".
TYPE	is an integer value referring to the type of message to be received. The type field is the only qualifier you can use to select a message to be received. You cannot select a message to be received based on the originator's process or node id. TYPE can be any value in the range of 0 to 32767. Negative values are reserved for use by the operating system.
LEN	is the number of bytes in (length of) the buffer pointed to by BUF.

Return Values

BUF	is the buffer where the received message is to be stored. Can be any data type except character. For information about receiving character strings, refer to Appendix C in the iPSC Program Development Guide.
CNT	is the number of message bytes received. If the value of CNT is less than or equal to the value of LEN, the entire message was received. A value of CNT greater than LEN indicates a message was received whose length is greater than the size of the message buffer. Only LEN bytes were stored in the message buffer. The remainder of the message was truncated and cannot be recovered.
NODE	is the location where the node id of the process that sent the message is stored.
PID	is the location where the process id of the process that sent the message is stored.

RECVW (cont.) Node

Description

The execution of the RECVW routine causes the calling process to be blocked until the message has been received. When the calling process resumes execution, the message that was requested has been received and stored in the user message buffer and all return values are updated.

If the calling process invokes RECVW prior to the completion of a previous send or receive request on the channel, execution of the routine is delayed until the previous request is completed.

Errors

Allocated buffer space less than length	The message buffer, as defined by the buffer pointer (buf) and the buffer length (len) parameters, extends beyond the end of the data segment in which it starts. This means that the entire message buffer cannot be accessed. This error is usually caused by the length parameter being too large.
Invalid buffer length	
Invalid buffer pointer	
Invalid channel	
Invalid count pointer	
Invalid node pointer	
Invalid pid pointer	
Invalid type	
Out of system buffers	See **-b** switch on "load" command.

REDLED — Node

Allows the process to turn its node board's red LED on or off.

Calling Sequence

```
INTEGER ISTATE
  .
  .
CALL REDLED(ISTATE)
```

Input Parameters

ISTATE	is a value: either 0 (for off) or 1 (for on).

Return Values

None.

Description

The node board's red LED can be turned either on or off depending on the value of "ISTATE". Results are not predicted when "ISTATE" is not 0 or 1.

Error Messages

None.

SEND — Node

SEND initiates the transmission of a message to another process or to all other processes if -1 is specified as the node parameter. The user message buffer should not be rewritten until a "status" indicates that the send operation is complete.

Calling Sequence

```
INTEGER CI, TYPE, LEN, NODE, PID
DIMENSION BUF(*)
 .
 .
 .
CALL SEND(CI, TYPE, BUF, LEN, NODE, PID)
```

Input Parameters

CI	is the channel identifier that was returned to the process as a result of a previous "copen".
TYPE	is an integer value referring to the type of message to be sent. This can be any value in the range of 0 to 32767. Negative values are reserved for use by the operating system.
BUF	is the buffer that contains the message to be sent. Can be any data type except character. For information about sending character strings, refer to Appendix C in the iPSC Program Development Guide.
LEN	is the number of bytes to be transmitted from the buffer. The maximum number of bytes that may be sent in a single message is 16383 (16 kilobytes).
NODE	is the id of the node being sent the message. Valid node id's range from 0 to 127, but may not exceed the value implied by the "dim" parameter in the "load" command. The cube manager's id (-32768) is also valid. Specify this parameter as -1 to send the message to all processes currently executing on the cube whose PID matches the one given in this call.
PID	is the process id of the process being sent the message. Process id's range from 0 to 32767. Negative values are reserved for operating system functions.

Return Values

None.

SEND (cont.) Node

Description

The return to the calling process from the "send" routine does not imply that the message has been sent, only that the system has been notified that a message is ready for transmission. Therefore, **the contents of the message buffer should not be modified until a subsequent "status" operation indicates that the send is complete.** A successful send does not imply reception of the message by the destination process only that the message has been successfully sent.

If the calling process invokes "send" prior to the completion of a previous message request on the channel, execution of the routine is delayed until the previous request is completed.

The "global" send capability can be used to send messages to a surrounding subcube of dimension *d* (less than or equal to cubedim()) by specifying (d - cubedim() - 1) as the node parameter. For example, in a D5 cube, to have node 10 send a message to a surrounding subcube of dimension 2, the node parameter would be calculated as:

2 - 5 - 1 = -4

The call would be:

```
CALL SEND(CI, TYPE, BUF, LEN, -4, PID)
```

This would send the message to the surrounding 3 nodes (since d = 2, 4 nodes are involved). The message would be sent to nodes 8, 9, and 11.

In the node parameter calculation, *d* could be set up as a variable that is decremented, in which case, *d* could be 0 making the subcube dimension 0. If this occurs, no messages are sent to any nodes, and the process continues executing.

Subcubes are groups of contiguously numbered nodes. For example, a subcube of dimension 3 has eight nodes; nodes 0, 8, 16, 32, etc. are the "boundary" nodes. A subcube of dimension 2 has four nodes; nodes 0, 4, 8, 12, 16, etc. are the boundary nodes. If node 5 sends a message to a dimension 2 subcube, nodes 4 through 7 will receive the message since node 5 is in that group of contiguous nodes. For more information, refer to Tables 5-5 through 5-8 in the iPSC System Overview Manual.

SEND (cont.) Node

Errors

Allocated buffer space less than length	The message buffer, as defined by the buf and len parameters, extends beyond the end of the data segment in which it starts. This means that the entire message buffer cannot be accessed. This error is usually caused by the length parameter being too large.
Invalid buffer length	The message length is greater than 16,384.
Invalid buffer pointer	
Invalid channel	
Invalid node	The specified node is greater than 127, negative, or not equal to -32768.
Invalid pid	The valid range is 0 - 32767.
Invalid type	The valid range is 0 - 32767.

SENDW — Node

SENDW initiates the transmission of a message to another process or to all other processes if -1 is specified as the node parameter. Execution of this routine causes the calling process to be blocked until the message has been sent.

Calling Sequence

```
INTEGER CI, TYPE, LEN, NODE, PID
DIMENSION BUF(*)
 .
 .
 .
CALL SENDW(CI, TYPE, BUF, LEN, NODE, PID)
```

Input Parameters

CI	is the channel identifier that was returned to the process as a result of a previous "copen".
TYPE	is an integer value referring to the type of message to be sent. This can be any value in the range of 0 to 32767. Negative values are reserved for use by the operating system.
BUF	is the buffer that contains the message to be sent. Can be any data type except character. For information about sending character strings, refer to Appendix C in the iPSC Program Development Guide.
LEN	is the number of bytes to be transmitted from the buffer. The maximum number of bytes that may be sent in a single message is 16383 (16 kilobytes).
NODE	is the id of the node being sent the message. Valid node id's range from 0 to 127, but may not exceed the value implied by the "dim" parameter in the "load" command. The cube manager's id (-32768) is also valid. Specify this parameter as -1 to send the message to all processes currently executing on the cube whose PID matches the one given in this call.
PID	is the process id of the process being sent the message. Process id's range from 0 to 32767. Negative values are reserved for operating system functions.

Return Values

None.

SENDW (cont.) Node

Description

The return to the calling process from the "sendw" routine does not imply that the message has been received by the destination process. It does, however, mean that the user message buffer is available for reuse.

If the calling process invokes "sendw" prior to the completion of a previous message request on the channel, execution of the routine is delayed until the previous request is completed.

The "global" send capability can be used to send messages to a surrounding subcube of dimension *d* (less than or equal to cubedim()) by specifying (d - cubedim() - 1) as the node parameter. For example, in a D5 cube, to have node 10 send a message to a surrounding subcube of dimension 2, the node parameter would be calculated as:

```
2 - 5 - 1 = -4
```

The call would be:

```
CALL SENDW(CI, TYPE, BUF, LEN, -4, PID)
```

This would send the message to the surrounding 3 nodes (since d = 2, 4 nodes are involved). The message would be sent to nodes 8, 9, and 11.

In the node parameter calculation, *d* could be set up as a variable that is decremented, in which case, *d* could be 0 making the subcube dimension 0. If this occurs, no messages are sent to any nodes, and the process continues executing.

Subcubes are groups of contiguously numbered nodes. For example, a subcube of dimension 3 has eight nodes; nodes 0, 8, 16, 32, etc. are the "boundary" nodes. A subcube of dimension 2 has four nodes; nodes 0, 4, 8, 12, 16, etc. are the boundary nodes. If node 5 sends a message to a dimension 2 subcube, nodes 4 through 7 will receive the message since node 5 is in that group of contiguous nodes. For more information, refer to Tables 5-5 through 5-8 in the iPSC System Overview Manual.

SENDW (cont.) Node

Errors

Allocated buffer space less than length	The message buffer, as defined by the buffer pointer (buf) and the buffer length (len) parameters, extends beyond the end of the data segment in which it starts. This means that the entire message buffer cannot be accessed. This error is usually caused by the length parameter being too large.
Invalid buffer length	The message length is greater than 16,384.
Invalid buffer pointer	
Invalid channel	
Invalid node	The specified node is greater than 127, negative, or not equal to -32768.
Invalid pid	The valid range is 0 - 32767.
Invalid type	The valid range is 0 - 32767.

STATUS — Node

STATUS is an integer function which allows a process to determine the state of the user message buffer associated with sending or receiving a message.

Calling Sequence

```
INTEGER RESULT, CI
INTEGER STATUS
 .
 .
 .
RESULT = STATUS(CI)
```

Input Parameters

CI	is the channel identifier that was returned to the process as a result of a previous "copen".

Return Values

RESULT	returns either a 0 or a 1 and their meanings are as follows:

- Following a "send" request, a 0 indicates that the user message buffer is available for reuse. Following a "recv" request, a 0 indicates that the buffer contains the message.
- 1 indicates that the user message buffer is currently being read from or written to and is **not** available for reuse.

Errors

Invalid channel

SYSLOG — Node

SYSLOG enables a process to write a message into the system log file.

Calling Sequence

```
INTEGER  PID
  .
  .
  .
CALL SYSLOG(PID, 'DESIRED MESSAGE')
```

Input Parameters

PID	is the process id of the process writing the message into the log file.
'DESIRED MESSAGE'	Maximum message length is 120 characters. Use string constant or character variable.

Return Values

None.

Description

The desired message is prefixed with a timestamp and the process id and node id of the originating process when the entry is written into the file. For example:

```
CALL SYSLOG(3,'hello')
```

would produce an entry similar to

```
11/15/86 14:32 NODE: 5 PID: 3 : hello
```

in the system log file.

For more information about the log file, refer to the "cubelog" command in Chapter 2.

Errors

Invalid message pointer	The message to be logged cannot be accessed using the supplied pointer.
Invalid pid	The valid range is 0 - 32767.

GLOSSARY

Note

The definitions in this glossary indicate the way these terms are used in reference to Intel's iPSC system.

aggregate memory bandwidth

The number of memory read and write operations possible in a one second time frame. In a distributed memory system, the effective memory bandwidth (data access rate) increases with the number of processor/memory elements. Aggregate memory bandwidth is, therefore, the sum total of all processor/memory elements working simultaneously. In shared memory systems, the memory element is a bottleneck to higher performance.

application

One or more processes executing on the system.

binary n-cube

See "*hypercube.*"

boolean hypercube

See "*hypercube.*"

central processing unit

The portion of a computer system that contains the arithmetic logic unit (ALU) and the control unit. The ALU performs arithmetic and logical calculations. The control unit contains the sequencing and processing controls for instruction execution, interrupt control, dynamic address translation, and other control and processing functions.

centralized control

A control structure in which work is passed out to individual processors. The central point of control is often the factor that limits system performance. (Contrasted to "distributed control.")

commercial VLSI

Very Large Scale Integration that is a commercially available product, contrasted to a *custom* or *semi-custom* product. This means that there are many users of the product, the product is fully supported with software, documentation, etc. It also means that the product is less expensive than custom products because of the economies of producing the product in large volumes. Intel's 80286 and 80287 are representative commercial VLSI products. This type of product is also referred to as *"off the shelf" VLSI.*

communication bandwidth

The maximum data rate which can be sustained by the CPU and the I/O channels of a node. In the iPSC, the aggregate communication bandwidth is the communication bandwidth of a node multiplied by the total number of nodes in the cube.

communication latency

Time required for a message to pass from a sending node to a destination node and be acknowledged by the receiver.

concurrency

Concurrency is used to denote independent operation of a collection of simultaneous computing activities, as opposed to the lockstep connotations of the familiar term, *"parallel."* Concurrency might also be thought of as *"interactive parallelism."* Concurrency allows asynchronous operation of processors in a multiprocessor system. (Contrasted to "parallelism.")

connected network

Any network in which nodes are directly connected by communication channels to other nodes. (Contrasted to "switched network.")

cosmic cube

The name given the Caltech hypercube developed by Geoffry Fox and Chuck Seitz. See *"hypercube."*

cube

In Intel's iPSC system, an ensemble of microcomputers connected by high-speed communication channels into a concurrent architecture (hypercube) thus providing a multiprocessor computational element which can be configured in 32, 64, or 128-node versions.

cube manager

An Intel System 310AP microcomputer with attached alphanumeric processor. In the Intel iPSC system, it serves as the link between the cube and the external environment.

distributed control

A control structure in which each processor is independent and operates asynchronously, dealing with a limited portion of the computational task. This type of control minimizes the bottlenecks common in centralized control systems and allows linear growth in both performance and capacity. (Contrasted to "centralized control.")

distributed memory

Each processor in the system has its own dedicated local memory. This allows each processor to work on a small portion of the overall computational problem, thus distributing the load. Because each memory is independent, aggregate memory bandwidth increases proportionally with cube size. Also referred to as "*local memory.*"

heterogeneous parallel system

A multiprocessor system consisting of any number of ***different*** types of processors. (Contrasted to "homogeneous" parallel system.)

homogeneous parallel system

A multiprocessor system consisting of any number of ***identical*** processors (nodes). Typically, each node is a complete computer with its own arithmetic unit and its own memory. (Contrasted to "heterogeneous" parallel system.)

hypercube

An interconnection scheme for connecting a collection of microcomputers (*nodes*) together. This is a *d*-dimensional cube where "*d*" represents the number of directly-connected nodes which establish the cube's dimension. A 32-node cube is a 5-dimensional (2^5) system with each node connected to its five nearest neighbors. A 64-node cube is a 6-dimensional (2^6) structure and a 128-node cube is a 7-dimensional (2^7) structure. Each node communicates with its neighbors by queued message passing.

The hypercube is also referred to as "*binary hypercube,*" "*binary n-cube,*" and "*boolean hypercube.*"

iPSC

An acronym for "Intel Personal Supercomputer." This is an MIMD machine implemented by a loosely-coupled, distributed memory, multiprocessor architecture using a hypercube interconnect topology. A message-based operating system resident in each processor (node) provides distributed control and interprocessor communication.

local memory

The memory contained in each node of the cube. See "*distributed memory.*"

loosely-coupled control — A control scheme in which many processors function independently. (Contrasted to "tightly-coupled.")

message — The method of communicating between nodes. The message contains the address of the sending processor (source), the address of the receiving processor (destination), the type indicator, the data, and other pertinent information.

message-based operating system — Software that controls all message passing functions ...between processes running on the same node, between processes running on different nodes, and between node processes and cube manager processes. Allows sending/receiving to be either synchronous or asynchronous and provides reliable message delivery to the nearest neighbor. Messages are routed from node to node, if needed, to reach the final destination.

message passing — A form of communication between concurrent processes that allows a process to send data, or other types of information, to another process and allows the recipient process to receive the message. This type of scheme allows processors to work on demand in an asynchronous fashion.

MFLOPS — Million floating-point operations per second.

MIPS — Million instructions per second.

multiple instruction multiple data (MIMD) machines — An architecture in which independent computing elements work concurrently on portions of a problem. The computing problem is partitioned so that the load is shared by multiple processors operating concurrently on different data sets.

multiprocessor — A computer system containing two or more processing units under integrated control.

node

1. A small but complete individual microcomputer that is one of many identical microcomputers in a concurrent processing machine. Each node contains both a processor and memory.

2. One of the microcomputers in the iPSC ***cube***. A node is an independent board-level processor supporting 32-, 64-, and 80-bit floating-point formats with a high performance version of the 80286/80287 microcomputer chip set. A node also contains 512 Kbytes of RAM as well as a 64 K ROM with initialization and self-test firmware. Eight independent communications coprocessors are also on each board.

node operating system

The operating system for a node. It consists of software that resides in each iPSC node and controls such activities as message passing, memory management, process scheduling, error detection, and initialization.

numeric processing unit (NPU)

A hardware component (Intel's 80287) that works in conjunction with the CPU (Intel's 80286) by extending the numeric processing capabilities of the system. It can typically handle 8-, 16-, 32-, 64- and 80-bit integer and floating-point data types.

off-the-shelf VLSI

See *"commercial VLSI."*

parallelism

Parallelism, as a somewhat general term, is defined as "objects structured in an ordered fashion." Parallelism, as used in computer technology, can have different meanings as explained by the following hierarchy:

Pipelined Parallelism	is exhibited in the **micro** architecture of a machine such as the multi-staged ***operations*** which are in a 5-stage pipelined multiplier.
Functional Parallelism	is exhibited in the **macro** architecture of a machine through the use of multiple functional ***units*** such as a processor having several arithmetic units, or an adder and multiplier which can function on different data simultaneously.
Processer Parallelism	is exhibited in the **global** architecture of a machine through the use of multiple ***processors*** operating simultaneously on a given computational problem(s).

pipelining

A method of "time" parallelism which achieves high performance at the cost of programming complexity. For example, a multiplier which takes three machine cycles (multiply the mantissas, add the exponents, normalize the result) to compute a result can be sped up by a factor of 3x if pipelining is used. Pipelining allows a new set of arguments to be introduced into the multiplier with each cycle while requiring the full three cycles to get the first product out. A new product then appears each cycle thereafter.

process

The basic unit of computation in the iPSC. It is basically a portion of a sequential program (such as FORTRAN or C) that includes statements that send and receive messages.

scalar

A quantity characterized by a single number. This can often be the limiting factor in the optimum performance of vector supercomputers (contrasted with "vector").

shared memory

A common memory that is used by all processors in a multiprocessor system. Shared memory often becomes the bottleneck in a multiprocessor system because of simultaneous demands from many processors. This limits expansion performance of the system and requires high performance memory components.

supercomputer

The fastest processor available at any point in time.

Generally, a computer that performs hundreds of millions of floating-point operations each second, that has a memory capable of storing millions of words, and that handles data words 64 bits long.

switched network

A network in which connections between nodes/processors/ memory are established by a switching network (contrasted to "connected network").

tightly-coupled control

A control scheme in which a few high-performance processors collaborate on a common task...usually through a shared memory. (Contrasted to "loosely-coupled.")

topology

The interconnection network in a multiprocessor system which allows communication between the individual processors within the system.

vector

A quantity usually represented by an ordered set of numbers. Can be multi-dimensional as in the form of arrays. (Contrasted with "scalar").

vector-scalar ratio

Scientific computational problems are characterized by the large number of computations performed on vectors...or "arrays of data." This characteristic is used to advantage in vector supercomputers to achieve high levels of computational performance.

However, in any problem, these vector calculations are mixed with scalar calculations which cannot be executed efficiently on vector hardware. Thus, the vector/scalar ratio determines the maximum performance of a vector machine. Infinite increases in vector performance will have limited impact on the overall performance because of this "scalar bottleneck."

The iPSC is a scalar machine which uses the concurrency of multiple scalar processing elements to achieve high performance. Because vector and scalar computations are treated equally, increasing the number of processing elements has a proportional effect on increased performance.

Von Neuman architecture

A computer design that is based on a single central processor performing operations sequentially. Operations are performed serially, one at a time.

- A -

- B -

- C -

- D -

- E -

For more information on the iPSC, write to:

Intel Scientific Computers
15201 N.W. Greenbrier Parkway
Beaverton, OR 97006

or Call:
(503) 629-7629