April 1991
Order Number: 311532-007

iPSC®/2 AND iPSC®/860
USER'S GUIDE

intel® Corporation

REV.	REVISION HISTORY	DATE
-001	Original Issue	12/87
-002	Revision	03/88
-003	Revision	03/89
-004	Revision	10/89
-005	Preliminary	12/89
-006	Revision	06/90
---	Revised by Change Notice 312003-001	10/90
-007	Revision	04/91

RESTRICTED RIGHTS

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

PREFACE

This manual tells how to use the following Intel Supercomputer Systems Division (SSD) products: iPSC®/2, iPSC®/2S, iPSC®/860, iPSC®/860S, and iPSC®/860Plus.

NOTE

This manual uses the term "iPSC system(s)" to refer to these products.

This manual assumes that you are an application programmer proficient in the C and Fortran languages and the UNIX operating system. The manual provides you with enough detail to begin using your iPSC system.

ORGANIZATION

Chapter 1	Provides an overview of the iPSC system architecture, describes some typical configurations, introduces the iPSC system software that runs on the host and nodes, and summarizes the iPSC system commands and system calls.
Chapter 2	Describes the iPSC system commands that you can enter at the keyboard. It tells how to allocate a cube, load it with one or more node programs, and then deallocate the cube.
Chapter 3	Describes the system calls available to host and node programs. It contains a detailed section on message-passing.
Chapter 4	Tells how to prepare an application for the iPSC system. The steps described are applicable to applications that are written for a parallel computer and applications that are ported from a sequential computer. This chapter discusses three examples: an integration, a matrix*vector multiplication, and the N-Queens problem.

Chapter 5	Describes ways to improve the performance of parallel programs.
Chapter 6	Describes the concurrent file system used by the nodes. This file system resides on a number of disks that connect to the cube through the SCSI interface on I/O nodes.
Chapter 7	Describes how to use TCP/IP software to communicate between a program on a remote host and a node program.
Chapter 8	Describes how a node program can use Xlib calls and access an X Window server running on a remote host.
Appendix A	Summarizes iPSC commands and system calls (both C and Fortran versions) and applicable UNIX system calls.
Appendix B	Lists iPSC features.
Appendix C	Lists iPSC specifications.
Appendix D	Summarizes the node TCP/IP system calls.

NOTATIONAL CONVENTIONS

This manual uses the following notational conventions:

Bold Identifies command names and switches, system call names, reserved words, and other items that must be used exactly as shown.

Italic Identifies variables, filenames, directories, processes, user names, and writer annotations in examples. Italic type style is also occasionally used to emphasize a word or phrase.

`Plain-Monospace`
Identifies computer output (prompts and messages), examples, and values of variables. Some examples contain annotations that describe specific parts of the example. These annotations (which are not part of the example code or session) appear in *italic* type style and flush with the right margin.

Bold-Italic-Monospace
Identifies user input (what you enter in response to some prompt).

`Bold-Monospace`
Identifies the names of keyboard keys (which are also enclosed in angle brackets). A dash indicates that the key preceding the dash is to be held down *while* the key following the dash is pressed. For example:

`<Break>` **`<s>`** **`<Ctrl-Alt-Del>`**

[ ]	(Brackets) Surround optional items.
. . .	(Ellipsis dots) Indicate that the preceding item may be repeated.
|	(Bar) Separates two or more items of which you may select only one.
{ }	(Braces) Surround two or more items of which you must select one.

APPLICABLE DOCUMENTS

For more information, refer to the following manuals:

iPSC® System Manuals

iPSC®/2 and iPSC®/860 FORGE User's Guide
Tells how to use the FORGE tool set to analyze Fortran programs and to port them to a parallel machine.

iPSC®/2 and iPSC®/860 Hardware Installation Manual
Describes installation and powering up of all iPSC system configurations.

iPSC®/2 and iPSC®/860 Interactive Parallel Debugger Manual
Tells how to use the iPSC Interactive Parallel Debugger and provides command reference information.

iPSC®/2 and iPSC®/860 Math Libraries Reference Manual
Describes the math libraries available on the iPSC system.

iPSC®/2 and iPSC®/860 Network Queueing System Manual
Describes and tells how to use the Network Queueing System (NQS) software to queue and manage batch/device processes in the iPSC system.

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

iPSC®/2 and iPSC®/860 ProSolver-DES User's Guide
Tells how to use the ProSolver-DES software to solve very large matrices.

iPSC®/2 and iPSC®/860 ProSolver-SES User's Guide
Tells how to use the ProSolver-SES software to solve sparse matrices.

iPSC®/2 and iPSC®/860 Site Preparation Guide
Tells the customer how to prepare a site for the installation of an iPSC system.

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

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

iPSC®/2 and iPSC®/860 VME Interface Reference Manual
Describes the installation and development of software drivers for the VME Interface Adapter board.

iPSC®/2 Ada Program Development Guide
Describes and tells how to use the tools for developing Ada programs for the iPSC/2.

iPSC®/2 Ada Program Development Guide Change Notice
Adds information on how to use the Ada cross-debugger for developing Ada programs for the iPSC/2 system.

iPSC®/2 Ada Programmer's Reference Manual
Describes all Ada routines and commands for the iPSC/2 system.

iPSC®/2 Fortran Language Reference Manual
Describes the Green Hills Fortran compiler for the iPSC/2 system.

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

iPSC®/2-VX User's Guide
Describes development of programs for the iPSC/2-VX vector processing system.

iPSC®/860 Basic Math Library User's Guide
Describes the math library routines (including BLAS and FFT routines) for the iPSC/860 systems.

iPSC®/860 C Compiler User's Guide
Tells how to use the iPSC/860 C compiler driver.

iPSC®/860 Fortran Compiler User's Guide
Tells how to use the iPSC/860 Fortran compiler driver.

iPSC®/860 Parallel Performance Analysis Tool Manual
Describes and tells how to use the Performance Analysis Tools (PAT) utilities to capture and analyze execution, communication, and event data on the iPSC/860 system.

Intel® Manuals

UNIX System V Manual Set
Describes UNIX System V.

i860™ Manual Set
Describes the i860 microprocessor.

SYP301 Installation and User's Guide
Tells how to install and start the System Resource Manager. Also provides hardware technical data.

Other Manuals

C: A Reference Manual - Harbison and Steele
Describes the C programming language.

Reference Manual For The Ada Programming Language - ANSI/MIL-STD-1815A-1983
Describes the Ada programming language.

The C Programming Language - Kernighan and Ritchie
Describes the C programming language.

The X Window System Manual Set
Describes the X Windows System application programming.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

CHAPTER 2
USING iPSC® SYSTEM COMMANDS

CHAPTER 3
USING iPSC® SYSTEM CALLS

CHAPTER 4
DESIGNING A CONCURRENT APPLICATION

CHAPTER 5
PROGRAMMING TECHNIQUES

CHAPTER 6
THE CONCURRENT FILE SYSTEM

CHAPTER 7
TCP/IP ON THE NODES

CHAPTER 8
THE X WINDOW SYSTEM ON THE iPSC®/860

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

APPENDIX B
iPSC® SYSTEM FEATURES

APPENDIX C
iPSC® SYSTEM SPECIFICATIONS

APPENDIX D
TCP/IP SYSTEM CALLS

LIST OF ILLUSTRATIONS

LIST OF TABLES

LIST OF TABLES

INTRODUCTION 1

NOTE

In this manual, the terms "iPSC system" and "iPSC systems" refer to the iPSC®/2, iPSC®/2S, iPSC®/860, iPSC®/860S, and iPSC®/860Plus products.

The iPSC systems offer a solution for large-scale applications such as computational mechanics, petroleum exploration, electronic design, molecular modeling, and tactical and strategic systems. In an iPSC system, a large number of processors or nodes work concurrently on the parts of a single problem.

This chapter introduces the following iPSC system components:

- System hardware (host, nodes, and peripheral units)
- Host and node system software
- Development environment (languages, libraries, debuggers and other utilities)
- Runtime environment (system commands and system calls)

iPSC® SYSTEM HARDWARE

Figure 1-1 shows the two major hardware components of an iPSC system:

- A front-end processor called the "host" (local system resource manager (SRM) or remote workstation)
- One or more cabinets (standard or compact) containing nodes (compute, I/O, and integrated) and storage devices (tape and disk drives)

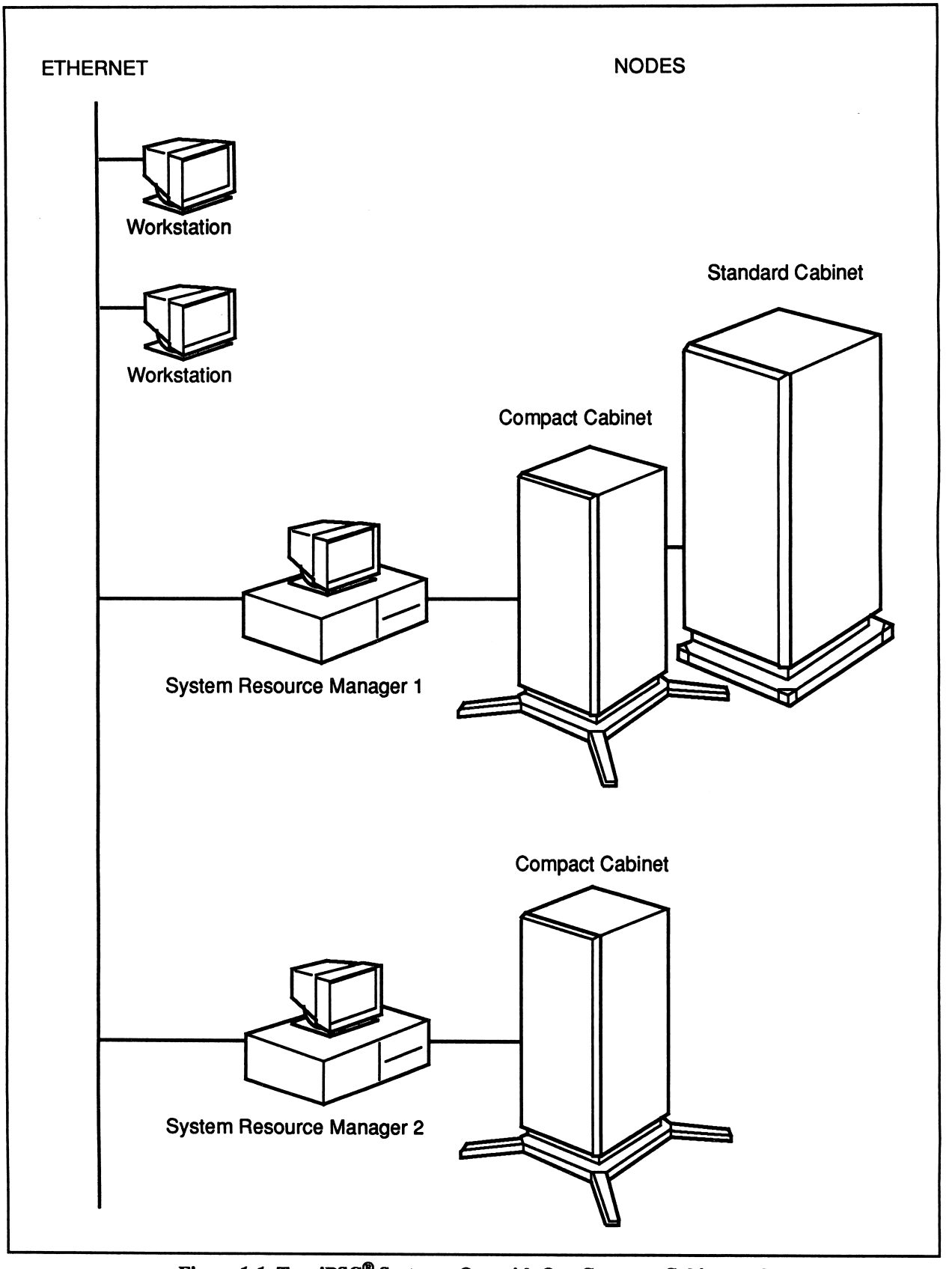


Figure 1-1. Two iPSC® Systems: One with One Compact Cabinet and One Standard Cabinet, and the Other with One Compact Cabinet

The Host—Provides Access to the Nodes

You communicate with the nodes in the cabinet through a front-end processor called the host. There are two kinds of hosts:

- An iPSC system comes with a local host (the system resource manager, or SRM) that is connected directly to the iPSC nodes.
- You can also configure a workstation (by installing remote host software) to serve as a remote host. A remote host is connected to the SRM over an Ethernet link.

The host (local or remote) runs the UNIX operating system and provides the interface between you and the nodes. In the typical user model, you compile and link your host and node programs on the host, and then load and run your host programs on the host and your node programs on the nodes.

The Cabinets—Contain Nodes and Storage Devices

The system cabinets contain cardcage modules (containing node boards) and peripheral modules (containing storage devices). There are two kinds of cabinets:

Standard — Each standard cabinet can contain cardcage modules and peripheral modules configured in the following ways:

- One cardcage module and four peripheral modules
- Two cardcage modules and two peripheral modules
- Four cardcage modules and no peripheral modules

The 17-slot cardcage holds the Unit Services Module (USM) and up to 16 node boards, VME Bus Interface Adapter (BIA) boards, or vector boards.

The peripheral module can hold up to four 760M-byte disk drives or 2G-byte tape drives.

Compact — Each compact cabinet can contain only one 34-slot cardcage, which holds one USM and up to 32 compute nodes (one slot is reserved for future use).

NODES

A node is a processor/memory pair. Each node's memory is distinct from the host and from other nodes. Each node runs the NX/2 operating system, communicates with the other nodes (by passing messages), and can access both the host file system and the iPSC Concurrent File System™ (CFS).

The collection of nodes belonging to an iPSC system is called a cube. The number of nodes in a cube is defined by its dimension (expressed as d*n*). For example, a d7 cube has 2^7 or 128 nodes.

Each node is fully connected to every other node through its Direct Connect Module™ (DCM). Each DCM has eight channels (numbered 0 through 7):

- Channels 0 through 6 are connected to the node's "nearest neighbors." For example, in a d7 cube (the largest iPSC system), each node has seven nearest neighbors and uses all seven of the available DCM channels. Because of the DCM, there is no "store and forward" of messages. Nodes that are nearest neighbors have nearly the same message latency as those that are not.

- Channel 7 has a special use. On all systems, channel 7 on node 0 connects to the SRM (giving the SRM the same kind of access to the DCM network that the nodes have). In addition, on a system that includes a Concurrent I/O (CIO) System, channel 7 on each I/O node is connected to a CIO Ethernet adapter.

There are four kinds of nodes:

- Compute nodes
- I/O nodes
- Integrated nodes
- Service nodes

Compute nodes and service nodes can reside in standard cabinets or in compact cabinets. I/O nodes and integrated nodes can reside only in standard cabinets.

Compute Nodes

Compute nodes are designed for performing computational tasks. There are five kinds of compute nodes:

CX nodes	These nodes are based on the Intel386™ processor and have an 80387 numeric coprocessor
SX nodes	These nodes are CX nodes in which the 80387 numeric coprocessor is replaced with an SX processor (for doing high-speed, floating-point, scalar arithmetic)
VX nodes	These nodes are CX nodes with a companion VX processor (for doing high-speed, vector arithmetic)
SXVX nodes	These nodes are SX nodes with a companion VX processor
RX nodes	These nodes are based on the i860™ microprocessor

NOTE

CX, SX, VX, and SXVX compute nodes are often collectively referred to as CX nodes because they are all based on the Intel386 microprocessor.

An iPSC/2 system contains no RX nodes; it consists entirely of CX, SX, VX, or SXVX compute nodes.

An iPSC/860 system either consists entirely of RX nodes or of a combination of all nodes (of which at least one is an RX node).

I/O Nodes

I/O nodes are based on CX compute nodes. The DCM of each I/O node connects to channel 7 of an associated compute or integrated node. There are two kinds of I/O nodes:

- Standard I/O nodes are connected to disk and tape drives in the Concurrent I/O (CIO) system. These nodes have a SCSI (Small Computer System Interface) controller and a modified DCM (i.e., a DCM that has only channel 7).
- Special I/O nodes are connected to the CIO Ethernet and VME Bus Interface Adapter (BIA) options. These nodes are the same as a standard I/O node except that the SCSI controller is replaced with a PBX bus.

Integrated Nodes

Integrated nodes are Intel386-based nodes that combine compute and I/O node functions.

Service Nodes

Service nodes are (essentially) I/O nodes to which no disk is attached. The primary purpose of a service node is to run a node shell (**nsh**) without using any compute nodes. By default, when you invoke the **nsh** command, the node shell runs on a service node. The **-s** option to **nsh** makes the shell run on the current cube.

STORAGE DEVICES

An iPSC system can contain two kinds of storage devices:

- Disk drives for the Concurrent File System (CFS)
- Tape drives for system backups and installation of software

The Concurrent I/O (CIO) option provides each node with access to the Concurrent File System (CFS), whose array of disks appears as a single virtual disk that several node processes can access simultaneously. Available file storage is in excess of 40G bytes.

The BIA board is an interface between an I/O (or integrated) node board with a PBX interface and a VME controller board. (The VME bus is a general purpose bus used to connect computer devices.) The BIA resides in a slot adjacent to the node board, connected to the node board's PBX interface, and the VME controller plugs into the BIA.

iPSC® SYSTEM SOFTWARE

The iPSC system software consists of:

- Host operating system software
- Node operating system software
- Software development environment
- Runtime environment

Host Operating System Software

The host (local or remote) runs the UNIX operating system with iPSC extensions and TCP/IP networking software.

- The iPSC extensions include commands, libraries, and background processes that support communication with the nodes.
- The TCP/IP networking software links the SRM with remote workstations.

iPSC® EXTENSIONS

The iPSC extensions consist of iPSC system commands, system calls, application libraries, and background system processes. Together, these extensions provide the interface to the cube.

To provide access to the cube and support communication between the host and the cube, the host runs the following iPSC system processes (in background):

adminproc.srm For non-CFS systems, the administration process runs on an SRM and keeps track of node processes.

commser The communication server process routes host-to-node and host-to-host messages. It is started by **bootcube** and runs on the SRM and on each remote host.

fserver One or more file server processes are started automatically by **getcube**. A file server process allows the node to use standard I/O functions on the host. File server processes run on both the SRM and remote hosts.

lifeline The lifeline process monitors the existence of host user processes. When a host user process terminates, *lifeline* notifies *commser*, which then cleans up any appropriate data structures. *lifeline* runs on the SRM.

loader	For CFS systems, the loader process runs on an I/O node and loads programs onto the cube.
loader.srm	For non-CFS systems, the loader process runs on the SRM and loads programs onto the cube.
rcam	The remote cube allocation manager process runs on the SRM and waits for a request from a remote host. When a remote host requests a cube, the two processes *tocube* and *tohost* are started. These processes handle message passing between the cube and the remote host. *rcam*, *tocube*, and *tohost* run on the SRM.
ripscd	The remote iPSC daemon process handles remote requests (such as a remote compile) from the remote host. *ripscd* runs on the SRM.
adminproc	For CFS systems, the administration process runs on an I/O node and keeps track of node processes.
tocube	The tocube process handles message passing between the remote host and the SRM.
tohost	The *tohost* process handles message passing between the SRM and the remote host.

TCP/IP NETWORKING SOFTWARE

The SRM and remote workstations also run the TCP/IP networking software, which lets you remotely log into other systems on the network and transfer files between systems.

Optionally, the SRM and remote workstations can run the NFS networking software. NFS allows files to be shared transparently over a local network, making it possible to access files on other workstations as though they resided on your workstation (eliminating the need to log into another system and copy the files).

The CIO Ethernet option consists of TCP/IP running on the nodes of the system, and X Window client libraries that allow you to run and develop X Window applications.

Node Operating System Software

Each node runs the NX/2 operating system, which provides message-passing capability, memory management, and process management. The NX/2 operating system also manages the numeric coprocessor and optional vector processor.

iPSC® Software Development Environment

The iPSC software development environment includes:

- Language processors
- Remote host software
- Interactive parallel debugger

LANGUAGE PROCESSORS

The iPSC system supports a number of language processors that, depending on the specific application, can be run on the SRM, a remote host, or another workstation. For example:

- The Lisp user interface runs only on the SRM
- Host programs must be compiled and linked on the host on which they will run.
- CX node programs can be compiled and linked on the SRM or on a remote host.
- RX node programs can be compiled and linked on the SRM or a workstation (but not on a remote host).

REMOTE HOST SOFTWARE

Remote host software lets you use the facilities of the SRM without actually logging onto the SRM. For example, you can create a node source file on a remote host, and then use a remote host command to copy that code to the SRM, compile and link the code on the SRM, and return the resulting executable file to the remote host.

Figure 1-2 shows the compilation and linking of a Fortran application by a user on a remote host. Note that the host code is compiled and linked on the host and the node code is compiled and linked on the SRM.

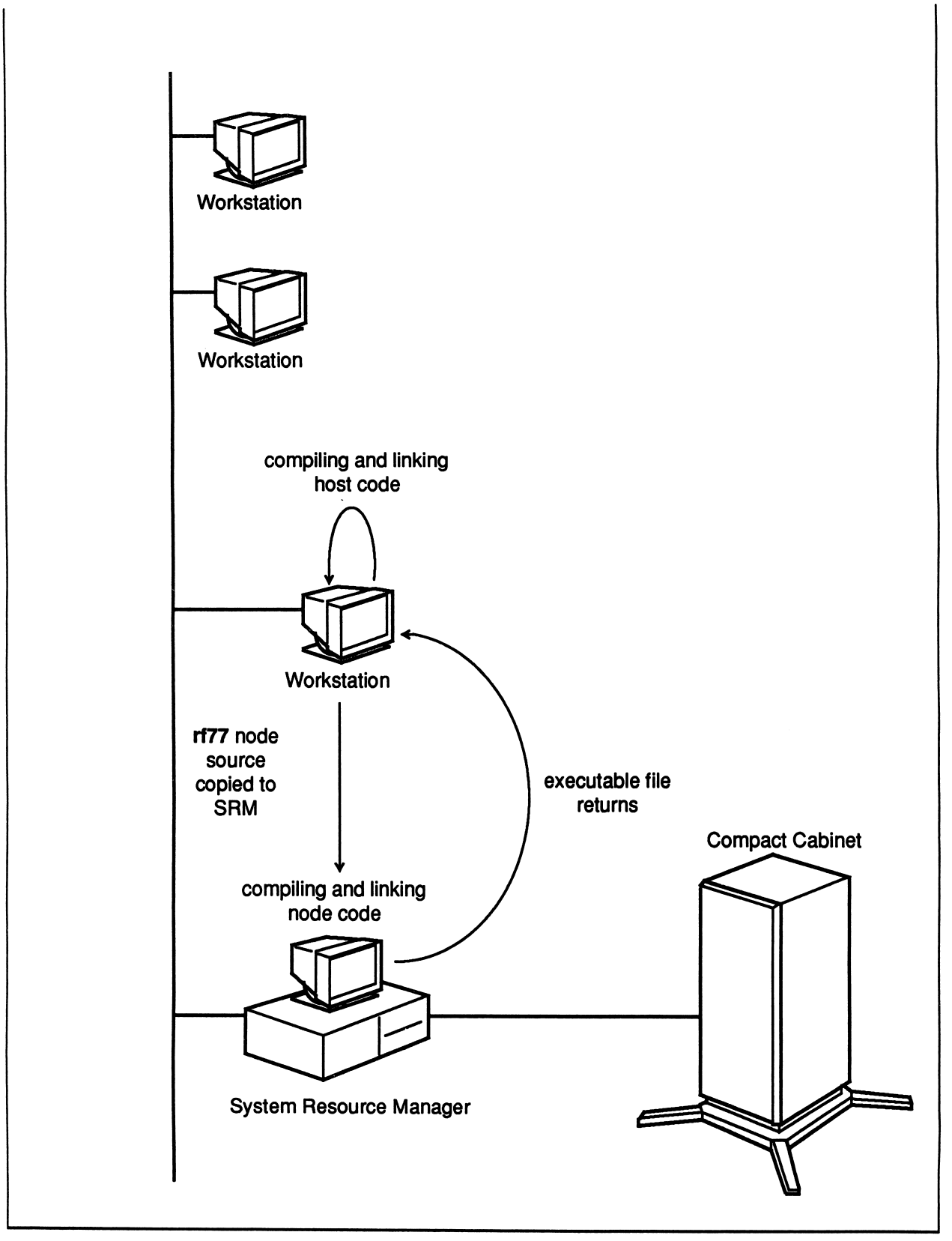


Figure 1-2. Compiling and Linking an Application for an iPSC® System

INTERACTIVE PARALLEL DEBUGGER

The iPSC Interactive Parallel Debugger (IPD) is a full-featured, source-level, symbolic debugger for debugging parallel C and Fortran programs that run on the iPSC system (on both Intel386-based and i860-based nodes).

IPD lets you load your program, control its execution, and modify the program in a variety of ways. You can also display and change program variables using their symbolic names. The IPD user interface supports the same language conventions as the program being debugged; you use Fortran language conventions when debugging Fortran programs and C language conventions when debugging C programs.

IPD combines the features of traditional serial-program debuggers with features that support the parallel programming model. For example, you can set breakpoints for individual processes, monitor message queues, and display the process status table.

IPD lets you control the debug environment, including:

- Customizing the environment
- Logging debug sessions
- Executing IPD command files

IPD lets you control program execution, including:

- Setting and removing breakpoints on code and data
- Starting from the beginning, continuing after halting, and single-stepping
- Listing source and assembly code (i860 only), including line numbers
- Displaying message queues
- Displaying and assigning values of program variables

IPD lets you control programs that have multiple processes, including:

- Setting the debug context (specifying the nodes and processes to which IPD commands apply)
- Starting and stopping individual processes
- Stopping after the program completes, after the first process completes, or after all nodes encounter breakpoints or terminate

Refer to the *iPSC®/2 and iPSC®/860 Interactive Parallel Debugger Manual* for complete information on invoking and using the IPD.

iPSC® Runtime Environment

The runtime environment consists of a set of system commands that you can use on the host, and a set of system calls that you can use in host and node programs. The system calls on the host are part of the iPSC system UNIX extensions. The system calls on the node are part of the NX operating system.

SYSTEM COMMANDS

The system commands let you perform the following kinds of activities:

- Allocate and release cubes
- Load and manage node processes
- Redirect node output to the host file system
- Compile and link C and Fortran applications
- Log host output
- Obtain information about allocated cubes
- Reboot the cube

Chapter 2 contains detailed information about using system commands.

SYSTEM CALLS

The system calls let you perform the following kinds of activities:

- Allocating, loading and releasing a cube
- Controlling processes
- Redirecting input and output
- Handling errors and exceptions
- Message passing

Chapter 3 contains detailed information about using system calls.

USING iPSC® SYSTEM COMMANDS 2

INTRODUCTION

Running a concurrent application consists of allocating a cube, loading one or more programs on the nodes, running the node processes, and then deallocating the cube. The iPSC system provides a number of commands that let you control cube allocation, manage node processes, redirect node output to the host file system, and log host output. This chapter describes those commands and gives examples of their use. You can also perform these functions from within a host program with iPSC system calls, as described in Chapter 3.

This chapter describes how to create executable host and node files in C and Fortran. To illustrate this process, the chapter shows you how to create an executable version of a Fortran program that calculates pi. The source code for this example is provided in */usr/ipsc/examples/f/pi*.

The executable version of the pi example consists of a host part that runs on either the SRM or a remote workstation and a node part that runs on the nodes. The two parts are compiled and linked separately.

If you want to see the iPSC system work right away without a lot of explanation, follow the steps in the next section.

A QUICK EXAMPLE

This section lists the steps that you must follow to run the Fortran example that calculates pi on an iPSC/2 system (all CX nodes):

1. Log onto the SRM. Create a directory in your user area. Copy all the files in */usr/ipsc/examples/f/pi* into this directory.

```
mkdir manual_ex
cd manual_ex
cp /usr/ipsc/examples/f/pi/* .
```

2. Use the **make** command, specifying CX nodes.

```
make cx
   f77 -c host.f
host.f:
   f77 -c prompt.f
prompt.f:
   f77 -o host host.o prompt.o -host
   f77 -c node.f
node.f:
   f77 -c fx.f
fx.f:
   f77 -o node node.o fx.o -node
```

3. Allocate a cube with four nodes.

```
getcube -t4
getcube successful: cube type 4m1cxn0 allocated
```

4. Run the host program. This program loads a program on each of the nodes and asks the user for input.

```
host
LOADING THE CUBE ...

***************** INTEGRATION EXAMPLE *******************

This Example uses the iPSC to calculate pi by integrating
the function:
   f(x) = 4 / (1 + x**2)
between x=0 and x=1.
The method used is the n-point rectangle quadrature rule.
```

```
How many points do you want (0 or neg. value quits)?
1000
What cube size (1 - 4) should I use ?
4
pi is approximately : 3.1415927369231266
elapsed time =  0 min. 0.014 sec.

***************** INTEGRATION EXAMPLE ******************
This Example uses the iPSC to calculate pi by integrating
the function:
   f(x) = 4 / (1 + x**2)
between x=0 and x=1.
The method used is the n-point rectangle quadrature rule.

How many points do you want (0 or neg. value quits)?
0
PLEASE WAIT WHILE I CLEAN OUT THE CUBE ...
%
```

5. Display the list of allocated cubes.

```
cubeinfo -s
CUBENAME      USER   SRM        HOST       TYPE       TTYS
iocube        root   srm_name   srm_name   0
defaultname   you    srm_name   srm_name   4m1cxn0    ttyT1
```

6. Deallocate the cube.

```
relcube
relcube released 1 cube
```

ALLOCATING AND RELEASING CUBES

iPSC system commands:	**attachcube** **cubeinfo** **getcube** **relcube**

Before you can load programs onto the nodes, you must allocate a cube. The cube may consist of all the nodes in an iPSC system or a subset of the nodes, but the number of nodes is always a power of two. That power is called the cube's dimension.

The examples in this section make a number of assumptions that may not be true about your particular iPSC system. For example, you can't try the examples that allocate vector processors unless you have nodes equipped with that option.

If you decide to type in the examples as you read along, be sure to release cubes after allocating them. The nodes making up your cubes cannot be allocated by other users until you release them. To release a cube, use the **relcube** command. With the **-a** switch, **relcube** releases all your allocated cubes. For example,

```
relcube -a
relcube released xx cubes
```

where *xx* represents the number of cubes that you have allocated. Most of the examples in this section end with a **relcube** just to ensure that, if you do type in the examples, you don't end up with unwanted cubes.

To allocate a cube, use the **getcube** command. If you use this command without arguments, you get the maximum number of available nodes. If your system contains both RX and CX nodes, **getcube** without arguments ignores the distinction and may allocate a mixed cube.

NOTE

If you allocate a cube containing a mixture of CX and RX nodes, you must load the correct type of executable programs on each type of node. That is, CX executables must be loaded on CX nodes and RX executables on RX nodes.

Redirecting Output

If you redirect the output of **getcube**, then the standard output or standard error of the node processes are also redirected. For example, the following command redirects the node's standard output to *myfile*:

```
getcube > myfile
relcube
relcube released 1 cube
```

Normally, **getcube** prints a response on the screen, indicating whether **getcube** was successful and how many nodes were allocated. In this example, however, the response is recorded in *myfile*.

Specifying the Number or Type of Nodes

Use **getcube**'s **-t** switch to specify a particular number or type of nodes. For example, to allocate a 64-node CX cube, use the following command:

```
getcube -t64cx
getcube successful: cube type 64m4cxn0 allocated
relcube
relcube released 1 cube
```

The cube type `64m4cx` means 64 CX nodes, each with at least 4M bytes of memory was allocated. The `n0` indicates that the allocated cube consists of contiguous nodes beginning with the node physically located in slot 0.

getcube always allocates a cube whose dimension is a power of two. If you give something that is not a power of two, **getcube** rounds up. For example, the following command allocates a four-node cube, even though you specified three nodes:

```
getcube -t3
getcube successful: cube type 4m4cxn0 allocated
relcube
relcube released 1 cube
```

If the requested number of nodes is unavailable, **getcube** fails.

You can also specify the number of nodes as a dimension. For example, the following command allocates 16 nodes, a four-dimensional cube:

```
getcube -td4
getcube successful: cube type d4m4cxn0 allocated
relcube
relcube released 1 cube
```

An iPSC system may have both RX and CX nodes. To choose RX nodes, include **rx** as part of the **-t** switch. For example, the following command

```
getcube -td4rx
getcube successful: cube type d4m8rxn0 allocated
relcube
relcube released 1 cube
```

If your system has only RX nodes, the **rx** appears in **getcube**'s response even though there are no CX nodes in the system.

To choose CX nodes in a system that includes both CX and RX nodes, include **cx** as part of the **-t** switch. As with **rx**, if your system has only CX nodes, **cx** appears in **getcube**'s response even though there are no RX nodes in the system.

A CX node may be equipped with a vector processor, an SX scalar processor, or have different amounts of memory.

Choosing the Node Memory Size

With **getcube**'s **-t** switch, you can choose nodes with a particular amount of memory. For example, to allocate a four-dimensional cube (16 nodes) with at least 8M-byte nodes, use the following command:

```
getcube -td4m8
getcube successful: cube type d4m8cxn0 allocated
relcube
relcube released 1 cube
```

There's no guarantee that you will get only 8M-byte nodes. The specification is that you get at least 8M-byte nodes. That is, **getcube** may satisfy **-td4cxm8** with 16M-byte nodes, even if 8M-byte nodes are available.

Choosing CX Nodes with Intel387™ Coprocessors

If you want to choose only those nodes that have the Intel387 coprocessor, include **f** as part of the **-t** switch. For example, the following command allocates 4 CX nodes that have Intel387 coprocessors installed:

```
getcube -t4f
getcube successful: cube type 4m4cxn0 allocated
relcube
relcube released 1 cube
```

Choosing CX Nodes with SX Processors

CX nodes can improve their floating point performance with the SX option. A CX node with the SX option is called an SX node. To choose SX nodes, use **getcube**'s **-t** switch. For example, the following command allocates 16 SX nodes:

```
getcube -t16sx
getcube successful: cube type 16m4sxn0 allocated
relcube
relcube released 1 cube
```

You can specify a cube with both SX and VX nodes. Here is an example:

```
getcube -t16m4sxvx
getcube successful: cube type 16m4sxvxn0 allocated
relcube
relcube released 1 cube
```

RX nodes achieve their floating point performance without the addition of an SX option.

Choosing CX Nodes with VX Processors

A CX node that has a vector processor (called a VX node) may also have an SX processor and may have up to 8M bytes of memory.

To choose VX nodes, use **getcube**'s **-t** switch. For example, the following command allocates a 16-node cube with VX nodes:

```
getcube -t16vx
getcube successful: cube type 16m4vxn0 allocated
relcube
relcube released 1 cube
```

Because you didn't specify a memory size, you got whatever memory size is available. The **16m4vx** in the **getcube** return message signifies that the minimum memory size of a node in your cube is 4M bytes, but you may have some nodes with more than 4M bytes.

Also note that if you didn't specify VX nodes, but VX nodes were all your system had, then **getcube** would give them to you. However if your system had a mixture of node types and you didn't specify VX nodes, your cube might end up with some VX nodes and some non-VX nodes — a type of hybrid cube.

RX nodes have their own vector capability and do not use a separate vector processor.

Choosing Hybrid Cubes

A hybrid cube is one whose nodes have different options or memory sizes. In some situations, you may want to specify a hybrid cube. Consider the following example. You want an eight-node cube, and you want at least four of those nodes to be VX nodes with 4M bytes. If you used ***getcube -t8m4vx***, **getcube** would try for eight VX nodes (more than you need) and fail if they were unavailable. Instead, use the following command:

```
getcube -t4m4vx/4m4
getcube successful: cube type 4m4vxn0/4m4 allocated
relcube
relcube released 1 cube
```

The / on the **getcube** command line separates the cube types. What if, when you used that last **getcube**, your iPSC system only had 8M-byte nodes? 8M bytes is certainly at least 4M bytes; so you'd still get a cube, but **getcube**'s return message would be:

```
getcube successful: cube type 4m8vxn0/4m8 allocated
```

Now, let's say that you want to allocate an 8-node cube that contains 4 RX nodes and 4 CX nodes with at least 4M bytes of memory. Use the following command:

```
getcube -t4rx/4cxm4
getcube successful: cube type 4m8rx/4m4cx allocated
relcube
relcube released 1 cube
```

Choosing Specific (Contiguous) Nodes

If you want to allocate a particular set of contiguous nodes, use **getcube**'s **n** specifier. For example, if you want nodes 24 through 31 to make up an eight-node cube, use the following command:

```
getcube -t8n24
getcube successful: cube type 8m8cxn24
relcube
relcube released 1 cube
```

If you use the **n** specifier and the specified nodes are unavailable, the cube is not allocated. For example, if you specified **8m8n24** and any of nodes 24 through 31 were already allocated, the **getcube** would fail, even if eight other nodes were available.

If you allocate a cube and don't use **n**, you are not guaranteed to have contiguous nodes. This configuration should not present a problem when you use the cube because from the program's point of view, the nodes are numbered 0, 1, 2, etc.

When a contiguous cube is allocated, **getcube**'s response contains n*number* where *number* is the starting node number for the cube.

Allocating Multiple Cubes

If you use multiple **getcubes** without intervening **relcubes**, you allocate multiple cubes. If you have multiple cubes, you must give them different names. To name a cube, use **getcube**'s **-c** switch. If you don't use the **-c** switch, the cube is given the name *defaultname*. For example, to allocate a cube and call it *mycube*, use the command:

```
getcube -c mycube -td4m8
getcube successful: cube type d4m8cxn0 allocated
relcube
relcube released 1 cube
```

getcube won't let you allocate two cubes with the same name. For example, if you try to allocate two cubes without using **-c**, what you are really trying to do is to get two cubes with the name *defaultname*. That's not allowed.

```
getcube -t2
getcube successful: cube type 2m8cxn0 allocated
getcube -t2
(host) getcube: Cubename already exists
relcube
relcube released 1 cube
```

Specifying the Current Cube

The current cube is the last cube allocated. Without arguments, the **cubeinfo** command gives information about the current cube.

If you have allocated multiple cubes, you can change the current cube to another of your cubes with **attachcube**. Use the **-c** switch with **attachcube** to specify the cube you want to be the current cube. **attachcube** without any arguments selects the cube called *defaultname*.

The **relcube** command also has a **-c** switch, allowing you to release a particular cube. If you have several allocated cubes and want to release all of them, use **relcube**'s **-a** switch.

Here is an example that allocates two cubes and changes the current cube from the second to the first. The first cube takes the default name; the second is called *mycube*.

```
getcube -t2
getcube successful: cube type 2m8rxn0 allocated
getcube -c mycube -t2
getcube successful: cube type 2m8rxn2 allocated
cubeinfo
CUBENAME      USER   SRM        HOST       TYPE     TTYS
mycube        you    srm_name   srm_name   2m8rxn2  ttyT1
attachcube
```

```
cubeinfo
CUBENAME      USER   SRM        HOST       TYPE      TTYS
defaultname   you    srm_name   srm_name   2m8rxn0   ttyT1
relcube -a
relcube released 2 cubes
```

Here is another example that allocates three cubes and then uses **relcube** to release the current cube. Without arguments, **relcube** releases only the current cube. The first allocated cube takes the default name. The other two are called *one* and *two*.

```
getcube -t2
getcube successful: cube type 2m8rxn0 allocated
getcube -cone -t2                                  You don't need a space
getcube successful: cube type 2m8rxn2 allocated        after the -c,
getcube -c two -t2                                 but you can put one in
getcube successful: cube type 2m8rxn4 allocated        if you like.

cubeinfo
CUBENAME      USER   SRM        HOST       TYPE      TTYS
two           you    srm_name   srm_name   2m8rxn4   ttyT1
relcube
relcube released 1 cube
cubeinfo
(host) cubeinfo: There is no attached cube
```

You still have two allocated cubes, but because neither of those cubes is attached, **cubeinfo** doesn't return any information about them.

Use **cubeinfo**'s **-a** switch to get information about *all* of *your* cubes that have been allocated from the workstation on which you use the **cubeinfo**. The **-a** switch does not pick up cubes belonging to other users.

The TTYS field shows the name of the device special file to which the cube's owner is attached. This is **console** for a user on the SRM, **REMOTE** for a user on a remote host, and a tty designation for a user who is remotely logged in to the SRM. Note that unattached cubes have a blank TTYS field.

```
cubeinfo -a
CUBENAME      USER   SRM        HOST       TYPE      TTYS
defaultname   you    srm_name   srm_name   2m8rxn0
one           you    srm_name   srm_name   2m8rxn2
```

cubeinfo's **-s** switch gives you even more information. How it operates depends on whether you are issuing it on a remote workstation or on an SRM.

- **On a Remote Workstation.** If you use ***cubeinfo -s*** on a remote workstation, you get information about all the cubes for all users that have been allocated from that remote workstation. If you have several iPSC systems, you may see cubes belonging to several SRMs.

- **On an SRM.** If you use ***cubeinfo -s*** on an SRM, you get information about all the cubes allocated on the iPSC system belonging to that SRM. These cubes may have been allocated from the SRM or from a remote workstation.

cubeinfo also has **-n** and **-h** switches. These switches have meaning only when you use **cubeinfo** on a remote workstation. The **-n** switch returns information about all the cubes in a network of iPSC systems. Actually, **-n** lists all the cubes allocated on the iPSC systems belonging to the SRMs listed in the *srms* file on the remote host, which is in */usr/ipsc/lib* by default. It's as if you used a ***cubeinfo -s*** on each SRM in the network. Use the **-h** switch to specify a particular SRM.

For example, assume that your workstation is an SRM, and you execute **cubeinfo -s**:

```
cubeinfo -s
CUBENAME        USER     SRM         HOST          TYPE       TTYS
iocube          root     srm_name    srm_name      0
defaultname     other    srm_name    rhost_name    2m8rxn0    REMOTE
one             you      srm_name    srm_name      2m8rxn2
relcube -a
relcube released 2 cubes
```

Notice the cube named *iocube*. It is owned by root and is used by the Concurrent File System. It appears only on the SRM.

MANAGING PROCESSES

iPSC system commands:	**killcube** **load** **startcube** **waitcube**

After you've allocated a cube, your next action is usually to load one or more processes onto the cube and run them. If you have RX nodes, you are restricted to one process per node, and its NX pid must be 0.

To load a node process on the cube, use the **load** command. With no switches, **load** puts the specified file onto every node of the current cube and assigns an NX pid of 0 to the node process. Each node process starts running as soon as it's loaded. For example, if you have an executable called *node*, you can allocate a cube and load *node* on each node of that cube as follows:

```
getcube -t4
getcube successful: cube type 4m4cxn0 allocated
load node
relcube
relcube released 1 cube
```

Consider the following node program written in C. It exists in */usr/ipsc/examples/c/hello/nodes* and is called *node.c*. When this program runs on a node, the node writes its node number and process id to the host's screen.

```
#include <stdio.h>
main()
{
   printf("Hello world from node %d process id %d!\n",
      mynode(), mypid());
}
```

Copy it to your own directory and compile it. Specify the executable as *node* and be sure to include the **-node** switch so that the appropriate libraries are linked in. A makefile is also available.

```
cc -o node node.c -node
```

If you have RX nodes, you must use the **icc** compiler:

```
icc -o node node.c -node
```

Then, allocate a two-dimensional cube and load the node program on each node. Note that, unlike many of the examples in this manual, this example does not require a host program. You can use iPSC commands instead of a host program to control and manage the cube. There's no guarantee that the "Hello world." messages come out in the order shown below.

```
getcube -t4
getcube successful: cube type 4m4cxn0 allocated
load node
Hello World from node 0 process id 0!
Hello World from node 1 process id 0!
Hello World from node 2 process id 0!
Hello World from node 3 process id 0!
```

If you want to load a program on only some nodes in your cube, list those node numbers on the **load** command line. For example, to load *node* on nodes 1 and 3, use the following command:

```
load 1 3 node
Hello World from node 1 process id 0!
Hello World from node 3 process id 0!
relcube -a
relcube released 1 cube
```

The **load** command has a **-c** switch that allows you to load a program onto a specified cube. Note that you can load programs on an unattached cube. For example:

```
getcube -c one -t4
getcube successful: cube type 4m4cxn0 allocated
getcube -c two -t4
getcube successful: cube type 4m4cxn4 allocated  Cube two is attached.
load -c one node                                 Load the file node
Hello World from node 0 process id 0!                on cube one.
Hello World from node 1 process id 0!
Hello World from node 2 process id 0!
Hello World from node 3 process id 0!
relcube -a
relcube released 2 cubes
```

If you have CX nodes and you want to load multiple programs on a node, use multiple **load** commands. Of course, programs on the same node must have distinct pids. By default, **load** assigns an NX pid of 0 to the node program that it loads. You can specify a pid with **load**'s **-p** switch.

```
getcube -c one -t4
getcube successful: cube type 4m4cxn0 allocated
load -c one -p 1 node
Hello World from node 0 process id 1!
Hello World from node 1 process id 1!
Hello World from node 2 process id 1!
Hello World from node 3 process id 1!
relcube -a
relcube released 1 cube
```

If the file you're loading requires arguments, just include them on the **load** command line after the filename. For example, if *node* required the argument **debugmode**, you can load it as follows:

```
load node debugmode
```

Normally, a program loaded with **load** starts executing immediately after being loaded. That may not be what you want. You may want to load the program and then start it running at a later time. **load** provides the **-H** switch for this purpose.

To start a program that was loaded with **-H**, use the **startcube** command. This command also has a **-c** switch to select a particular cube and a **-p** switch to select a particular pid. It also accepts node numbers.

For example, assume that you have CX nodes and that you load *node* on a cube called *one* twice, once with pid 1 and once with pid 0 (the default). You can selectively start up each process.

```
getcube -c one -t2
getcube successful: cube type 2m4cxn0 allocated
getcube -c two -t2
getcube successful: cube type 2m4cxn2 allocated
load -c one -p 1 -H node                    One is not the current cube.
load -c one -H node
startcube -c one -p 1 1
Hello World from node 1 process id 1!
startcube -c one -p 1 0
Hello World from node 0 process id 1!
attachcube -c one
startcube
Hello World from node 0 process id 0!
Hello World from node 1 process id 0!
relcube -a
relcube released 2 cubes
```

The **killcube** command kills one or more processes on a cube. It also cleans up after a process (for example, getting rid of pending messages). For that reason, issuing a **killcube** before each **relcube** is good practice. **killcube** takes the same options as **startcube**. You can specify a particular cube with **-c** and a particular process with **-p**. You can also select individual nodes by listing their node numbers on **killcube**'s command line.

The **waitcube** command lets you wait until one or more processes on a cube have completed. It is particularly useful in shell scripts that assign sequential work for a cube. **waitcube** takes the same options as **killcube** and **startcube** and has three additional switches: **-f**, **-i**, and **-s**.

The **-f** switch, which doesn't take an argument, lets you wait for only the first process that meets the other requirements. The **-i** and **-s** switches let you use the interrupt signal (usually the **<DEL>** key) to kill node programs. Without either of these switches, the interrupt signal kills the **waitcube** command without affecting node processes.

With the **-i** switch, an interrupt signal kills the node process you were waiting for, then causes **waitcube** to exit. The **-s** switch is designed for use when you load a shell on a node. If you load the node shell with the **nsh** command, you may never need to use **waitcube** with the **-s** switch. But if you write your own shell, use a ***waitcube -s*** after loading it. This command lets interrupt signals pass through the shell to processes on your cube.

In addition, standard input to node programs may come from the standard input of **waitcube** (that is, ***waitcube < my_prog_data***).

INPUT/OUTPUT WITH THE HOST

iPSC system commands: **newserver**
syslog

A node program inherits its standard input, output, and error from **getcube**. Earlier, you saw that you could redirect the standard output and standard error of a cube when you used the **getcube** command. Note, however, that I/O performed by a host program is not redirected. Use the **syslog** command to also redirect the standard output and standard error of the host program.

For example, assume that you have a host program called *host* and a node program called *node*. Also assume that you redirected the cube's standard output to *myfile* when you allocated the cube. To redirect the host program's standard output to *myfile*, in addition to the standard outputs from the node programs, use the following commands:

```
getcube -t2 > myfile
host | syslog
relcube -a
relcube released 1 cube
```

If you work in the C shell and want both standard output and standard error from the node and standard output from the host to go to *myfile*, use the following commands:

```
getcube -t2 >& myfile
host | syslog
relcube -a
relcube released 1 cube
```

You can also redirect the node's standard output and error after the cube is allocated and the node processes are running. Use the **newserver** command to do this. **newserver** kills the old file server and starts a new one. For example:

```
getcube -t2
getcube successful: cube type 2m4n0 allocated
load -H node
newserver > myfile
startcube
waitcube                                  Must wait for the nodes to finish before
cat myfile                                output is guaranteed to be in the logfile.
Hello World from node 0 process id 0!
Hello World from node 1 process id 0!
relcube
relcube released 1 cube
```

COMPILING AND LINKING C AND FORTRAN APPLICATIONS

iPSC system commands:	cc	f77
	rcc	rf77
	icc	if77

An iPSC application can contain two kinds of programs:

- Host programs (run on local or remote hosts only)
- Node programs (run on CX or RX nodes)

You can compile and link these programs on three kinds of platforms:

- A local host (the SRM)
- A remote host (a workstation that is attached to the same network as the SRM and communicates with the SRM via the Remote Host Software)
- A workstation that is attached to the same network as the SRM and communicates with the SRM via TCP/IP

To compile and link host programs, you use the appropriate host command (making sure to link in the iPSC host libraries). For example:

cc -host ...	Host C programs on the SRM
cc -lhost ...	Host C programs on a remote host
f77 -host ...	Host Fortran programs on the SRM
f77 -lhost ...	Host Fortran programs on a remote host

To compile and link CX and RX node programs, you use the appropriate iPSC command (making sure to link in the iPSC node libraries). For example:

cc -node ...	CX node C programs on the SRM
rcc -node ...	CX node C programs on a remote host
icc -node ...	RX node C programs on an SRM or workstation
f77 -node ...	CX node Fortran programs on the SRM
rf77 -node ...	CX node Fortran programs on a remote host
if77 -node ...	RX node Fortran programs on an SRM or workstation

Note that you cannot compile and link host programs or CX node programs on a workstation unless that workstation is running the Remote Host Software. In addition, you cannot compile and link RX node programs on a remote host. Table 2-1 shows which compiler driver to use based on the kind of program and platform. In this table, "—" means that you cannot compile and link the indicated program on the indicated platform.

Table 2-1. Which Compiler Driver to Use

Kind ofProgram	Platform		
	Local Host (SRM)	Remote Host	Workstation
C Host	**cc -host ...**	**cc -lhost ...**	—
C CX Node	**cc -node ...**	**rcc -node ...**	—
C RX Node	**icc -node ...**	—	**icc -node ...**
Fortran Host	**f77 -host ...**	**f77 -lhost ...**	—
Fortran CX Node	**f77 -node ...**	**rf77 -node ...**	—
Fortran RX Node	**if77 -node ...**	—	**if77 -node ...**

General Comments

The following sections tell how to create host-executable and node-executable C and Fortran programs. But first, some general comments:

- The **cc**, **rcc**, **icc**, **f77**, **rf77**, and **if77** commands invoke drivers that preprocess, compile, and link C and Fortran programs, creating executable files (as determined by optional switches).
  - For the sake of brevity, these discussions also refer to these drivers as commands, compilers, or some other variation. The preprocess and link steps are implied even when not explicitly mentioned; they are usually called out only when important to a specific discussion.
  - Unless stated otherwise, these discussions apply to all versions of all drivers. When a discussion specifies C or Fortran, then the discussion applies only to the drivers for that language. When a discussion specifies a specific driver, then the discussion applies only to the specified driver.
  - In most cases, the examples in these discussions assume a C source program. However, the discussions usually apply to Fortran source programs as well. When a corresponding Fortran example is significantly different from its C counterpart, a Fortran example is also given.

- For information on the **cc** and **f77** commands, refer to the UNIX documentation. Tables 2-2, 2-3, and 2-4 summarize the switches for compiling host programs, CX node programs, and RX node programs, respectively.

- For information on the **rcc** and **rf77** commands, refer to the *iPSC®/2 and iPSC®/860 Programmer's Reference Manual.*

- For information on the **icc** command, refer to the *iPSC®/860 C Compiler User's Guide.*

- For information on the **if77** command, refer to the *iPSC®/860 Fortran Compiler User's Guide.*

Table 2-2. Switches for Host Programs (cc, f77)

Switches	Description
-host	(SRM only: **cc** or **f77**) Uses the SRM's native compiler and links in the SRM host libraries.
-lhost	(Remote host only: **cc** or **f77**) Uses the remote host's native compiler and links in the iPSC host libraries located in *libhost.a* in the standard libraries directory.

Table 2-3. Switches for CX Node Programs (cc, rcc, f77, rf77)

Switches	Description
-node	Compiles and links for a CX node (a Intel386-based node that has an 80387 numeric coprocessor).
-node -vec	Compiles and links for a CX node and links in the 80387 version of the vector routines.
-node -sx	Compiles and links for an SX node (a CX node that has an SX processor in place of the 80387 numeric coprocessor).
-node -sx -vec	Compiles for an SX node and links in the SX version of the vector routines.
-node -vx -vec	Compiles for a VX node (a CX node that has a companion VX processor) and links in the VX version of the vector routines.
-node -vx -vecdb	Compiles and links for a VX node and links in the VX debug version of the vector routines.
-node -sx -vx -vec	Compiles for an SXVX node (SX node that has a companion VX processor) and links in the SXVX version of the vector routines.
-node -sx -vx -vecdb	Compiles for an SXVX node and links in the SXVX debug version of the vector routines.

Table 2-4. Switches for RX Node Programs (icc, if77)

Switches	Description
-node	Compiles and links for an RX node (an i860-based node).
-node -lvec	Compiles and links for an RX node that uses the i860 version of the vector routines (node). There are no name clashes between the i860 and the Intel386 vector libraries (VecLib).
-node -lsocknode	Compiles and links for an RX node and links in the socket libraries used by node TCP/IP.

- As a general rule, always include the file *cube.h* in all host and node C programs. This file contains definitions and declarations needed by the iPSC C system calls. Although a specific application may not need the definitions and declarations contained in *cube.h*, the overhead involved in including it in all programs is minor. Include it in your C programs as follows:

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

  For Fortran programs, the corresponding file is *fcube.h*. Include it in your Fortran programs as follows:

```
include 'fcube.h'
```

  If your include files are in a nonstandard directory, you must use the **-I** switch on the driver command line to identify the nonstandard directory.

- The example command lines in the following sections assume that the host and CX node libraries are located in the directory */usr/lib* and the RX node libraries are located in */usr/i860lib*. If this is not the case at your site, then you must modify the example command lines in one of the following ways:

  - Use the **-L** switch to provide the pathname of the directory in which these libraries are located. For example, the following command line compiles and links an RX node program at a site where the node libraries are located in the directory */usr/local/lib/ipsc*:

```
icc -node -L/usr/local/lib/ipsc myprog.c
```

  - Omit the **-lhost** switch and specify the complete pathname of the appropriate library. For example, the following command line produces the same result as the previous example:

```
icc -node myprog.c /usr/local/lib/ipsc/libnode.a
```

- If your Fortran program is in a file that ends with an uppercase ".F" extension (rather than the standard lowercase ".f"), then the drivers run the C preprocessor on the file. This enables you to use lines like the following in a Fortran program:

```
#include <fcube.h>

#define MAX 87
```

  Refer to the section "Resolving Include Files" (on page 2-25) for more information on using the C preprocessor with Fortran source files.

- Most driver switches are not order sensitive. However, order is important for the **-L**, **-l**, **-O**, and **-g** switches, and for listing libraries when linking. When constructing driver command lines, keep the following guidelines in mind:

  - List libraries in the order in which they should be searched. The iPSC linker is a single pass linker; it cannot resolve a backward library reference (i.e., a reference to a library object that was defined in a library that has already been searched). Backward references between objects, however, is not a problem, as all listed objects are linked unconditionally.

  - The **-L** switch affects only the search path of libraries that are listed (using the **-l** switch) after the **-L** switch. You may specify the **-L** switch repeatedly, in which case the first path searched is the last path specified. Note that system libraries are also affected by the **-L** switch.

  - A **-g** switch nullifies any **-O** switch that appears before it, setting the optimization to **-O0**. Any **-O** switch that appears after a **-g** switch is taken, however.

Compiling and Linking a Host Program

A host program runs on the host computer and communicates with the iPSC system. Therefore, the program must be compiled by the host's native compiler and linked with the iPSC host libraries.

- On the SRM, use the **cc** or **f77** command with the **-host** switch to specify the native compiler and the host libraries located in */usr/lib/libhost.a*. For example, consider the pi example, which consists of three source files (*host.c*, *prompt.c*, and *node.c*), two of which (*host.c* and *prompt.c*) are host programs. The following command line compiles and links the files *host.c* and *prompt.c* to create an executable host program called *host*:

```
cc -host -o host host.c prompt.c
```

- On a remote host, use the **cc** or **f77** command with the **-lhost** switch, not the **-host** switch. For example, compiling and linking the above programs on a remote host requires the following command line:

```
cc -lhost -o host host.c prompt.c
```

- You cannot compile and link a host program on a workstation.

Compiling and Linking a CX Node Program

As implied by the switches described in Table 2-3 (on page 2-20), CX nodes come in several flavors:

- The "vanilla" CX node has a Intel386 processor and an 80387 numeric coprocessor.
- An SX node is a CX node in which the 80387 numeric coprocessor is replaced with an SX processor (for doing high-speed, floating-point, scalar arithmetic).
- A VX node is a CX node with a companion VX processor (for doing high-speed, vector arithmetic).
- An SXVX node is an SX node with a companion VX processor.

Regardless of the flavor, all CX node programs run on a Intel386-based node. Therefore, the programs must be compiled by the host's Intel386 cross-compiler and linked with the CX node libraries. Note that you can compile CX node programs only on the SRM or a remote host, not on a workstation.

COMPILING AND LINKING ON THE SRM

On the SRM, you compile a CX node program using the **cc** or **f77** command with the **-node** switch. This combination specifies the Intel386 compiler and the CX node libraries located in */usr/lib/libnode.a*. Again, consider the pi example. The following command line creates a program called *node* for execution on a vanilla CX node:

```
cc -node -o node node.c
```

SX Node Programs

If the node program is intended for an SX node, use the **-sx** switch. (The 80387 numeric coprocessor and the SX processor return results in different registers. The **-sx** switch makes sure that **cc** or **f77** produces code that is correct for the SX processor.) For example:

```
cc -node -sx -o node node.c
```

VX Node Programs

If the node program is intended for a VX node, use the **-vx** switch (which brings in the interface library for the vector board) and either the **-vec** or **-vecdb** switch:

- The **-vec** switch brings in the appropriate library of vector routines (depending on whether you specified processing for a CX, SX, VX, or SXVX node).
- The **-vecdb** switch is like **-vec**, except that it brings in a debug version of the vector library routines. The debug version performs additional testing such as checking for proper data types and alignment.

For example, the following command line creates an executable for a VX node using the standard vector library routines:

```
cc -node -vx -vec -o myprog myprog.c
```

Note that if you specify the **-vx** switch, you must also specify either **-vec** or **-vecdb**.

SXVX Node Programs

If the node program is intended for an SXVX node, use both the **-sx** switch and the **-vx** switch (plus either the -vec or **-vecdb** switch). For example:

```
cc -node -sx -vx -vecdb -o myprog myprog.c
```

COMPILING AND LINKING ON A REMOTE HOST

On a remote host, you compile a CX node program using the **rcc** or **rf77** command, which does the following:

1. Copies the node source files from the remote host to the SRM
2. Compiles and links the files on the SRM (using the SRM's Intel386 compiler and CX node libraries)
3. Copies the resulting executable file from the SRM to the remote host

For example, to compile the pi example's node program from a remote host, use the following command line:

```
rcc -node -o node node.c
```

Note that because the **rcc** or **rf77** command uses the SRM's **cc** or **f77** command to do the actual preprocessing, compiling, and linking, **rcc** or **rf77** supports the same switches as the **cc** or **f77** command for processing programs for CX, SX, VX, and SXVX nodes.

Selecting a Particular SRM

If you have several iPSC systems on a network, **rcc** or **rf77** sends your source files to the first available SRM listed in the *srms* file unless you use the **-h** switch to select a particular SRM. For example, to compile the pi example's node program on the SRM called *jaxom*, use the following command line:

```
rcc -node -h jaxom -o node node.c
```

Resolving Include Files

Normally, include files are resolved (included) when the program is preprocessed on the SRM. However, the **rcc** or **rf77** command's **-cpp** switch causes the C preprocessor to run on the remote host (before copying the node source file to the SRM). The **-cpp** switch lets you control where include files are resolved:

- If a C include file name is surrounded with double quotation marks (e.g., `"filename"`) or if a Fortran include file name is surrounded with single quotation marks (e.g., `'filename'`), then the file is looked for on the remote host. If the file is not found on the remote host, then it is looked for on the SRM.

- If a C or Fortran include file name is surrounded with angle brackets (e.g., `<filename>`), then it is looked for on the SRM. Note, however, that this applies only to include files that are listed in the source file referenced on the driver command line (i.e., the "top-level" include file). Any include file listed in a host-resolved include file is also resolved on the host, even if the filename is surrounded by angle brackets.

For example, assume that the node source code contains the following line:

```
C source:       #include "rhost.inc"
Fortran source: include 'rhost.inc'
```

Also assume that the file *rhost.inc* contains the following line:

```
C source:       #include <suntool.h>
Fortran source: include <suntool.h>
```

Because *rhost.inc* is in quotation marks, the **-cpp** switch causes **rcc** or **rf77** to look for *rhost.inc* on the remote host. And because the file *suntool.h* is referenced in the file *rhost.inc*, **rcc** or **rf77** also looks for *suntool.h* on the remote host, despite the fact that *suntool.h* is surrounded by angle brackets. However, if the *suntool.h* file were included in the node source file, then its delimiting angle brackets would cause **rcc** or **rf77** to look for it on the SRM.

Note that if your source file does not specify a full pathname for an include file, and the include file is not a valid relative pathname (i.e., relative to the directory you were in when you used the **rcc** or **rf77** command), then you must use the **-I** switch to specify a directory to search for include files. For example, the following command line specifies */usr/ipsc/myincludes* as the include file directory:

```
rcc -node -cpp -I/usr/ipsc/myincludes -o node node.c
```

Including the **-cpp** switch with **rf77** causes the remote host to run the C preprocessor on your Fortran source file before copying the file to the SRM. Using the C preprocessor is sometimes useful for Fortran programs. Note that if the Fortran source program is in a file that ends with an uppercase ".F" extension (rather than the standard lowercase ".f"), the C preprocessor is run on both the remote host and the SRM.

Compiling and Linking an RX Node Program

All RX node programs run on a i860-based node. Therefore, the programs must be compiled by the host's i860 cross-compiler and linked with the RX node libraries. Note that you can compile RX node programs only on the SRM or a workstation, not on a remote host.

Whether on the SRM or a supported workstation, you compile an RX node program using the **icc** or **if77** command with the **-node** switch. This combination specifies the i860 cross-compiler and the RX node libraries located in */usr/i860lib/libnode.a*. For the pi example described earlier, the following command line creates a program called *node* for execution on an RX node:

```
icc -node -o node node.c
```

If you are compiling a node program for use on an RX node with TCP/IP libraries, you must also link in the socket libraries using the **-lsocknode** switch. For example:

```
icc -o node node.c -lsocknode -node
```

Refer to Chapter 8 for more information on the version of TCP/IP that runs on RX nodes.

Runtime Profiling

To incorporate runtime profiling into your node program, compile your application using the **-p** switch, and then load the program and run it as usual. At the end of execution, the profiling data are stored in the current directory in a file called *mon.out.nodenum*, where *nodenum* is the node id. Then, use the UNIX **prof** tool to analyze the data. For information about using the **prof** tool, refer to the *UNIX System V/386 Programmer's Guide* and the *UNIX System V/386 Programmer's Reference Manual*.

For example:

1. Compile using the **-p** switch:

```
cc -node -p -o prog prog.c                    C version

f77 -node -p -o prog prog.f                   Fortran version
```

2. After executing the program on node 1, invoke the **prof** tool:

```
prof -m mon.out.001 prog
```

Example Makefiles

This section describes example makefiles that let you compile the pi example described earlier in this manual for CX, SX, or RX nodes by invoking **make** with the appropriate argument.

Note that in a makefile, the **-sx** and **-vx** switches must appear in both the compile and link steps. The **-vec**, **- lvec**, **-vecdb**, **-node**, **-host**, and **-lsocknode** switches, on the other hand, are needed only on the link step. The **-sx**, **-vx**, and **-node** switches result in the definitions of preprocessor symbols _ _SX, _ _VX, and _ _NODE, respectively. These are used for conditional compilation in programs that run in multiple environments.

Here is the C version of the makefile:

```
#
# This makefile compiles and links the host.c, prompt.c,
# and node.c files for the pi numerical integration example.
#
help:
                                                        @echo
      @echo "You must specify the kind of node for which you want"
     @echo "to build a node executable; use one of the following:"
                                                        @echo
          @echo "make cx    (for 386 nodes with 387 coprocessors)"
           @echo "make sx    (for 386 nodes with SX coprocessors)"
                             @echo "make rx    (for i860 nodes)"
                                                        @echo
```

```
cx:
                                        make "COMPILER=cc" host
                                        make "COMPILER=cc" node

sx:
                                        make "COMPILER=cc" host
                          make "COMPILER=cc" "SWITCHES=-O -sx" node

rx:
                                        make "COMPILER=cc" host
                             make "COMPILER=icc" "SWITCHES=-O" node

host:                                                host.o prompt.o
                          $(COMPILER) -o host host.o prompt.o -host

node:                                                    node.o fx.o
                  $(COMPILER) $(SWITCHES) -o node node.o fx.o -node

clean:
                               rm host node host.o node.o prompt.o

.c.o:
                                     $(COMPILER) $(SWITCHES) -c $<
```

Here is the Fortran version of the makefile:

```
#
# This makefile compiles and links the host.f, prompt.f,
# and node.f files for the pi numerical integration example.
#
help:
                                                              @echo
      @echo "You must specify the kind of node for which you want"
     @echo "to build a node executable; use one of the following:"
                                                              @echo
          @echo "make cx    (for 386 nodes with 387 coprocessors)"
           @echo "make sx    (for 386 nodes with SX coprocessors)"
                             @echo "make rx    (for i860 nodes)"
                                                              @echo

cx:
                                        make "COMPILER=f77" host
                                        make "COMPILER=f77" node

sx:
                                        make "COMPILER=f77" host
                          make "COMPILER=f77" "SWITCHES=-O -sx" node
```

```
rx:
                                        make "COMPILER=f77" host
                                make "COMPILER=if77" "SWITCHES=-O" node

host:                                                   host.o prompt.o
                                $(COMPILER) -o host host.o prompt.o -host

node:                                                   node.o fx.o
                        $(COMPILER) $(SWITCHES) -o node node.o fx.o -node

clean:
                                    rm host node host.o node.o prompt.o

.f.o:
                                          $(COMPILER) $(SWITCHES) -c $<
```

REBOOTING THE CUBE

iPSC system commands: **rebootcube**

Occasionally, a problem can occur that requires rebooting of the iPSC system (for example, a problem with the host daemons or a reload and reset of the node operating system). Normally, the reboot is performed by a person with superuser access to the system.

If you don't have superuser access, you can use **rebootcube** to perform a limited system reboot. You can do this from the SRM or a remote host. If your remote host is a diskless client, **rebootcube** will not work. This is because **rebootcube** is a setuid program owned by *root* on your server. When you execute **rebootcube** on a diskless client, it runs as *root* on your client, but this is not the *root* on the server. You get the following error message:

```
(host) setuid: Not owner
```

In this situation, you must become *root* on your diskless workstation and run **bootcube.**

Here is an example of how you could reboot an iPSC/860 system with CFS:

1. Release all the cubes that you have allocated:

```
relcube -a
relcube released 1 cube
```

2. Verify that no one else has any cubes allocated:

```
cubeinfo -s
CUBENAME    USER     SRM          HOST          TYPE          TTYS
iocube      root     everglad     everglad      0
```

NOTE

All cubes must be released before using the **rebootcube** command.

3. Reboot the system:

```
rebootcube
```

As the system reboots, it displays it status. For example:

```
1.  Reset driver
2.  Scan cardcages
3.  Reset nodes
4.  Scan nodes
5.  Download 386 boot loader: /usr/ipsc/lib/bootld
6.  Download 860 boot loader: /usr/i860/ipsc/lib/bootld
7.  Start boot loader
8.  Load SRM Direct Connect Module
Load SRM DCM with /usr/ipsc/lib/hbits.g
9.  Query nodes
10.  Load 386 node Direct Connect Modules
Load 3 node DCM's with /usr/ipsc/lib/386nbits.e
11.  Load 860 node Direct Connect Modules
Load 4 node DCM's with /usr/ipsc/lib/860nbits.d
Load 2 node DCM's with /usr/ipsc/lib/860nbits.f
12.  Initialize node Direct Connect Modules
13.  Test nodes
14.  Reset Direct Connect Modules
15.  Check configuration file
16.  Execute startup run file: /usr/ipsc/lib/rc1  -Q /usr/ipsc/conf/cubeconf
17.  Send load command to nodes
18.  Get boot partition
19.  Download /usr/ipsc/lib/nx.b
20.  Download /usr/i860/ipsc/lib/nx.b
21.  Release boot partition
22.  Execute startup run file: /usr/ipsc/lib/rc2  -Q /usr/ipsc/conf/cubeconf
23.  Execute startup run file: /usr/ipsc/lib/rc3  -Q /usr/ipsc/conf/cubeconf
Found 4 volumes of file system network nuggies
Starting CFS Checkers:

Checking directory entries ...
9 Entries checked

Checking file entries ...
8 Files checked

Checking block maps ...
CFS Checkers done
CFS initialization complete
24.  Start TCP services
```

USING iPSC® SYSTEM CALLS 3

INTRODUCTION

iPSC system calls are available to host and node programs. These system calls provide such functions as allocating and releasing a cube, passing messages between processes, performing I/O, and controlling node processes. Some system calls can be issued by either host or node programs. Others are available only to host or only to node programs.

Host and node programs may also issue UNIX system calls. The SRM runs the UNIX System V operating system while a remote host often runs a variant of the Berkeley UNIX 4.3 operating system. Note, however, that a node program's system calls resemble UNIX System V calls, even if its host is a remote workstation running Berkeley UNIX 4.3.

This chapter introduces most of the iPSC system calls. Each section describes a particular functional group, and numerous examples are given in both C and Fortran. Refer to the *iPSC®/2 and iPSC®/860 System Administrator's Guide* for information about calls that require root privilege, and refer to the *iPSC®/2 and iPSC®/860 Math Libraries Reference Manual* for information about the vector calls used with the VX processor. The next chapter describes the calls used with the Concurrent File System.

Appendix A summarizes all the iPSC system calls.

CUBE CONTROL

You control the operation of the cube with calls that let you:

- Allocate, load, and release a cube
- Control processes
- Redirect input and output
- Handle errors and exceptions

Allocating, Loading, and Releasing a Cube

System Call	Environment
attachcube()	**host**
cubeinfo()	**host**
getcube()	**host**
load()	**host/node**
myhost()	**host/node**
mynode()	**host/node**
mypid()	**host/node**
nodedim()	**host/node**
numnodes()	**host/node**
relcube()	**host**
setpid()	**host**

When you allocate a cube, you can assign it a name or accept the name *defaultname*. If you allocate more than one cube at a time, you must give each cube a different name.

The last cube allocated is the current cube. When you issue a **cubeinfo()**, a **load()**, or a **relcube()** without specifying a cube name, the call assumes the current cube. If you have several cubes allocated, you can change the current cube with **attachcube()**.

Here is a Fortran code fragment for a host program that checks to see if any cubes are allocated, and if none are, then allocates a cube called *mycube* and loads each node in the cube with a node program called *node_prog*. The system call **cubeinfo()** returns an integer that is the number of allocated cubes. If no cubes are allocated, the call returns 0.

```
      include '/usr/include/fcube.h'
      integer NUMSLOTS
      parameter (NUMSLOTS = 10)
      integer CURRENTCUBE
      parameter (CURRENTCUBE = 0)
         .
         .
         .
C SLOTSIZE is defined in fcube.h as
C   PARAMETER (slotsize = 11)
C cubeinfo is declared in fcube.h as
C   INTEGER*4 cubeinfo
C
```

```
character*16 ct(SLOTSIZE, NUMSLOTS)
if(cubeinfo(ct, NUMSLOTS, CURRENTCUBE) .eq. 0) then
   call getcube('mycube', '', '', 1 '',)
   call setpid(3)
   call load('node_prog', -1, 0)
      •
      •
      •
   call relcube('')
endif
```

Notice that when you load a node process, you must specify its NX pid. The example just shown chooses process 0 on all nodes (-1). For CX nodes, you can choose any unsigned integer as a pid, but for RX nodes you *must* choose 0. Later on, your node program may want to make use of its NX pid. When executed on a node, the call **mypid()** returns a process's NX pid.

A host also has an NX pid, in addition to its UNIX pid. You choose the host's NX pid with **setpid()**. When you pass messages between the host and the nodes, you use this pid. A host program must do a **setpid()** before passing any messages (or before making most iPSC system calls).

You also may want the number of the node on which your process is executing. Use the **mynode()** call. This call returns the logical node number, not the physical node number.

The slot in which a node board resides determines its physical node number, and so every node in an iPSC system has a unique physical node number. As a programmer, you don't need to be concerned with this number. Nodes in a cube are identified by a logical node number in the range from zero to one less than the number of nodes in the cube. For example, if you allocate two 16-node cubes from the same iPSC system, both cubes have nodes with numbers 0 through 15.

Other informational calls are **myhost()**, **nodedim()**, and **numnodes()**. For message passing purposes, the host is considered to have a node number, which is always one more than the highest numbered node in the cube (or equal to the number of nodes in the cube). For example, the host's node number in a 16-node cube is 16. The call **myhost()** returns the host's node number.

numnodes() returns the number of nodes in a cube, and **nodedim()** returns the cube's *dimension*. The number of nodes in a cube is $2^{dimension}$.

Controlling Processes

System Call	Environment
flick()	**host/node**
killcube()	**host/node**
killproc()	**host/node**
waitall()	**host/node**
waitone()	**host/node**

If you have more than one process running on a CX node, they have time-sliced access to the node's CPU. If a process comes up for execution and it wants to defer to the next process in line, it can issue the **flick()** call. On the host, **flick()** is a no-op.

Even though RX nodes have only one user process, **flick()** is still useful. When your application process relinquishes the processor with **flick()**, control returns to the NX operating system. If your RX process has set up a number of **hrecv()**'s and has nothing else to do, it should issue **flick()**. The operating system can then more efficiently respond to an incoming message and wake up your RX process.

All the system calls that take a pid number as an argument also work for RX nodes. When dealing with an RX node, however, you must always specify the node process's pid as 0.

The two wait calls, **waitall()** and **waitone()**, aid the synchronization of node processes. You can wait for a particular node process to complete with **waitall()**. For example:

```
waitall(1, 0);                                   Cversion
call waitall(1, 0)                               Fortran version
```

causes the executing process to wait until process 0 on node 1 completes. **waitall()** also takes the **-1** argument to indicate all nodes or all pids. For example:

```
waitall(-1, 0);                                  Cversion
call waitall(-1, 0)                              Fortran version
```

waits for process 0 on all nodes to complete.

A word of warning is in order here. If you specify "process 0 on all nodes" from within process 0 on any node, you will, of course, hang. The process would be waiting for itself to complete. Also, specifying all processes on all nodes may make sense in a host program, but do not issue it in a node program.

You can specify a set of processes with **waitone()** just like with **waitall()**, except that **waitone()** returns after the first process in the specified set completes. **waitone()** also returns information about the completed process. For example:

```
waitone(-1, 0, &cnode, &cpid, &ccode);          Cversion
call waitone(-1, 0, cnode, cpid, ccode)         Fortran version
```

waits for the first process 0 on any node that completes and then returns the node number of that process in *cnode*, its pid in *cpid*, and a completion code identifying the reason for completion in *ccode*. *cpid* is, of course, 0 in this example; but it would have been unknown if you had specified all processes. In C program, *ccode* contains the value of the program's *exit()* argument.

You can also kill a particular process on a particular node with the **killproc()** call. For example:

```
killproc(15, 3);                                Cversion
call killproc(15, 3)                            Fortran version
```

kills the process with NX pid 3 on node 15. A -1 chooses all nodes or all NX pids. For example:

```
killproc(-1, 0);                                Cversion
call killproc(-1, 0)                            Fortran version
```

kills process 0 on all nodes; and:

```
killproc(-1, -1);                               Cversion
call killproc(-1, -1)                           Fortran version
```

kills all processes on all nodes, including the node program that called it.

killproc() does not flush any messages related to the killed process. When you clear out a cube, you want to flush any messages left over after you kill all the node processes. One way to do this is to issue the following sequence from a host program:

```
killproc(-1, -1);                               Cversion
waitall(-1, -1);
flushmsg(-1, -1, -1);

call killproc(-1, -1)                           Fortranversion
call waitall(-1, -1)
call flushmsg(-1, -1, -1)
```

The first call sends out a kill message. The **waitall()** forces the host program to wait until all the node processes have been terminated. The **flushmsg()** removes any messages intended for the killed processes from the system buffers. For convenience, iPSC system software provides the **killcube()** system call, which is equivalent to the sequence of three calls just described.

Redirecting Input and Output

System Call	Environment
killsyslog()	host
newserver()	host
setsyslog()	host

Host process I/O is sent to the terminal or host file system as determined by the host program. Standard C and Fortran I/O calls may be used in your host program. You may also use standard UNIX redirection for host I/O.

Node to host I/O is handled by one or more iPSC system file servers running on the host. The default standard output and standard error of this file server is the terminal, but, as seen in Chapter 2, they may be redirected to a file when the **getcube** command is invoked.

The **newserver()** call kills current file servers and starts new ones, just like the **newserver** command. However, unlike a node program, when a host program does I/O, it doesn't go through an iPSC system file server. This means if you redirect node output with the **getcube** command or start a new file server, host output will not follow.

For example, assume that you allocate a cube and redirect node standard output to a file called *myfile*:

```
%getcube > myfile
```

Your intention is to create a log of your cube session in *myfile*. Node programs writing to their standard output will write to *myfile* instead of to the host's terminal, but host programs still go to the terminal.

This may not be what you want. If you also want host output to go to *myfile*, use **setsyslog()**. This call takes an integer argument. When the argument is 1, the standard output of the host is redirected to the standard output of the file server. When the argument is 2, the standard output of the host is redirected to the standard error of the file server. **killsyslog()** negates the redirection set up by the **setsyslog()** call.

For example, here is a host program written in C that redirects its standard output to the file server's standard output (assumed to be *myfile*) and then resets it to the terminal:

```
#include <stdio.h>
main()
{
   printf("This string appears on the terminal\n");
   fflush(stdout);
   setsyslog(1);
   printf("This string goes into the file myfile\n");
   fflush(stdout);
   killsyslog();
   printf("This string also appears on the terminal\n");
}
```

Handling Errors and Exceptions

System Call	Environment
handler()	**node**
_calls()	**host/node**

When a node process experiences a hardware exception, the default action is to print an error message and kill the process. With the **handler()** call, you can attach your own routine to a hardware exception. For example, the following statement attaches a user-written routine *proc0* to hardware exception 0:

```
handler(0, proc0);                    Cversion
call handler(0, proc0)                Fortran version
```

Note that the RX nodes may have a different set of exception numbers than CX nodes. Refer to the *iPSC®/2 and iPSC®/860 Programmer's Reference Manual* for more information.

Exception handling on the host obeys the standard UNIX convention. Consult the UNIX documentation, especially the **signal()** and **sigset()** calls, for information about exception handling. The default response to an exception is an error message and program termination.

If you are writing in C and you want to handle a software problem in line, use the underscore version of the iPSC system call. For example, **_getcube()** is the underscore version of **getcube()**. The underscore version returns -1 if the call encounters an error and a 0 or positive value if the call is successful. Fortran does not have underscore versions.

For example, if **getcube()** fails, the message:

```
(host) getcube: Cubetype not found
```

appears, and the host process terminates. If you use **_getcube()**, you can have your program test the return value and continue to try to allocate a cube in the hope that some other user may soon release one.

In addition, the underscore C routines set the system variable *errno*. This variable contains the error code. You can print out the error message with **perror()**.

MESSAGE PASSING

Message passing is the means of communication among processes in an iPSC system. As independent processor/memory pairs, the nodes do not share physical memory. In addition, even node processes running on the same node maintain distinct memory spaces. If the node processes need to communicate, they do so via message passing. Message passing calls can be synchronous or asynchronous.

Message Characteristics

Messages are characterized by a length, a type, and an id:

- The message length is measured in bytes. The message-sending routines will send exactly the specified message length. The message-receiving routines will place no more than the specified length into the receive buffer. It is an error if the receive buffer is not large enough to hold the message.
- The type is an identifier whose use is determined by the programmer.
- The id is an identifier used to check for the completion of asynchronous messages.

You can also send zero-length messages. Sometimes all your program is interested in is that it received a message of a particular type, and there is no need to supply any message content. Messages between nodes can be any length. Messages between the host and a node cannot exceed 256K bytes.

You determine the type of a message when you send it. The message receive routines also have a type selection parameter, allowing you to receive only messages of a particular type. This type selection parameter can also indicate a set of types, allowing you to receive only messages belonging to that set.

Synchronous and Asynchronous Message Passing Calls

System Call	Environment
crecv()	**host/node**
csend()	**host/node**
csendrecv()	**host/node**
irecv()	**host/node**
isend()	**host/node**
isendrecv()	**host/node**
msgdone()	**host/node**
msgwait()	**host/node**

SYNCHRONOUS SENDS AND RECEIVES

A synchronous send means that the program executing the send waits until the send is complete. This waiting is referred to as blocking. Completing the send, however, does not guarantee that the message has been received. It only means that the message has left the sending process and that the buffer can be reused. The synchronous send is **csend()**.

A synchronous receive means that the program executing the receive waits until the message arrives in the specified buffer. The synchronous receive is **crecv()**. A **csendrecv()** is like a **csend()** followed by a **crecv()**.

Here are two code fragments in C that perform a synchronous send and a synchronous receive. Note that for RX nodes, **mypid()** always returns 0.

Node 1 does a send:

```
#define MSG_TYPE 0
#define DEST_NODE 0
char send_buf[100];
   •
   •
   •
csend(MSG_TYPE, send_buf,
   sizeof(send_buf), DEST_NODE, mypid());
```

Node 0 does the receive:

```
#define MSG_TYPE 0
char rec_buf[100];
   •
   •
   •
crecv(MSG_TYPE, rec_buf, sizeof(rec_buf));
```

ASYNCHRONOUS SENDS AND RECEIVES

An asynchronous send or receive does not block. It returns a unique message id, which is not reused until released. You can use this id to check for completion at a later time. The asynchronous send is **isend()**, and the asynchronous receive is **irecv()**. An **isendrecv()** is like an **isend()** followed by an **irecv()**; it returns one message id.

To check if an asynchronous operation has completed, use the **msgdone()** call. It returns a 1 when the asynchronous call has completed and a 0 otherwise. You can also decide to block on the completion of an asynchronous call. Use **msgwait()** for this. Both **msgdone()** and **msgwait()** take the message id as an input parameter. The message id belonging to an asynchronous receive is distinct from the message id belonging to any companion asynchronous send.

Because an iPSC system has a limited number of message ids, you must take care to release ids that are no longer needed. There are three ways to release a message id. You can issue a **msgwait()**; you can keep issuing **msgdone()**s until a **msgdone()** returns a 1; or you can issue a **msgcancel()**. See the section "Flushing and Canceling Messages" (on page 3-16) for a discussion of **msgcancel()**.

For example, assume that your application knows that it's going to need a message up ahead. So it posts an asynchronous receive with **irecv()**. It then does work that does not require the message, believing that by the time it needs the message, it will have arrived. When the program comes to where it needs the message, it issues a **msgwait()**. If the message has in fact arrived, the **msgwait()** returns immediately. Otherwise, it blocks until the message arrives. Here is a Fortran code fragment that implements this technique.

Node 1 does an asynchronous send:

```
integer result, msg_sid
double precision send_buf(100)
parameter (MSG_TYPE = 1)
parameter (DEST_NODE = 0)
   •
   •
   •
msg_sid = isend(MSG_TYPE, send_buf,
   100*8, DEST_NODE, mypid())
```

```
      •
      •
      •
      call msgwait(msg_sid)          Frees the asynchronous send id.
```

Node 0 does the asynchronous receive:

```
      integer result, msg_rid
      double precision rec_buffer(100)
      parameter (MSG_TYPE = 1)
         •
         •
         •
C
C Post the receive
C
      msg_rid = irecv(MSG_TYPE, rec_buffer, 100*8)
         •
         •
         •
C
C Now you need the message.
C
      call msgwait(msg_rid)          Frees the asynchronous receive id.
C
C Now you have the message.  You may have blocked on the
C msgwait() if the message had not yet arrived.  You may now
C assign another value to msg_rid.
C
```

Pending Messages

System Call	Environment
cprobe()	**host/node**
iprobe()	**host/node**

When a message type arrives for which a receive has not been issued, it goes into a system buffer. It is referred to as a pending message, a message that is available for receipt, but not yet received. The message is received when you issue a receive for that message type. If a receive has already been issued, when the message arrives, it goes directly into the application's buffer and bypasses the system buffer. Figure 3-1 illustrates the relationship between the application and system buffers.

The **cprobe()** and **iprobe()** calls determine whether there is a message pending in the system buffer for your pid. If you call **cprobe()** or **iprobe()** with a type selector and a message is pending for a pid that is not the caller's, the probe call won't see it. The **cprobe()** call is a blocking call. It takes a type selection parameter as input and returns when a message has arrived. The **iprobe()** call is similar to **cprobe()**, except that it is nonblocking. **iprobe()** returns a 1 if the message is pending and a 0 if it is not.

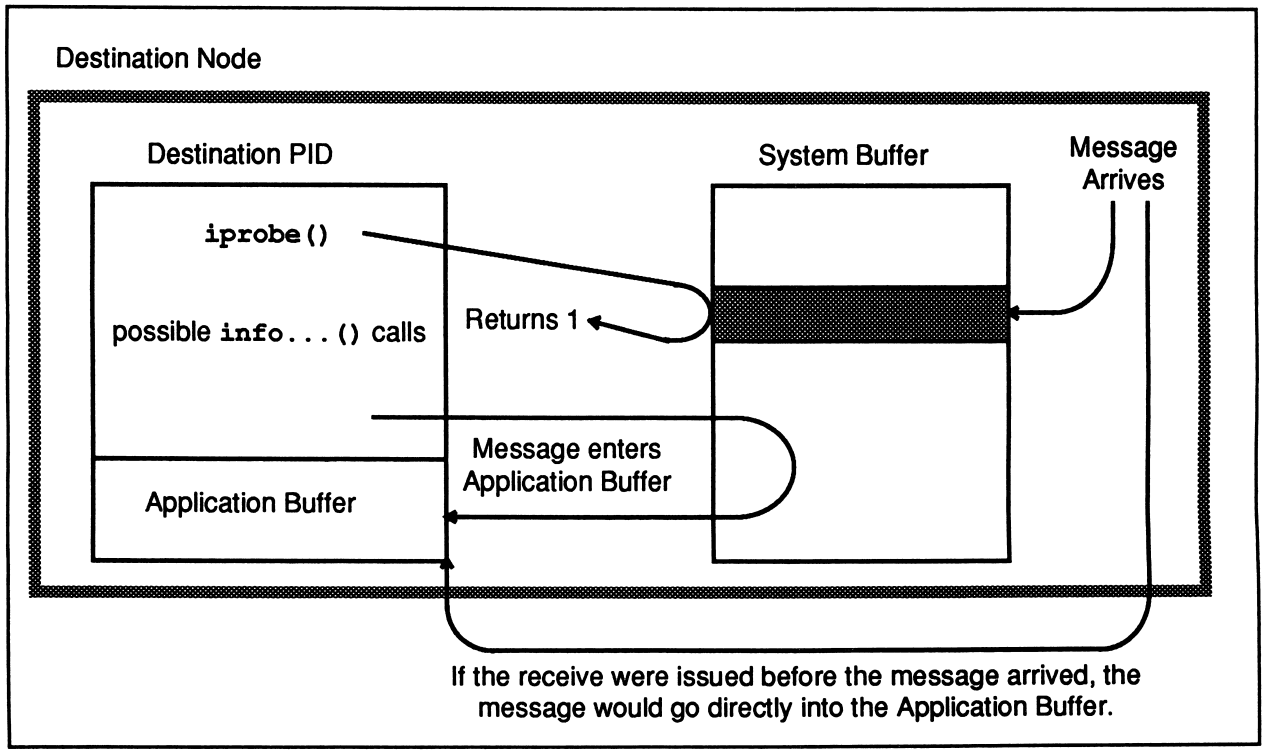


Figure 3-1. The Relationship Between the Application and System Buffers

Getting Information About Pending or Received Messages

System Call	Environment
infocount()	**host/node**
infonode()	**host/node**
infopid()	**host/node**
infotype()	**host/node**

The **info...()** calls return information about received or pending messages. You can obtain the size of the message, its type, and the node number and NX pid of the sending process. The NX pid of the sending process on an RX node is always 0.

For example, assume that you define a large array that you know is always larger than your largest message. You also know that you seldom fill that array up, but it's important that you know how much of the array you've used. You may want to add some data to that array later and need its present size as an offset. Here is a C code fragment that performs a synchronous receive and then issues **infocount()** to get the size of the received message:

```
#define BIGNUM 262144
long buf[BIGNUM], msg_type, msg_len;
    •
    •
    •
msg_type = 0;
crecv(msg_type, buf, sizeof(buf));
msg_len = infocount();
    •
    •
    •
```

The return value of the **info...()** calls is undefined except in the following cases:

- After a **crecv()**, **cprobe()**, or **msgwait()**
- After **iprobe()** or **msgdone()** returns a 1

If you want information about pending messages, you must issue the **info...()** call before you do another message passing operation, including internal message passing (for example, **printf()**)

For example, here is another C code fragment that blocks until any message arrives. Then, the code allocates a buffer just large enough to hold the message and receives the message.

```
char *buf;
long msg_type, msg_len;
   •
   •
   •
cprobe(-1);
msg_type = infotype();
msg_len = infocount();
buf = (char *) calloc(msg_len, 1);
crecv(msg_type, buf, msg_len);
   •
   •
   •
```

Flushing and Canceling Messages

System Call	Environment
flushmsg()	**host/node**
msgcancel()	**host/node**

If after inspecting a pending message with the **info...()** calls, you decide you don't want to receive it after all, you can get rid of it with **flushmsg()**. The **flushmsg()** call clears pending messages from the system buffer.

For example, here is a C code fragment that checks to see if a pending message has type 1; and if it does, flushes it. Otherwise, the program receives the message and continues.

```
long buf[100], msg_type;
   •
   •
   •
cprobe(-1);
msg_type = infotype();
if (msg_type == 1)
   flushmsg(1, mynode(), mypid());
else
   crecv(msg_type, buf, sizeof(buf));
   •
   •
   •
```

Note that **flushmsg()** only flushes messages pending to be received (that is, waiting in a system receive buffer), not those pending to be sent (that is, waiting in a system send buffer). That is, the call can be issued by either the sender or the receiver, but the parameters refer to the receiver. Specifically, the node id in the second parameter specifies the destination node, not the source node; and the node pid in the third parameter specifies the destination pid, not the source pid.

The first parameter is really a type selection parameter and not just a type. For example:

```
flushmsg(-1, mynode(), mypid());                    Cversion
call flushmsg(-1, mynode(), mypid())                Fortran version
```

would flush all messages waiting to be received by **mypid()** on **mynode()**.

msgcancel() cancels an asynchronous send or receive operation. For example, assume that you post an asynchronous receive and then determine that you don't want the message. Issue a **msgcancel()**. When **msgcancel()** returns, you don't know whether the receive operation completed, but you do know that the application buffer that you set up for the received message is no longer in danger of being written into.

Consider the following situation. The sender does an asynchronous send. The receiver has not yet issued a receive call. The message goes into a system buffer on the destination node. The sender then decides that it didn't want to send the message after all. The appropriate response is for the sender to issue a **msgcancel()** followed by a **flushmsg()**.

Why both? The message is either all or partially in a system buffer on the destination node. **msgcancel()** releases the sender's message id and gets rid of any partially sent message. Then, **flushmsg()** gets rid of the message if it has completely arrived in the destination system buffer. **flushmsg()** has no effect if the message has only partially arrived. Figure 3-2 illustrates this situation.

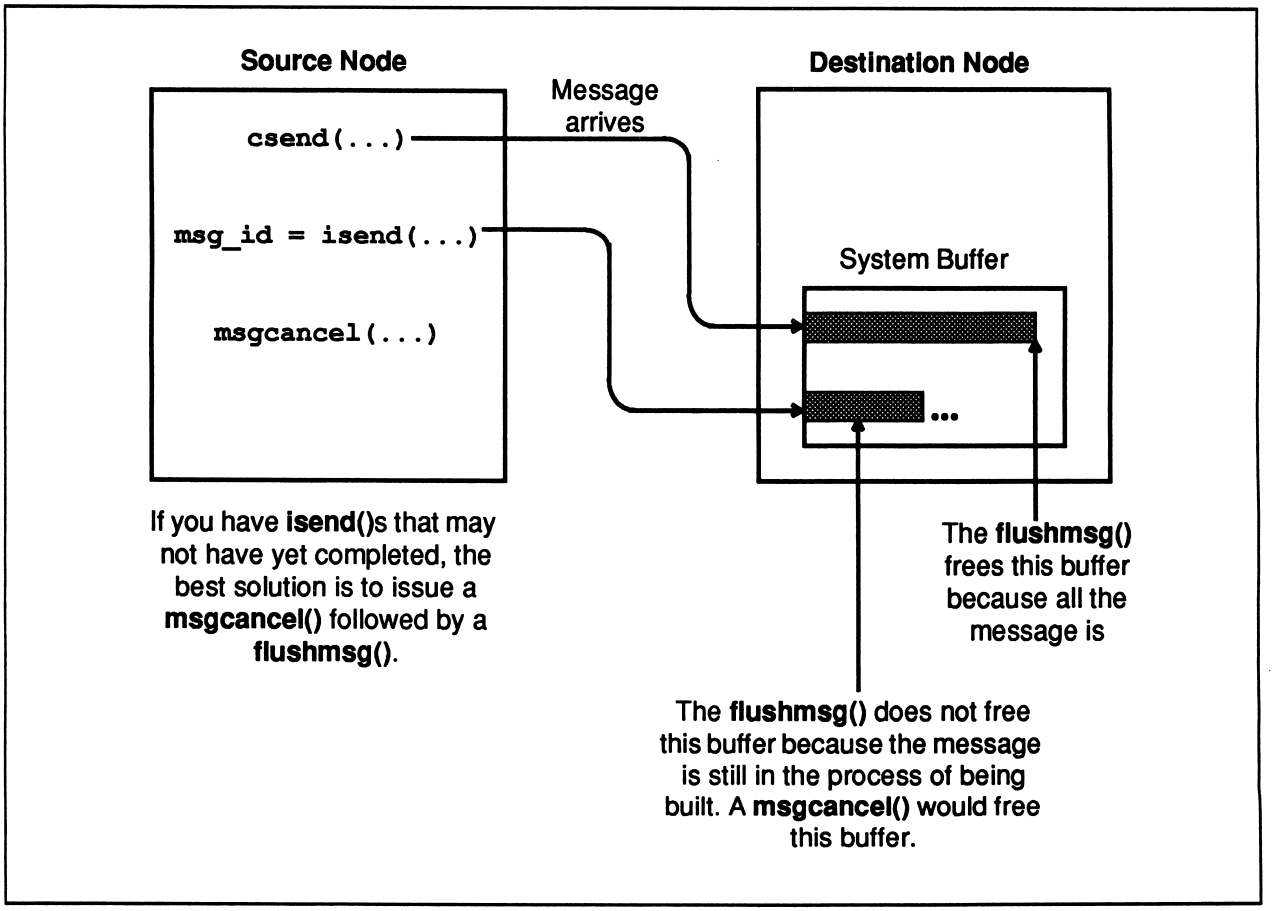


Figure 3-2. The Operation of flushmsg() and msgcancel()

Treating a Message as an Interrupt

System Call	Environment
hrecv()	**node**
hsend()	**node**
hsendrecv()	**node**
masktrap()	**node**

With the **h...()** message passing calls, you treat incoming messages as interrupts and attach a handler to the message's receipt. That procedure then is invoked when the message of that type is either sent or received.

For example, consider a node program that waits to receive a message and then performs some action based on the type of that message. One way to implement this program is to block the program at a **crecv()** for messages of all types and then do an **infotype()**. In C, you could then use a **case** statement to perform the appropriate action. In Fortran, you might use a computed **GOTO**.

Another way is to issue a number of **hrecv()** calls. Each call attaches a function to a particular message type or set of types. Your program does not block. You can continue with other work; but when the appropriate message comes, your program automatically stops what it was doing to take care of it.

The function you defined must have the following four arguments: *type*, *count*, *node*, *pid* (whether or not the function actually uses them). These arguments correspond to the values returned by the **info...()** calls.

For example, here's a C code fragment that attaches the functions *funct0()*, *funct1()*, and *funct2()* to message types 0, 1, and 2, respectively. The message types that have handlers are referred to as handler types.

```
char buf0[100], buf1[100], buf2[100];

hrecv(0, buf0, sizeof(buf0), funct0);
hrecv(1, buf1, sizeof(buf1), funct1);
hrecv(2, buf2, sizeof(buf2), funct2);
```

```
    •
    •                                   Now perform other work. No blocking happens.
    •
                                                          A message of type 1 arrives.
                                        Current processing is suspended, and funct1() is invoked.
                                You should not try to receive a message whose type is the same as a handler type.

void funct1(type, count, node, pid)
long type, count, node, pid;
{
        •                                                 The function attached to the type selection
        •                                                       must have these four arguments,
        •                                                  which correspond to the values returned
}                                                                  by the info...() calls.
```

hsend() operates the same as **hrecv**(), except that the handler is invoked when the message is sent rather than received. **hsendrecv**() is like an **hsend**() followed by an **hrecv**().

If you have one or more handlers set up and you have some critical code that you do not want interrupted, use the **masktrap**() call. A **masktrap(1)** masks all your handlers. A **masktrap(0)** re-enables them. Any pending interrupts are honored when the mask is removed.

GLOBAL OPERATIONS

System Call	Environment
gcol()	node
gcolx()	node
gdhigh()	node
gdlow()	node
gdprod()	node
gdsum()	node
giand()	node
gihigh()	node
gilow()	node
gior()	node
giprod()	node
gisum()	node
gixor()	node
gland()	node
glor()	node
glxor()	node
gopf()	node
gsendx()	node
gshigh()	node
gslow()	node
gsprod()	node
gssum()	node
gsync()	node

The global operations are high-level constructs for communication among node processes. For some applications, they are more efficient than the primitive message passing calls.

To illustrate the use of a global operation, consider the **gdsum()** call. This call is used by the pi example provided with the iPSC system software. This example evaluates pi by calculating a definite integral. The integral is partitioned among the nodes of a cube. The answer, then, is the sum of the answers from each of the participating nodes. Here's a code fragment from the C version of the example:

```
double partial_int, work;

/* The global sum appears in partial_int.  Work is
a temporary area used in the calculation.  1 is the
number of elements in the vector. */
    •
    •
    •
gdsum(&partial_int, 1, &work);
```

One way to get the answer is to have each node send a message to node 0 and then have the recipient perform the sum. This would require $(n\text{-}1)*t$ seconds, where n is the number of nodes in the cube and t is the time for one message. A **gdsum()**, however, requires only $d*t$ seconds, where d is the dimension of the cube. For example, if you have a 64-node cube, the first method requires 63*t seconds while the second requires only 6*t seconds.

To understand how the global communication calls (such as **gdsum()**) share their results, first consider the *e-cube* routing algorithm. (*e-cube* stands for edge of the cube. This has obvious meaning for two and three dimensions, but you have to use your imagination for the higher dimensions.) Communication between nearest neighbors occurs over a DCM channel. A DCM has eight channels, numbered 0 through 7. To determine the channel used by two nearest neighbors, take the exclusive OR of the node numbers. The channel number is then the bit position that holds the set bit.

Now, to simplify the problem, consider a four-node cube. Refer to Figure 3-3. In such a cube, each node has two nearest neighbors. If each node has a partial sum, node 0 requires three messages to get all the data that it needs to make the total. If you use a **gdsum()**, node 0, as well as every other node, gets its answer in two message times.

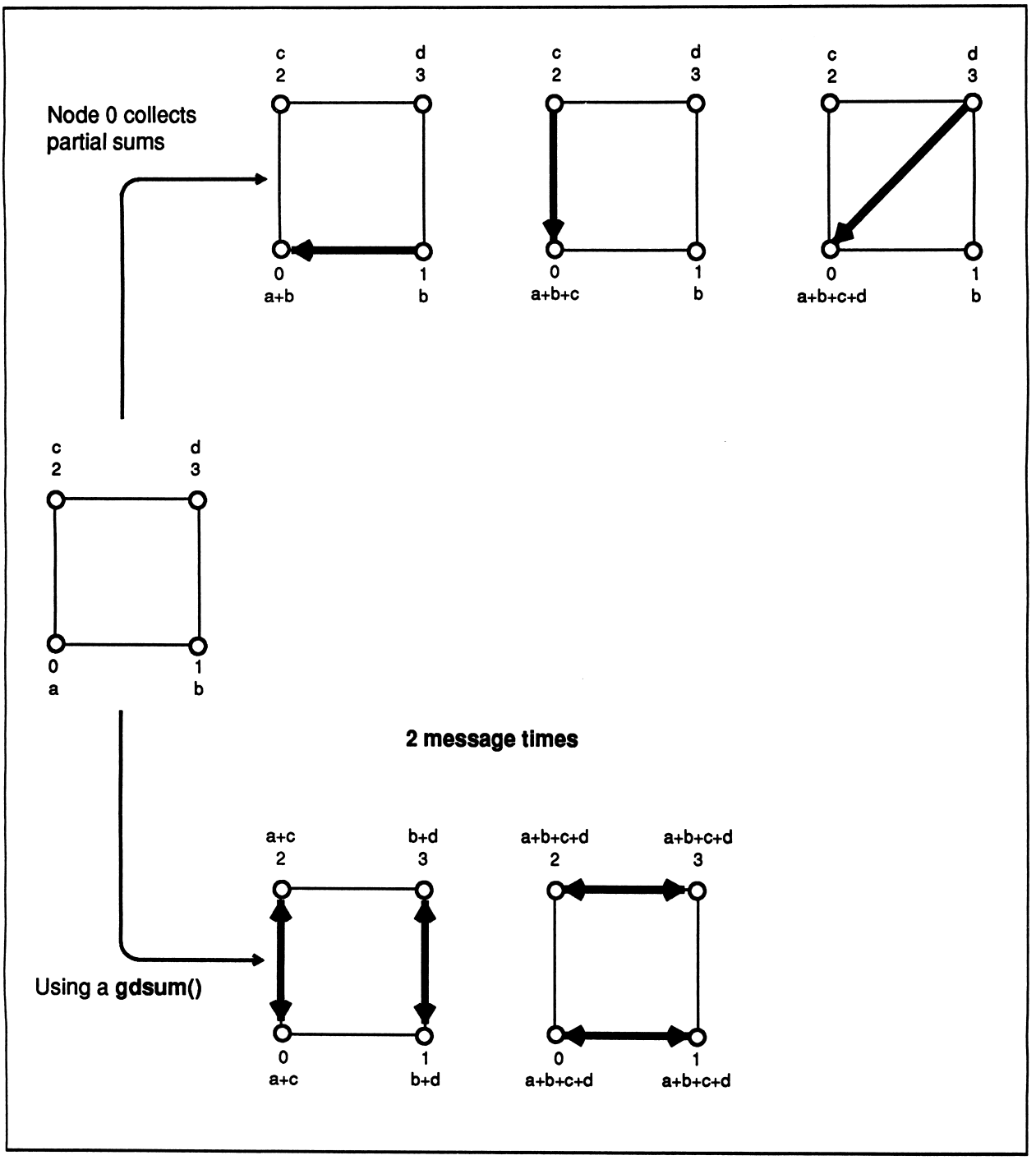


Figure 3-3. Operation of a gdsum()

For example, node 0 has nearest neighbors 1 and 2. The XOR of 00 (node 0) and 01 (node 1) is 01. The set bit is bit 0. Hence, channel 0 is used for communication between nodes 0 and 1. Similarly, nodes 0 and 2 communicate over channel 1.

The first message is a communication over the highest-numbered channel that is available for nearest neighbor communication (channel 1). Nodes 0 and 2 and nodes 1 and 3 exchange values. Because the channels are full-duplex, the exchange counts as one message. The next message is a nearest neighbor communication over channel 0. Nodes 2 and 3 and nodes 0 and 1 exchange values. Each node then has the complete sum.

MESSAGE PASSING WITH FORTRAN COMMONS

Because Fortran does not provide structures, users often use common blocks to send messages that contain data elements of different types. For example, consider the named common containing a double precision number and an integer. It is good Fortran practice to put the largest data element first in the common list.

```
integer i
double precision d
common/msg/ d,i
```

To send this common block, send the name of the first common element and specify the length of the entire common. For example, to send the common named *msg*, specify the variable *d*. The length is 12 bytes. This is the length for both Intel386 and i860 microprocessor-based computers. The following **csend()** call sends *msg* to process *pid* on node *node*.

```
call csend(MSGTYPE, d, 12, node, pid)
```

You may have to do some padding even if you are just sharing common blocks among Fortran routines. This occurs if two different routines on the same node share a common block with different views. Assume that one routine defines the common as follows:

```
integer i
double precision d
common/msg/ i, d
```

If the other routine wants to view the double precision number as a two-element integer array, it must pad after the first integer. Without the explicit pad, the i860 microprocessor will see three integers and not put in any padding. The other routine must define the common as follows:

```
integer i, ipad, id(2)
common/msg/ i, ipad, id(2)
```

BYTE SWAPPING

System Call	Environment
createstruc()	host (C only)
CTOHC()	host (C only)
CTOHD()	host
CTOHF()	host
CTOHL()	host
CTOHS()	host
HTOCC()	host (C only)
HTOCD()	host
HTOCF()	host
HTOCL()	host
HTOCS()	host
relstruc()	host (C only)

Use the byte-swapping calls when you send messages between the remote host and the cube. The byte-swapping calls are necessary if you're working on a remote workstation that doesn't use Intel's byte ordering convention. The Intel convention specifies that the least significant byte of an integer is stored at the lowest memory address. The byte-swapping calls have no effect when you are working on a workstation that uses Intel's byte ordering convention. However, you may decide to use them anyway to increase the portability of your code.

Each byte-swapping call takes two parameters. The first is the address of the data to be swapped; the second is the size of the data to be swapped. The unit of the size is the data type to be swapped. In other words, if the message is an array of two integers, the size is two, even though two integers may actually take up eight bytes.

Here is a code fragment for a host C program that converts an array of long integers on the host into the representation expected by the node. The byte-swapping calls obey a common naming convention. In C, HTOCL() means that a Host byte ordering is converted TO a Cube byte ordering, and the data consist of Long integers. Byte-swapping calls are also available for doubles, floats, shorts, and characters. In Fortran, HTOCL() means that a Host byte ordering is converted TO a Cube byte ordering, and the data consist of four-byte integers. Byte-swapping calls are also available for double precision numbers, real numbers, and two-byte integers. Note that the names of the byte-swapping calls are in upper case.

```
long datasend[100];
long datarec[100];
   •
   •
   •
```

```
HTOCL(datasend, 100);                                    Do the swap.
csend (NODE_TYPE, datasend,
   sizeof(datasend), -1, NODE_PID);                      Send the message to the node.
CTOHL(datasend, 100);                                    Swap the message back
   •                                                     (in case you need it again).
   •
   •
crecv (NODE_TYPE, datarec, sizeof(datarec));             Receive a message
CTOHL(datarec, 100);                                     from a node.
   •
   •
   •
```

In a C program, a message may be a structure consisting of different types. To convert a structure, ensuring that all its elements are aligned properly for a CX node, you must also use the **createstruc()** and **relstruc()** calls.

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

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

On the host, before you send this structure to the node, you must first define a character array whose size is *at least* large enough to accommodate the message structure, allowing for any padding that may be introduced to start each element on a four-byte boundary. These calls do not take into account the extra padding that occurs for i860 microprocessor-based nodes. Then, call **createstruc()** with that array as an argument and issue a byte-swapping call for each element of the message structure in the order in which it appears in the structure. Finally, close the structure with a **relstruc()**. This call returns the size of the padded message in bytes.

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

The following example starts with data in *out_msg_struct*. The host-to-cube calls copy the data to *out_msg_array*. The host then sends *out_msg_array* to the node. The node receives the message in the structure *in_msg*, copies it to the structure *out_msg*, and sends it back to the host. The host receives the message into *in_msg_array*. The cube-to-host calls put the message into *in_msg_struct*.

The host code appears as follows:

```
    •
    •
    •
struct msg_struct_tag  in_msg_struct, out_msg_struct;
char   in_msg_array[100], out_msg_array[100];
long   msg_size;
    •
    •
    •
setpid(100);
createstruc(out_msg_array);
HTOCC(out_msg_struct.a, 1);
HTOCS(out_msg_struct.s, 1);
HTOCC(out_msg_struct.b, 1);
HTOCL(out_msg_struct.l, 1);
HTOCC(out_msg_struct.c, 1);
HTOCF(out_msg_struct.f, 1);
HTOCC(out_msg_struct.d, 1);
HTOCD(out_msg_struct.ff, 1);
array_size = relstruc();

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

crecv(20, in_msg_array, array_size);

createstruc(in_msg_array);
CTOHC(in_msg_struct.a, 1);
CTOHS(in_msg_struct.s, 1);
CTOHC(in_msg_struct.b, 1);
CTOHL(in_msg_struct.l, 1);
CTOHC(in_msg_struct.c, 1);
CTOHF(in_msg_struct.f, 1);
CTOHC(in_msg_struct.d, 1);
CTOHD(in_msg_struct.ff, 1);
relstruc();
    •
    •
    •
```

On the host, the structure to be sent is out_msg_struct. This structure is copied to out_msg_array, which is sent to the node.

The host receives in_msg_array. The host then copies in_msg_array into the structure in_msg_struct. The bytes in in_msg_struct obey the host's byte-ordering convention.

The node code appears as follows:

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

USING GRAY CODES, BLINKING THE LED, AND TIMING EXECUTION

System Call	Environment
dclock()	**node**
ginv()	**host/node**
gray()	**host/node**
hwclock()	**node**
led()	**node**
mclock()	**host/node**

If your application uses gray codes, you may find the calls **gray()** and **ginv()** useful. **gray()** returns the gray code of an integer. **ginv()** is the inverse of **gray()**.

Each node has three LEDs: amber, red, and green.

- On a Intel386 microprocessor-based compute node, the amber LED lights up when floating point arithmetic (either on the Intel387 coprocessor or the SX processor) is taking place. On an i860 microprocessor-based compute node, the amber LED lights up when user (as opposed to system) code is executing. On an I/O node, the amber LED indicates that SCSI activity is going on.
- The red LED lights up when the node is communicating with the host or other nodes.
- The green LED indicates that processor activity (either system or user) is taking place.

The green LED is also user-programmable. To toggle the green LED use the **led()** call. For example:

```
led(1);                                          Cversion
call led(1)                                      Fortran version
```

turns the green LED on and:

```
led(0);                                          Cversion
call led(0)                                      Fortran version
```

turns it off.

hwclock() returns the value of the node's hardware counter as a 64-bit unsigned integer. This counter is set to 0 after a **bootcube** or **rebootcube**. The hardware count rate is defined in the constant **HWHZ**, which is defined in *cube.h* (for C) or *fcube.h* (for Fortran). For example, on an RX node running at 40 MHz, **HWHZ** is 10000000 because the counter is incremented after every four clock ticks.

In C, the value returned by **hwclock()** is a structure of type **esize_t** defined in */usr/include/estat.h.* In Fortran, it is returned as a two-element array of type **INTEGER**. Use the extended arithmetic calls (the **e...()** calls) to operate on this return value.

hwclock() is useful if you want accurate timings of small intervals. If you want to access the return value without the **e...()** calls, you must take into account that the high and low halves of the number are stored as signed.

dclock() also reads the hardware counter, but it translates the number into a double precision time in seconds.

mclock() is retained for compatibility with earlier versions of the NX operating system. It returns a relative time in milliseconds. In C, the number is an **unsigned long** (32 bits); in Fortran, the number is an **INTEGER** (also 32 bits). However, Fortran does not support unsigned numbers, causing **mclock()**'s return value to overflow in half the time.

Use these calls to return a relative value that you can use to measure execution time. To time an interval in your program, first obtain an initial value. Then obtain a final value and take the difference. The actual numbers returned by these calls are not important.

On the host, the clock calls measure the execution time of your program, any of its children, and any system operations initiated by your program or its children. On the node, they measure elapsed time (since the last **bootcube** or **rebootcube**).

Here is an example that shows how to time the execution of an iteration loop, using **dclock()**.

C version

```
double start_time, end_time, diff_time;
start_time = dclock();
for(i=0;i<imax;i++) {
        .
        .
        .
}
end_time = dclock();
diff_time = end_time - start_time;
printf("Timing = %e\n", diff_time);
```

Fortran version

```
      double precision start_time, end_time, diff_time
      start_time = dclock()
      do 100 i=1,imax
         .
         .
         .
```

```
100   continue
      end_time = dclock()
      diff_time = end_time - start_time
      write(*,10) diff_time
10    format('diff_time = ',D15.9)
```

Here is an example of using **hwclock()**. Note that **hwclock()**'s return value is an extended number, a 64-bit integer. Refer to the section, "Performing Extended Arithmetic," for more information about extended numbers.

C version

```
esize_t    start_time, end_time, diff_time;

char str_time[21];

hwclock(&start_time);
for(i=0;i<imax;i++) {
        •
        •
        •
}
hwclock(&end_time);
diff_time = esub(end_time, start_time);
etos(diff_time, str_time);
printf("Timing = %s\n", str_time)
```

Fortran version

```
      integer start_time(2), end_time(2), diff_time(2)
      character*20 str_time

      call hwclock(start_time)
      do 100 i=1,imax
         •
         •
         •
100   continue
      call hwclock(end_time)
      call esub(end_time, start_time, diff_time)
      call etos(diff_time, str_time)
      write(*,10) str_time
10    format('Timing = ', A20)
```

PERFORMING EXTENDED ARITHMETIC

System Call	Environment
eadd()	host/node
ecmp()	host/node
ediv()	host/node
emod()	host/node
emul()	host/node
esub()	host/node
etos()	host/node
stoe()	host/node

The primary use of the extended arithmetic calls is to manipulate the parameters for extended files in the Concurrent File System. An extended file is a file larger than 2G-1 bytes. Hence, some of its file parameters (like the file pointer and file size) must be represented as 64-bit integers.

64-bit integers are also called extended numbers. In Fortran, the extended numbers are stored in a two-element array of type **INTEGER**. In C, they are stored in a structure of type **esize_t** defined in */usr/include/estat.h*. This structure consists of two **longs**. *slow* contains the lower half of the integer; *shigh* contains the upper half. When at all possible you should use **e...()** calls to operate on the extended number, rather than access it internally.

Here's an example of adding two extended numbers.

```
e_sum = eadd(e1,e2);
```
C version

```
call eadd(e1, e2, e_sum);
```
Fortran version

Note that if you wanted to add one to an extended number, you can do it by creating your own structure or array.

```
esize_t e1,e2, e_sum;
```
C version

```
e2.slow  = 1;
e2.shigh = 0;
e_sum = eadd(e1,e2);
```

```
integer e1(2), e2(2), e_sum(2)
```
Fortran version

```
e2(1) = 1
e2(2) = 0
call eadd(e1, e2, e_sum)
```

But if you wanted to retain the extended numbers as abstract data types, you can use **stoe()**. This call converts a string into an extended number.

C version

```
esize_t e1,e2, e_sum;
char *one = {"1"};

e2 = stoe(one);
e_sum = eadd(e1,e2);
```

Fortran version

```
character*1 one
parameter (one ='1')
integer e1(2), e2(2), e_sum(2)

call stoe(one,e2)
call eadd(e1, e2, e_sum)
```

The other extended arithmetic calls allow you to subtract, multiply, and divide extended numbers. When you use **ediv()**, the divisor and answer must be **longs** or **INTEGERs**.

You can also compare two extended numbers and obtain a **long** or **INTEGER** that is modulo another **long** or **INTEGER. ecmp()** returns a -1, 0, or 1, depending on whether the first extended number is less than, equal to, or greater than the second.

DESIGNING A CONCURRENT APPLICATION 4

INTRODUCTION

Concurrent applications have varying degrees of parallelism. A perfectly-parallel application is one that requires no internode communication. In a perfectly-parallel application, if you double the number of nodes, you halve the computation time. Most applications involve a mix of computation and internode communication. One of the goals of parallel design is to develop a communication strategy that maximizes the time a node spends computing and minimizes the time it spends communicating.

This chapter describes some general design guidelines to follow when writing concurrent applications. But the best way to become skilled in concurrent programming is to do it. With that in mind, this chapter presents three examples of concurrent applications. Each example is intended to illustrate a different aspect of concurrent design technique.

- The first example is a nearly-perfectly-parallel application that evaluates a definite integral to calculate pi. This example illustrates how a sequential application can be ported to a parallel system with minimal effort. Much of the sequential algorithm can be maintained. The concurrent design consists of separating the user interface from the core computation and then distributing that core computation onto the nodes.

- The next example is the multiplication of a matrix by a vector. In addition to the numerical technique, this example illustrates the use of the concurrent file system by assuming a matrix that is too large to reside entirely in memory.

- The third example solves a classic computer science problem called the N-Queens problem. Given a chess board with N x N grid locations, where can you place N queens so that no queen is under attack? This example illustrates a technique called control decomposition. This technique also appears in more complicated real-world applications such as electronic design rule checking.

The concurrent programming model has the following characteristics.

- There is an ensemble of processor/memory pairs called nodes that make up a cube. They do not share memory. They can access the same file system, but they operate independently of each other.
- All the nodes are fully connected. They communicate with each other and the host by passing messages.
- Each node executes its own program. In many applications, it turns out that each node executes the same program on a different set of input data. There may be some conditional code that identifies one or more nodes that perform special actions.

Separating the User Interface from the Computation

The goal is to move to the nodes as much of the computation as possible. To do this, analyze the algorithm and separate the user interface from the computational kernel. This user interface will most likely end up in a host program, or you may choose to designate one of the nodes to handle the user interface.

In the pi example, a host program requests the number of integration intervals. It then sends that number to all the nodes who then do the calculation.

Load Balancing

Your goal is to keep all the nodes busy and have them finish at the same time. This obviously makes the best use of a parallel computer. You're wasting cycles if some nodes have to wait for others to finish. Analyze your application and distribute the computation among the nodes so that their computational load is evenly balanced.

Distributing a problem among the nodes is referred to as *problem decomposition*. A decomposition can be a *domain decomposition* or a *control decomposition*.

DOMAIN DECOMPOSITION

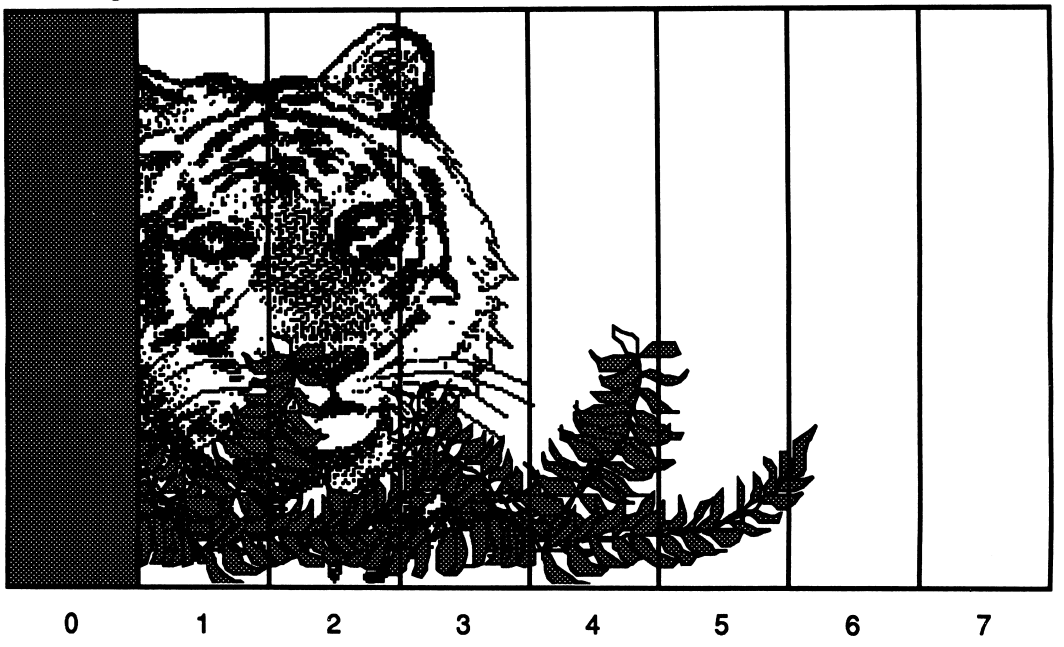
In domain decomposition, the input data (the domain) are partitioned and assigned to different processors. How you divide and distribute the data among the nodes can have a significant effect on the efficiency of your application.

For example, consider an application that performs image enhancement (see Figure 4-1). Because some parts of the image may be more detailed than others, they will require more processing. If you divide the image sequentially among the nodes (as shown in the top half of Figure 4-1), some nodes may get a partition that requires a lot of work and other nodes may get a partition that requires little or no work at all. As the top half of Figure 4-1 shows, node 0 gets a lot of work and node 7 gets no work at all. This is inefficient.

You can achieve better load balancing by dividing the image into smaller partitions and then distributing the partitions sequentially among the nodes (as shown in the bottom half of Figure 4-1). This is analogous to dealing out the partitions like cards in a deck; it spreads out the work more evenly among the nodes. As the bottom half of Figure 4-1 shows, each node gets some slices that require a lot of work, some slices that require a moderate amount of work, and some slices that require no work. This is more balanced and efficient.

(The shaded portion of Figure 4-1 shows the work done by node 0.)

Poor load balancing: Nodes 0 through 3 get most of the work. Nodes 4 through 7 have little or nothing to do.



Good load balancing: The partitions in the domain are dealt out to the nodes like cards from a deck. Now, each node has approximately the same amount of work.

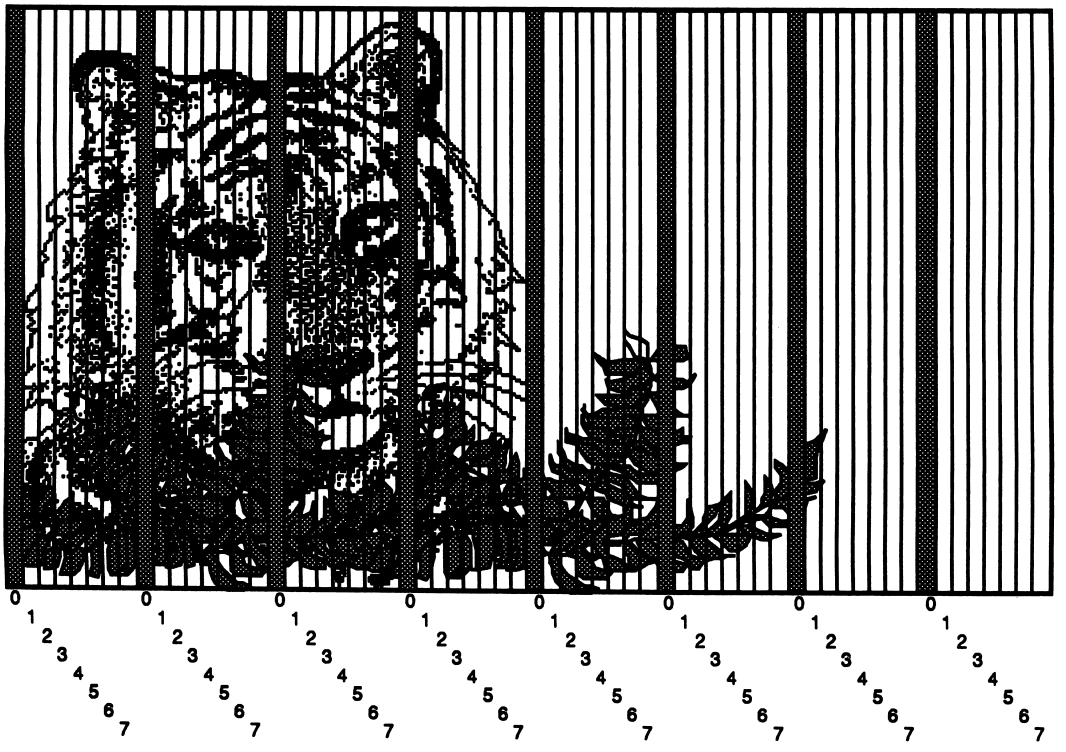


Figure 4-1. Using Domain Decomposition to Achieve Load Balancing

CONTROL DECOMPOSITION

Control decomposition, on the other hand, divides the *tasks* to be performed rather than the *data*. For many problems, this is a more natural decomposition.

For example, consider a tree-search used in a game playing algorithm. Assume that you're at some mid-level of the tree. You could approach the problem as a domain decomposition and divide the current branches among the nodes. Each node would then follow its branch down to the leaves and then return the leaves as an answer. The leaves in this case are the possible moves. Depending on the current state of the game, some of the branches may be quite involved and require a great deal of processing. Other branches may be simple. The result is that some nodes finish before others. This is a poor problem decomposition.

Approaching the problem as a control decomposition achieves better load balancing. In a control decomposition, you think of the branches not as data partitions but rather as tasks that need to be performed.

To manage these tasks, you have to introduce a little bureaucracy. Assign one node as a manager node. This manager node then gives tasks to idle nodes. When the node finishes a task, it reports its answer and requests another task. It's this "reporting for duty" that characterizes a control decomposition.

The manager node must, of course, do some initial setup. For example, it may follow the tree down until the number of branches exceeds the number of available nodes by some predetermined factor.

This method produces the best results when the tasks assigned near the end of the problem are about the same size. For example, if one of the last tasks assigned was a very long task, the other nodes may be idle while that last node finishes.

The N-Queens example (presented later in this chapter) shows control decomposition.

Designing a Communication Strategy

Your goal is to design your internode communication such that the nodes spend as little time in communication as possible. This may involve running some tests to determine an optimal message length. Often, you can decrease the number of messages by increasing the size of each message.

In addition, consider the use of the global operations described in Chapter 3. That chapter described a simple example of a global sum. Using **gdsum()** results in a significant improvement over having one node perform the global sum by explicitly collecting all the partial sums. Also, after the execution of the **gdsum()**, the global sum is available on each node.

The matrix*vector example in this chapter uses another global operation called **gcol()**. In that example, a large vector is distributed over the nodes. **gcol()** collects the components from each node and constructs the complete vector on each node. As with **gdsum()**, the answer is available on each node.

Because the nodes in an iPSC system are fully connected, there aren't any performance factors to consider when you decide what node is going to communicate with what other node. But you may find that the nature of your application lends itself more readily to certain node topologies.

For example, a common node topology is the ring. This topology is useful in certain types of many-body calculations. The technique consists of partitioning the particles into groups and assigning each group to a different node. A node then calculates the state of its group. This state information is then passed to another node which calculates the state of its own particles and takes into account the state received from the previous node. The state information moves from node to node around a ring.

Generalizing the Number of Nodes

Your goal is to write your application so that you can add more nodes to your cube, thus improving performance, without having to recode.

This method also turns out to be the most natural one to use when porting an existing sequential application. After you've separated the user interface from the core computation, you still have a sequential algorithm, but you can think of it as the special case of a cube with one node, a dimension 0 cube.

The pi example illustrates this technique. The number of nodes appears only as the variable *nodes*.

EXAMPLE APPLICATION: CALCULATING PI

This application uses an n-point quadrature rule to evaluate the definite integral,

$$\pi = \int_0^1 4 / (1 + x^2)dx$$

Admittedly, using the power of an iPSC system for such a simple application is overkill, but the application demonstrates concepts that are just as valid for more challenging problems.

The application consists of a host program that interfaces to the user and loads the same node program on each of the allocated nodes. Each node performs a portion of the integration. This is a domain decomposition. It is illustrated in Figure 4-2. Because the integration work is evenly balanced, there is no need to "deal out" portions. However, because the data are calculated, it is just as easy to deal out as not

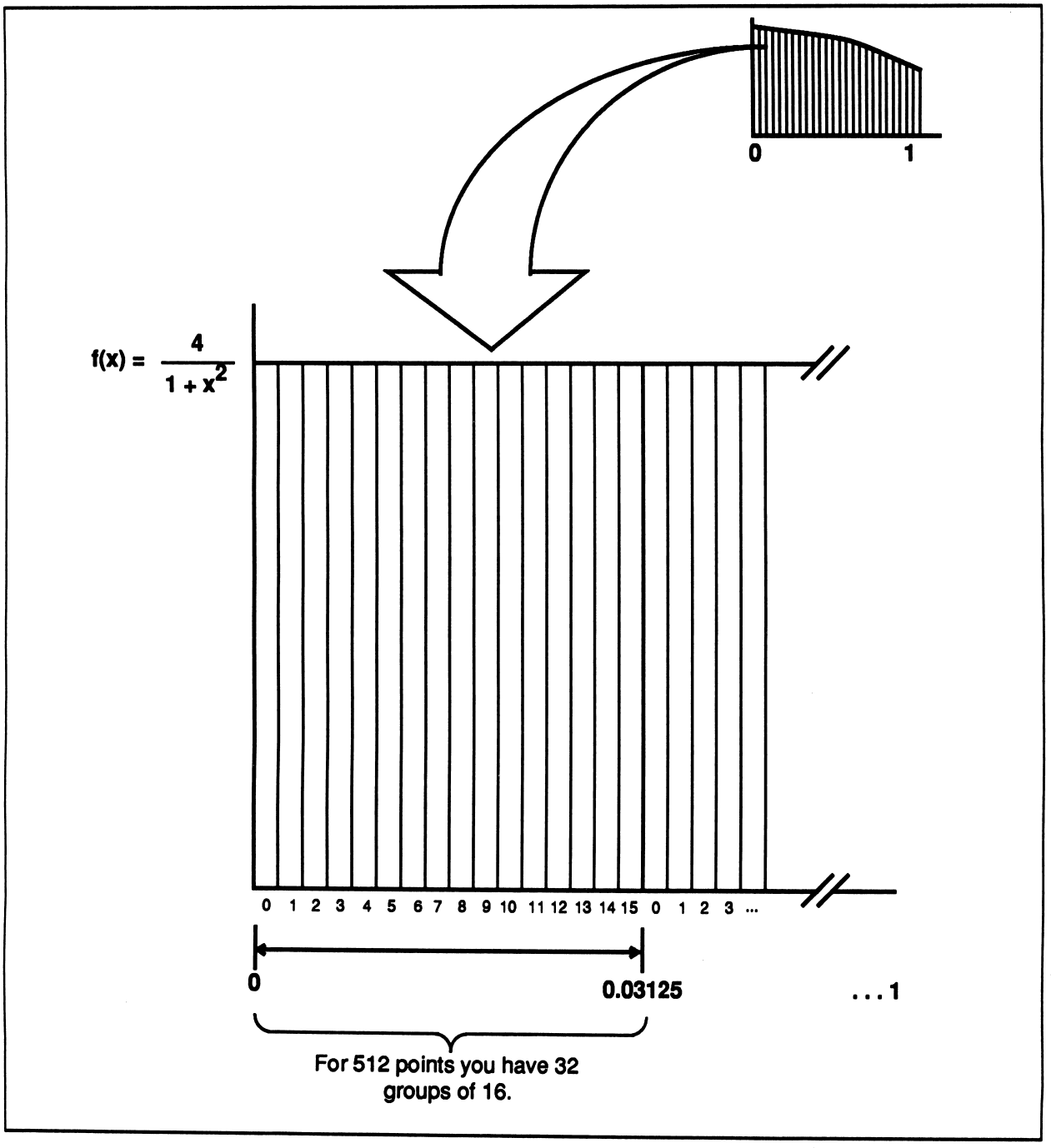


Figure 4-2. The Decomposition Used for the Pi Example

For example, if you choose 16 nodes and 512 points, each node gets 32 points. The first 16 points go to each of the 16 nodes. Then, the second 16 points go to each of the 16 nodes. And so on.

Figure 4-3 shows a sequential version of the pi example (written in Fortran). Note that the user interface consists only of a read statement that solicits the number of intervals.

```
      program pi
      double precision h, sum, x, pi, f, a
      integer n
c
c Define the function.
      f(a) = 4.d0/(1.d0 + a*a)
c
c Input the number of intervals.
1     print *,' Enter number of intervals:'
      read(5,*,end=100) n
c
c Calculate scaling factor.
      h = 1.d0/n
c
c Integrate.  The value of x used to calculate y is
c the value at the midpoint of the integration slice.
      sum = 0.d0
      do 10 i = 1,n
         x = h *(dble(i) - 0.5d0)
         sum = sum + f(x)
10    continue
      pi = h * sum
c
c Output the answer
      print *,' The value of pi for',n,' intervals is',pi
      go to 1
c
c Terminate
100   stop
      end
```

Figure 4-3. Calculating Pi: Fortran Code for a Sequential Version

Figures 4-4 and 4-5 show the host code and node code, respectively, for a parallel version. Note that the parallel version is not much longer than the sequential version. In the parallel version, the host program solicits the number of intervals, loads the computational part onto the nodes, sends the data to the nodes, receives the answer, and displays it on the screen.

Note also, that the decomposition takes place entirely in the **do** statement. The sequential version is:

```
i = 1, n
```

while the parallel version is:

```
i = iam+1, n, nodes
```

If you add more nodes to your cube, you don't have to change one line of the node program!

```
      program pi_host
      double precision pi
      integer n
c
c Cube stuff
      integer allnodes, npid
      integer intsiz, dblesiz
      data allnodes, npid / -1, 0/
      data intsiz, dblesiz / 4, 8/

      call setpid(npid)
      call load('pi_node', allnodes, npid)

c
c User interface: input the number of intervals
c                 send this number to all nodes
c                 receive the answer from node 0
1     print *,' Enter number of intervals:'
      read(5,*,end=100) n
      if(n .le. 0) go to 100
      call csend(100,n,intsiz,allnodes,npid)
      call crecv(200,pi,dblesiz)
c
c Output the answer
      print *,' The value of pi for',n,' intervals is',pi
      go to 1
c
c Terminate
100   call killcube(allnodes, npid)

      stop
      end
```

Figure 4-4. Calculating Pi: Fortran Host Code for a Parallel Version

```
      program pi
      double precision h, sum, x, pi, f, a, tmp
      integer n
      integer allnodes, npid, nodes, iam, host
      integer intsiz, dblesiz

      data allnodes / -1/
      data intsiz, dblesiz / 4, 8/
c
c Define the function
      f(a) = 4.d0/(1.d0 + a*a)
c
c Do some node bookkeeping
      iam = mynode()
      nodes = numnodes()
      npid = mypid()
      host = myhost()
c
c Receive the number of intervals from the host
1     call crecv(100, n, intsiz)
c
c Calculate scaling factor
      h = 1.d0/n
c
c Integrate
      sum = 0.d0
      do 10 i = iam+1,n,nodes
         x = h * (dble(i) - 0.5d0)
         sum = sum + f(x)
10    continue
      pi = h * sum
c
c Use a global sum function to collect all the partial sums.
      call gdsum(pi, 1, tmp)
c
c Node 0 sends the answer to the host.
      if(iam .eq. 0) then
         call csend(200,pi,dblesiz,host,npid)
      endif
c
c This is an infinite loop.
c Do it again until the host says to stop.
      go to 1
      end
```

Figure 4-5. Calculating Pi: Fortran Node Code for a Parallel Version

EXAMPLE APPLICATION: MATRIX*VECTOR MULTIPLICATION

This application computes the matrix vector product $y = Ax$ where A is an n x n matrix and x and y are vectors with n components. For simplicity, n is assumed to be divisible by p, the number of processors in the cube. There are then $m = n/p$ rows per node.

Also, A is assumed to be too large to fit in a node's memory. That is, the multiplication is not an "in-core" multiplication. Instead, A is stored by rows in a concurrent file called */cfs/example/matrix*. That is, the matrix is stored in memory as follows:

A(1,1), A(1,2), ... A(1,n), A(2,1), A(2,2)... A(2,n), ... A(n,1,), A(n,2)...A(n,n)

The vector x is originally stored in blocks on each processor. Each node contains n/p components of x. Node 0 has components 1 through n/p; node 1 has components $n/p + 1$ through $2*n/p$, etc. The answer, the vector y, will be stored in the same way.

The problem decomposition is again a domain decomposition. Each node collects all of x, but then takes only a portion of A (actually n/p rows) to form its portion of the product vector. There is no attempt to "deal out" the rows of A.

This example uses the two BLAS (Basic Linear Algebra Subroutines) routines **scopy()** and **sdot()**. These routines are part of VecLib. You must compile and link the node program with one of the vector switches. For an Intel386 microprocessor-based node, use either **-vx** or **-vec**. If you have VX nodes, use **-vx** on the compile step and **-vec -vx** on the link step. Use **-vec** on the link step if you have Intel386 microprocessor-based nodes without the VX option or you have i860 microprocessor-based nodes.

The routine **scopy()** takes the vectors $x_0(1..m)$ through $x_p(1..m)$ from each node. The subscript indicates the node number. The routine forms *xtotal(1..n)* on each node. Then the program reads m rows. Each row has n elements of length *REALSIZE*, and so m rows take up $m*n*REALSIZE$. For each node, the file pointer is offset by *REALSIZE*mynode()*n*m*. The routine **sdot()** does the actual multiplication.

Figure 4-6 is a Fortran code fragment that illustrates how a node program would collect the vector x, open the concurrent file, and perform its portion of the calculation.

```
      subroutine matvmul(m, n, x, y, xtotal, arow)
      integer REALSIZE
      parameter(REALSIZE = 4)
      integer ncnt, fileptr
      real x(m), y(m), xtotal(n), arow(n)
c
c  m is n/p where n is the dimension of A
c  and p is numnodes()
c
c  Collect all of x on each node.  Use the veclib call scopy()
c  and the global operation gcol().
      call scopy(m, x, 1, xtotal, 1)
      call gcol(x, REALSIZE*m, xtotal, REALSIZE*n, ncnt)
c
c  Open the concurrent file
c  and seek to the appropriate location

      open(unit=10, file = '/cfs/example/matrix',
     +     form = 'unformatted')
      fileptr = lseek(10, REALSIZE*mynode()*n*m, 0)
c
c  Read the rows and use the veclib call sdot() to do
c  the multiplication.
      do 10 i = 1, m
         call cread(10, arow, n*REALSIZE)
         y(i) = sdot(n, arow, 1, xtotal, 1)
10    continue
   .
   .
   .
```

Figure 4-6. Matrix Vector Multiplication: Fortran Code Fragment

EXAMPLE APPLICATION: THE N-QUEENS PROBLEM

This application collects all the board configurations that solve the N-Queens problem. This problem is: "Given an N x N chess board, where can you place N queens so that each queen is safe from capture?" In chess, queens attack in straight lines along the X, Y, and diagonal directions.

The N-Queens problem is typical of problems for which there is no analytical solution. Instead, there exists a large set of candidate solutions. You test each solution and accept those that pass.

The difficulty lies in the enormous size of the candidate set. For example, an 8 x 8 chess board has 64 squares. The total number of possible positions for 8 queens can be represented as the *combination* of n=64 things taken m=8 at a time. The formula for the number of combinations is:

```
n! / ( m!* (n-m)! )
```

which evaluates to 2^{32} possibilities. Even on a state-of-the-art sequential computer, it would take several hours to check every one of those combinations.

Even before you begin thinking about an algorithm, however, you can eliminate a large number of possibilities. For example, any solution that has more than one queen in the same column is unacceptable. This reduces the number of possibilities to 8^8 or 2^{24}. On a fast sequential computer this takes a minute or two.

This section shows how to use an iPSC system to evaluate those 2^{24} possibilities. You can arrange the possibilities into a tree. The technique involves following a tree down until it either reaches a dead end (an invalid state) or until it reaches a leaf (a valid solution). Figure 4-7 illustrates such a tree. To make the figure simpler, the chess board is shown as 4 x 4. Instead of 2^{24} possibilities, you have 2^8.

The root of the tree (the zero level) is the null board — no queens present. The next level (the first level) consists of states where a queen is in each of the positions that make up the first column. In Figure 4-7, there are four of those. In an 8 x 8 board, there would be eight.

The next level (the second level) consists of states with two queens on the board, one in the first column and one in the second. In Figure 4-7, there are four of those under each second level state. Notice, however, that some states are already invalid. There is no need to follow the tree any further down this branch. In Figure 4-7, the two leftmost states in the second level are invalid. The second state in the first level has three dead ends in its second level.

You can see how the algorithm is going. Some paths are going to finish early because they reach dead ends. Others are going to take longer and reach the solutions at the leaves. This is a problem for control decomposition.

The manager node follows the tree down until the number of states is larger than the number of available nodes. It turns out that a number of states equal to about five or ten times the number of nodes gives best results. As a further enhancement, the manager node may even enlist the aid of the other nodes when doing this initial processing.

Then, the manager node assigns a state to a node. The node follows that state down the tree and collects all the possible solutions. When, the node finishes it reports its solutions, if any, to the manager and requests more work. In the case of a 4 x 4 board, the tree is shallow and there are only two solutions. An 8 x 8 board results in 92 solutions.

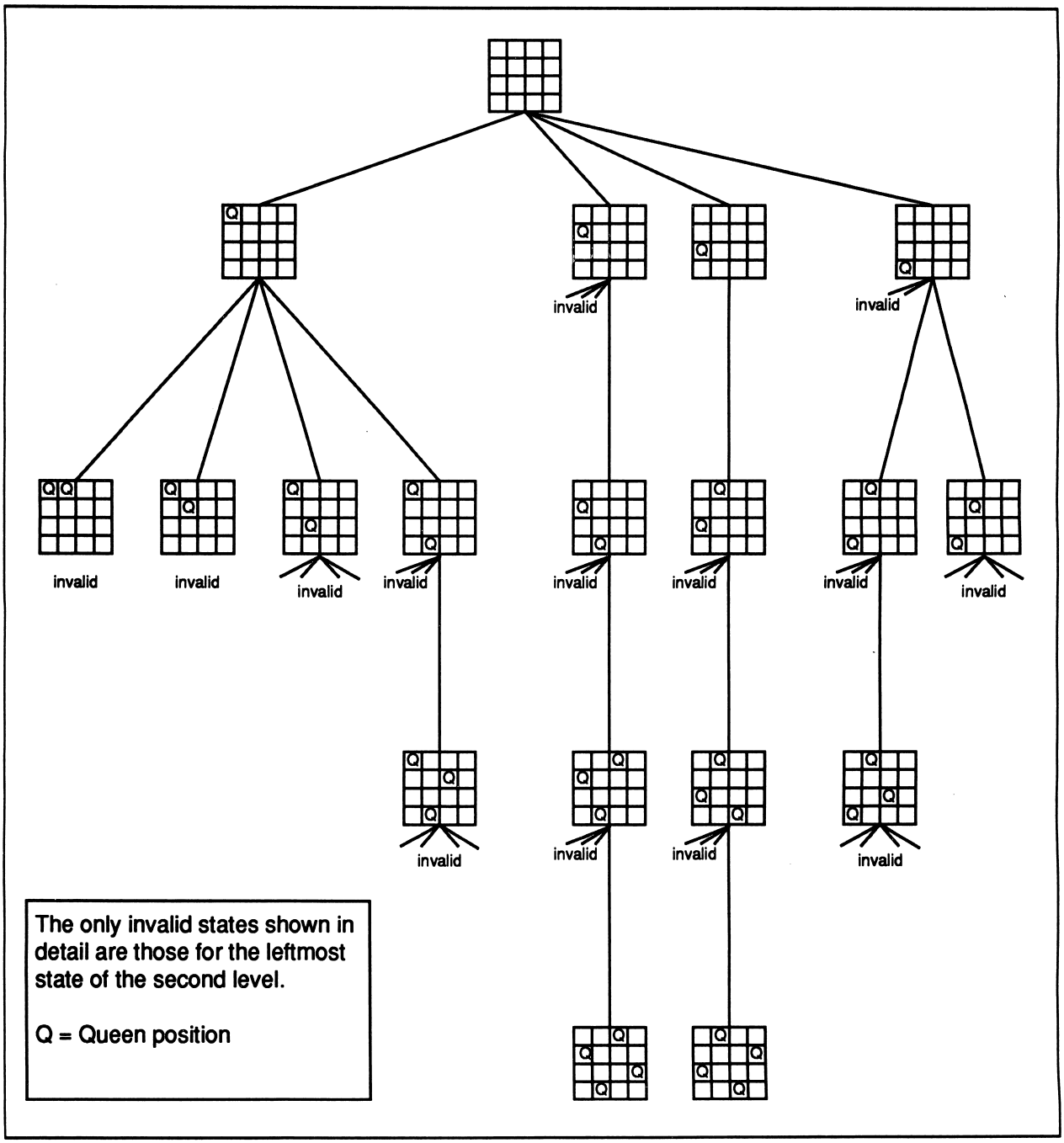


Figure 4-7. The N-Queens Solution Tree for a 4 x 4 Board

PROGRAMMING TECHNIQUES 5

INTRODUCTION

This chapter presents some techniques for improving the performance of already debugged parallel programs.

MEMORY ACCESSES

RAM memory is configured in 4K-byte pages. If you reference the same 4K-byte page repeatedly memory accesses to that memory page are much faster than if your references change pages. There is also an 8K-byte data cache on the i860 microprocessor. If you reference locations repeatedly that are contained in this cache, access is very fast. These are two different memory functions that have different sizes and different effects.

Try to access contiguous locations in memory (applicable to CX and RX nodes running any language). This has different effects for different languages. In C, the rightmost index of an array varies the fastest while in Fortran the leftmost index varies fastest. For example, if you partition a two-dimensional array, you would partition rows in C and columns in Fortran.

Accessing contiguous memory locations allows the microprocessor to minimize page translations and to make better use of any address translation lookaside buffers (the TLB). The 4K-byte memory page size is a function of memory design and is independent of data cache operation. When you repeatedly access locations within a memory page, you access data much faster than when you change pages to access data.

To keep your memory accesses within a page, you can use some of the following techniques:

- Do consecutive references down the rows or columns of matrices.
- Group a series of memory reads out of the same array.
- Strip-mine loops, so that you do several accesses at a time. For example, you read four elements of vector A, then four of B, instead of reading A[1], B[1], A[2], B[2]. . . .

Try to access memory locations that are already in the data cache. References to this cache can be for locations that are scattered throughout memory. Because they have been referenced previously, their contents are held in cache (they are "cached"). As long as those locations are cached, access to them is very fast.

MESSAGE PASSING

Message passing is an integral part of all parallel applications. These applications consist of both computation and communication. You achieve the best performance when you maximize computation and minimize communication. The following paragraphs describe some techniques for doing that.

Align Application Buffers (CX and RX Nodes Running C and Fortran)

The most important programming practice for the iPSC/860 system is to ensure that send and receive buffers are properly aligned and sized whenever possible. Although the message-passing system calls will work with any size or alignment of buffers, the hardware works only with well-aligned buffers. As a result, the software must copy messages which are in misaligned buffers, decreasing performance.

Ensure that the message's application buffers are properly aligned. On CX nodes, send buffers can be any size, but should be aligned on 4-byte boundaries. Messages greater than 100 bytes benefit most from this alignment. Receive buffers should always be aligned on 4-byte boundaries and should be a multiple of four bytes if the message is longer than 100 bytes.

Receive buffers should be no more that 4096 bytes larger than the message. This size refers to the length parameter in a send or receive call. The buffer may be declared to be larger than the length.

In Fortran, avoid using **CHARACTER** or **LOGICAL*1** buffers if **INTEGER**, **REAL**, or **DOUBLE PRECISION** will do. If you have performance-sensitive **CHARACTER** or **LOGICAL*1** buffers, equivalence them to an **INTEGER** and try to make them a multiple of four bytes.

In C, declare buffers as **int** or **long**, rather than **char** or **short**. If it is more convenient to declare the buffer **char** or **short**, make sure the size is a multiple of four bytes and that other **char** or **short** arrays are also multiples of four bytes. Buffers allocated with **malloc()** or its derivatives will be correctly aligned.

Avoid Message Buffering (CX and RX Nodes Running C and Fortran)

Try to avoid message buffering whenever possible. For example, assume that you have the same process running on two nodes. These processes must exchange information. Each process must issue a receive and a send. If you code the **csend**()'s before the **crecv**()'s, the messages are sent and buffered in a system buffer. When the **crecv**()'s are executed, the messages are copied from the system buffer into the application buffer.

Your code runs more efficiently if you can avoid the system buffer and copy the message directly into the application buffer. This happens if you issue **irecv**()'s before the **csend**()'s. For example, consider the following C routine, *shadow*. This routine might appear in an application that needs to have the nodes exchange the rows of a matrix. A Fortran version would probably exchange columns (refer to the section, "Memory Accesses" (on page 3-1)).

A typical application might be a Gauss-Seidel iteration or any technique based on nearest neighbor interactions. The application processes a two dimensional array called *s*[ ][ ] and exchanges rows between nodes. The first index in the array represents a row, and it is passed as a pointer. Each node contains a horizontal partition of the array with *range* rows. It has a top buffer (*s*[0]) and a bottom buffer (*s*[*range*+1]) containing the boundary rows from other nodes.

```
void shadow(topnode,botnode,s,range)
long topnode, botnode;
int (*s)[MAX_LATTICE], range;
{
   long topid, botid;

   topid = irecv(TOP,  s[0],       sizeof(s[0]));
   botid = irecv(BOT,  s[range+1], sizeof(s[range+1]));

   csend(BOT, s[1],     sizeof(s[1]),     topnode, 0);
   csend(TOP, s[range], sizeof(s[range]), botnode, 0);

   msgwait(topid);
   msgwait(botid);

/* Node sends upper boundary row s[1] to the bottom buffer
s[range+1] of the node controlling the upper partition,
(topnode).

Node sends lower boundary row s[range] to the top buffer s[0] of
the node controlling the lower partition, botnode
*/
}
```

Note that when the sends are performed the receives have already been posted. The data then go directly into the application buffer. This, of course, assumes that the nodes are running in synchronization. To ensure synchronization, you might consider placing a **gsync**() after the **irecv**()'s.

Use Static Buffers (CX Nodes Running C)

Use static buffers to improve performance. If you are coding in C and you have large messages, make your application buffer **static**. This may be beneficial if your messages are much larger than 4K bytes. Buffers declared as **static** are more likely to have contiguous 4K pages. Please note that this is an improvement of static buffers vs. automatic buffers, not static buffers vs. buffers allocated dynamically with a call such as **malloc()**. You may, however, notice some slight improvement if your message buffers are not dynamically allocated.

ERROR CHECKING (CX AND RX NODES RUNNING C)

Use the underscore calls to save the time used to check the return value. In C, you have two versions of a system call: the call name version and the underscore version. The underscore version is just the familiar call name with an underscore in front of it. For example **_crecv()** is the underscore version of **crecv()**.

The underscore version operates the same as the version described in the *iPSC®/2 and iPSC®/860 Programmers Reference Manual*, except that it doesn't check the return value. Rather, it provides the return value (`-1` if the call failed) and sets *errno* if the operation failed. You can check the return value yourself. This is useful if you want to handle an error yourself and not let the system do it. But if you are confident that your program is working, you may choose to use the underscore version and not check the return value, thereby improving performance.

MISCELLANEOUS

Avoid repeated use of system calls (applicable to CX and RX nodes running any language). This sounds like an obvious programming practice, but unfortunately it is missing from many applications. For example, a process may need its node and NX pid numbers to do message passing. Avoid using **mynode()** and **mypid()** each time you need those numbers. Instead, invoke each once and store their values in variables.

Turn off the LED (applicable to CX and RX nodes running any language). There is a tiny system overhead associated with running the green LED on the node board. This overhead can be eliminated by removing control of the green LED from the system. Use the **led()** call once at the beginning of your program.

THE CONCURRENT FILE SYSTEM 6

CONCURRENT FILE I/O

For high-speed simultaneous access to secondary storage, the nodes use the Concurrent File System™ (CFS). The files reside on a number of disks that connect to the cube through the SCSI interface on I/O nodes. Each disk has a unique volume number. The I/O nodes do not run application processes directly, but provide disk services for all users. All I/O nodes must have at least 4M bytes of memory.

A concurrent file is distributed over the disk drives (volumes) making up the Concurrent File System. Files are distributed in 4K-byte blocks in a round robin fashion in order by volume number.

The CFS also supports two kinds of tape drives: a 9-track tape drive, and an 8-mm cartridge tape drive. One 9-track tape reel has a capacity of 180M bytes while the 8-mm cartridge can hold up to 2.2G bytes.

The CFS is characterized by large storage (several disks, each with an unformatted 760M-byte capacity), large files (up to several gigabytes) distributed over several disk drives, and simultaneous access in one of four possible modes by different nodes.

The default CFS root directory is */cfs*. This name can be redefined by the host shell environment variable called CFS_MOUNT. If you want to redefine the name of the CFS root directory, you *must* define this environmental variable *before* you invoke *nsh*.

The environment variable DEV_MOUNT identifies the location of the backup device. By default the CFS device nodes are in */cfs*, and */dev* would identify a device on the SRM. If you set DEV_MOUNT to */dev*, then both */dev* and */cfs* refer to CFS tape drives.

CFS COMMANDS

iPSC system commands:	**cbackup** *(available on nodes only)* **crestore** *(available on nodes only)* **showvol**

The **showvol** command lists the volume, node, and drive numbers for each disk and tape drive making up the Concurrent File System (CFS). When describing a disk drive, the command shows the maximum size of each drive, how many bytes are still free for use, and the vendor, model, and revision number of each drive. Here is an example of **showvol** output.

```
% showvol

Volume Node Drive  Size        Free        Vendor   Model  Revision

  0    641  0 FD 649,359,360    648,548,352 MAXTOR  XT-8760S    B2
  1    641  2 FD 670,171,136    652,591,680 MAXTOR  XT-8760S    B5A
  2    641  1 FD 670,171,136    652,591,680 MAXTOR  XT-8760S    B3C
  3    641  3 FD 670,171,136    652,591,680 MAXTOR  XT-8760S    B5A
  4    642  0 RT            0 spdl den3 B P STK     2925        CS05
```

Each disk drive belonging to the CFS has a unique volume number. This is the number you need for use with the **restrictvol()** call described later. The node number is the number of the I/O node to which the drive is connected. This number is in the range from 641 to 767. It is equal to 640 plus the physical slot number of the compute node whose channel 7 is connected to the I/O node. The drive number is local to the I/O node.

Use the commands **cbackup** and **crestore** on the nodes to backup and restore the Concurrent File System.

NOTE

The **cbackup** and **crestore** commands are available on the nodes only. They must be executed from within the node shell (*nsh*). You should use the UNIX **backup** and **restore** commands to backup and restore files that are resident on the SRM.

cbackup and **crestore** are intended to backup and restore files onto a tape drive that is part of the CFS. You can use these commands to access the SRM tape drive. You still can backup SRM files or CFS files on the SRM's tape drive. However, the UNIX **backup and restore** are more convenient for SRM files, and the SRM tape drive usually does not have enough capacity for CFS files.

You m*ust* be in the node shell to use **cbackup** and **crestore**. If you are running the node shell on RX nodes, remember that you need at least four nodes.

By default, **cbackup** saves only those blocks that contain data. The **-a** switch, however, forces **cbackup** to save the entire disk image regardless of whether or not blocks contain data. If you want to back up only certain drives, list their volume numbers on the **cbackup** command line.

If you are backing up more than your device will hold, be sure to use the **-s** switch to specify the size of your backup device. For example, the following procedure saves a complete image from all the disk drives in your CFS on an Exabyte cartridge tape in a CFS tape drive. The device node is */cfs/tape*.

NOTE

If you are backing up your files on a tape drive in CFS (*/cfs/tape*), use an Exabyte 2G-byte tape cartridge.

1. Login as *root*.

2. Verify that no one has any cubes allocated:

```
% cubeinfo -s
CUBENAME        USER    SRM         HOST        TYPE        TTYS
iocube          root    srm_name    srm_name    0
```

3. Reserve the entire cube:

```
% getcube
getcube successful: cube type 8m8rxn0 allocated
```

4. Enter the node shell:

```
% nsh
```

5. Insert a tape cartridge in the SRM tape drive.

6. Perform the backup:

```
node% cbackup -s 1500M /cfs/tape
```

The -s switch specifies the amount of bytes to be stored on a cartridge. "M" is megabytes. The number in front of the magnitude specifier must be an integer. For example, **1.5G** would be illegal.

When the tape cartridge is full, the system prompts you to insert another tape.

```
Change media and press return
```

7. When the node shell prompt returns, exit the node shell and release the cube:

```
node% exit
% relcube
relcube released 1 cube
```

The **crestore** command restores previously backed up files. If you do not list volume numbers, it automatically restores all drives that were backed up. For example, assume that you issued that last **cbackup** command. Then, the following command restores all of drives 0, 1, and 3:

```
node% crestore /cfs/tape
```

If you wanted to restore only drive 1, issue the following command:

```
node% crestore 1 /cfs/tape
```

Listing a drive number that was not one of those backed up is an error.

CFS SYSTEM CALLS

Files are opened and closed with the familiar UNIX system calls and Fortran routines. For example, to open the file */cfs/mydata* for read and write access:

```
fd = open("/cfs/mydata", O_CREAT | O_RDWR, 0644);          Cversion

call open(unit=10, file = '/cfs/mydata', status = 'new',
x         form='unformatted')                              Fortran version
```

Note that the Fortran file must be opened as unformatted.

If you encode three or more # symbols into a filename, these symbols are replaced by the node number. For example, assume that you have the same node program running on several nodes. Each node opens a file called *file###*. The result is that each node opens a separate file. Node 0 opens *file000*; Node 1 opens *file001*; Node 2 opens *file002*, etc. Less than three # symbols in the filename appear as actual # symbols. For example, the file *file##1* is a true concurrent file accessible by each node.

Three or more # symbols only determine how the file is created. Once the file is created, it appears as a typical concurrent file. For example, assume that your node program creates file###, writes into it, and then closes the file. The result is that a number of concurrent files have been created. If your program now opens *file001*, it appears as a typical concurrent file. If more than one node opens *file001*, it is a shared file, accessible by all nodes.

A CFS file greater than 2G-1 bytes is called an extended file. These files are stored in the same way as non-extended files. The only difference is that some of the file parameters (like the file pointer and file size) do not fit into a 32-bit integer. Numbers greater than 2G-1 are called extended numbers, and you must use the **e...()** calls to operate on them. For files less than 2G-1 bytes, you can use either the extended calls or their familiar UNIX equivalents.

Concurrent File

System Call	Environment
eseek()	**node**
esize()	**node**
estat()	**node**
festat()	**node**
lsize()	**node**
restrictvol()	**node**

By default, the CFS makes use of all its volumes when distributing a file. If you want a particular file to be limited to one or more specific volumes, use **restrictvol()**.

restrictvol() controls allocation, not file access. If you restrict an existing file, you can still access the file blocks residing in volumes (disks) that aren't in your restricted set. When you add to the file, however, the new disk blocks come from the restricted set.

The calls **eseek()**, **estat()**, and **festat()** are for use with extended CFS files. You should also use the other **e...()** calls to perform extended arithmetic on extended file parameters. The calls **estat()** and **festat()** are only for use in C programs.

The only difference is that the **e...()** file calls deal with extended numbers. **estat()** and **festat()** use a structure called *estat*, which is the same as the UNIX *stat* structure except that the file size is of type **esize_t**. For information about **esize_t**, refer to the section "Performing Extended Arithmetic" in Chapter 3.

You can allocate more space to a file with **lsize()**. **esize()** is the extended version. The major use of these calls is to ensure that enough space is allocated before you do any sensitive calculations.

I/O Modes

System Call	Environment
iomode()	**node**
lseek()	**node**
setiomode()	**node**

A node program can access a concurrent file in one of four I/O modes. Use **setiomode()** to set a file I/O mode, and **iomode()** to determine a file's current mode.

The four possible I/O modes are:

Mode 0: Individual file pointer (IFP). This is the default mode.

Mode 1: Common file pointer (CFP)

Mode 2: Synchronized, CFP, variable length

Mode 3: Synchronized, CFP, fixed length

In mode 0, each node maintains its own file pointer. File access requests are honored on a first-come, first-served basis. If two nodes write to the same place in the file, the second node overwrites the data written by the first node.

The other three modes have a common file pointer. The consequence of a common file pointer is that after one node completes its operation, the next node begins in the file where the first node left off. Also, closing a file in modes 1, 2 and 3 is a synchronized operation. When one node closes a file, it waits until all the other nodes also close the file.

Mode 1 is an unsynchronized CFP mode. File accesses are done on a first-come, first-served basis. For example, mode 1 is useful for concurrent log files.

In mode 2, file accesses are done in order by node number. You can think of all the nodes operating in lock step. The synchronization is a time sychronization. Mode 2 has two consequences.

- All nodes must begin at the same position in the file. This means that the only valid use for **lseek()** is for all nodes to seek to the same position in a file prior to a read or write request. If nodes seek to different positions, an error is returned.
- All the nodes in your cube must open the file and perform the very same operation (a read or write) on the file. For example, if a node makes a second read request, it will wait until all the other nodes have completed their first reads. In addition to reads and writes, in mode 2, the calls **lseek()** and **iseof()** also perform a synchronization.

In mode 3, the nodes share a common file pointer, but they do not need to operate in lock step. Because the requests are of a fixed length, each node can determine where in the file it is reading from or writing to independently of the other nodes. Variable length requests provide more flexibility, but, because of the lock-step requirement, less performance than fixed-length requests. You can think of the synchronization in mode 3 as a spatial rather than a time synchronization.

The following example illustrates the differences between the various modes. Consider a program running on two nodes that opens a CFS file called *mydat*. Each node writes the current value of **mclock()** and the message "Hello from node x" to the file. If the program is running on node 0, it waits a certain amount of time before each write to the file. When a node has finished, it writes the current **mclock()** and a "Done" message to the screen.

Fortran version

```
      program mode3

      integer mode, nunit, iam, msglen, time, start_time, end_time
      character*(*) msg
      character*30 msgbuffer

      parameter (msg = 'Hello from node ')

      mode  = 3                                          The mode is 0, 1, 2, or 3.
      nunit = 12
     call open(unit=nunit, file = '/cfs/you/mydat',status = 'old',
     x         form = 'unformatted')

      call setiomode(nunit,mode)
      iam = mynode()

      msglen = len(msgbuffer)

      do 100 i=1,25

         start_time = mclock()
         if(iam .eq. 0) then
102         end_time = mclock()
            time = end_time - start_time
            if (time .ge. 500) then
               goto 101
            else
               goto 102
            endif
         endif
```

```
101       write(msgbuffer,11) start_time, msg, iam, char(10)
11        format(i9, a17, i3, a1)
          call cwrite(nunit,msgbuffer,msglen)
100    continue

       write(*,12) iam
12     format(' Done', I3)
       close(nunit)
       end
```

C version

```
#include <sys/types.h>
#include <fcntl.h>
#include <estat.h>
#include <cube.h>
main()
{
int           i,fd;
unsigned long time;
long          mode,iam;
char          msg[40];

   iam = mynode();
   mode = 3;
   fd = open("/cfs/you/mydat", O_RDWR, 0644);

   setiomode((long)fd,mode);

   for(i=0;i<25;i++) {
      if(iam==0) sleep(1);
      time = mclock();
      sprintf(msg,"%lu %s %ld\n",time,"Hello from node ",iam);
      cwrite(fd,msg,strlen(msg));
   }
   printf("Done %d\n",iam);
   close(fd);
}
```

The mode is 0, 1, 2, or 3.

The following example outputs came from the Fortran version of the example. Note that each node maintains its own value of **mclock()**. It does not make sense to compare values of **mclock()** from different nodes.

In mode 0, each node has its own file pointer. Node 1 finishes right away. Node 0 waits before each write and overwrites the message from node 1. The file shows only the writes from node 0.

Excerpt from file in mode 0

```
          •
          •
          •
502784361 Hello from node   0
502784886 Hello from node   0
502785395 Hello from node   0
502785904 Hello from node   0
          •
          •
          •
```

In mode 1, the nodes share a common file pointer, but there is no synchronization. As in mode 0, node 1 finishes right away; but this time, node 0 appends its data to the file rather than overwriting the data from node 1.

Excerpt from file in mode 1

```
          •
          •
          •
502782861 Hello from node   1
502782872 Hello from node   1
502782883 Hello from node   1
502782893 Hello from node   1
502784361 Hello from node   0
502784886 Hello from node   0
502785395 Hello from node   0
502785904 Hello from node   0
          •
          •
          •
```

In mode 2, the nodes share a common file pointer, and there is synchronization. Nodes 1 and 0 finish at around the same time. Because node 1 waits for node 0 after each write, the writes are interleaved within the file. Note that node 0 actually writes into the file before node 1, but its **mclock()** value is greater. This is because every node maintains its own clock.

Excerpt from file in mode 2

```
          •
          •
          •
501052969 Hello from node   0
501051233 Hello from node   1
501053500 Hello from node   0
501051756 Hello from node   1
501054016 Hello from node   0
501052278 Hello from node   1
          •
          •
          •
```

In mode 3, the nodes share a common file pointer, but there is no lock-step synchronization. Node 1 finishes right away. Then, node 0 goes into the file and fills in its data in the correct places. Because in this mode, the data are of a fixed length, node 0 has no trouble doing this. The result is that the data are in the same order as in mode 2.

Again, the absolute values of **mclock**() have no meaning. But notice that the time between the first two node 0 writes is 520 (greater than 500 as the program requires), but that the time between the first two node 1 writes is only 22.

Excerpt from file in mode 3

```
          •
          •
          •
501174992 Hello from node   0
501173255 Hello from node   1
501175512 Hello from node   0
501173277 Hello from node   1
501176016 Hello from node   0
501173286 Hello from node   1
          •
          •
          •
```

setiomode() is a global operation for processes with the same pid on all nodes. When you issue a **setiomode**(), it must be issued by all the nodes in the cube, even those that do not actually perform any I/O. For example, assume that your application has manager nodes that assign I/O work to the worker nodes. If you want to change the I/O mode, all the nodes, even the manager nodes, must issue the **setiomode**().

Note that a **setiomode**() always performs a synchronization. That is, it operates like a **gsync**() call. When a node performs a **setiomode**(), it waits until all other nodes also perform a **setiomode**(). This is why all nodes in a cube must issue the call.

Reading and Writing Files

System Call	Environment
cread()	**node**
cwrite()	**node**
iodone()	**node**
iowait()	**node**
iread()	**node**
iseof()	**node**
iwrite()	**node**

You can read and write files with the familiar UNIX system calls and Fortran routines. For example, here is a Fortran code fragment that opens a concurrent file whose path is */cfs/mydata* and reads some data into an array called *array* using the Fortran **read** routine:

```
call open(unit=10, file = '/cfs/mydata', form='unformatted')
read 10, (array(j), j=1, n)
```

For added performance in Fortran, use the CFS I/O calls whenever you can. With the CFS calls, you can do synchronous or asynchronous I/O. You can also test for the end of a file with **iseof()**.

The synchronous calls are **cread()** and **cwrite()**. For example, here is a Fortran code fragment that opens a concurrent file whose path is */cfs/mydata*, seeks to a location, and reads some data using the synchronous call **cread()**. The data represent a matrix stored in rows of *n* four-byte elements. Each node reads *m* rows and performs a calculation with each row. Because each node seeks to a different place in the file, you can't use modes 2 or 3. Mode 0 has the highest performance, and it is most appropriate for this case.

```
      .
      .
      .
      call open(unit=10, file='/cfs/mydata', form='unformatted')
      lseek(10, 4*mynode()*n*m, 0)

      do 10 i = 1, m
         call cread(10, arow, n*4)
         y(i) = sdot(n, arow, 1, xtotal, 1)      VecLib routine to calculate
                                                    the dot product
10    continue
         .
         .
         .
```

Note that when Fortran opens a file for access by the concurrent I/O calls, the file must be opened as sequential and unformatted. Sequential is the default access, but you must specify the file as unformatted. Also, your program cannot mix Fortran file I/O routines with CFS system calls on the same file.

The last example showed the use of a previously created file. If you want to create a file, one node must create it while all the others just open it. You have to be careful, though, that the nodes doing the open don't try to open the file until it has been created. This is a good application for the **gsync()** call, one of the global operations. Here is a C code fragment in which node 0 creates a file called *my_file* and the other nodes open it:

```
long fd;
   •
   •
   •
if (mynode() == 0) {
   fd = open("my_file", O_RDWR | O_CREAT, 0644);
   gsync(); /* let other nodes open file */
}
else {
   gsync(); /* wait for node 0 to create file */
   fd = open("my_file", O_RDWR);
}
setiomode(fd, 1);
   •
   •
   •
```

You must also take synchronization into account when you **unlink()** the file. For example, assume that after the **if** statement, all nodes do some processing and then node 0 executes an **unlink()**. If you have a large cube, it is possible that node 0 reaches the **unlink()** before some of the other nodes complete the **open()**. These nodes would then return an error -- they would be trying to open a nonexistent file.

The **setiomode()** would prevent this problem from occurring in the example just shown. The **setiomode()** causes a synchronization. However, if you wanted the default mode of 0, you may not have issued the **setiomode()**. **close()** is also a synchronized call; so if you close your file before a node unlinks it, no problem develops. The point is that you must synchronize before an **unlink()**; and if this synchronization is not achieved by either a **setiomode()**, a **close()**, or some other call, you can issue a **gsync()** before the **unlink()**.

If you are working in the C language, you also have available the underscore synchronous read and write calls, **_cread()** and **_cwrite()**. These are equivalent to the UNIX-compatible **read()** and **write()** calls, but they are included for symmetry with the Fortran calls.

The asynchronous calls are **iread()** and **iwrite()**. They return an I/O id much like the message id returned by the asynchronous message passing calls.

To check if an asynchronous I/O operation has completed, use the **iodone()** call. It returns a 1 when the asynchronous operation has completed and a 0 otherwise. You can also decide to block on the completion of an asynchronous call. Use the **iowait()** call for this. Both **iodone()** and **iowait()** take the I/O id as an input parameter.

Be sure to release ids that are no longer needed. There are two ways to release an I/O id. You can issue an **iowait()**, or you can keep issuing **iodone()**s until an **iodone()** returns a 1.

USING THE CFS TO STORE NODE PROGRAMS

Consider stored node executables in the Concurrent File System. This is useful for two reasons. Not only is loading from the CFS faster, but it also saves space in your host file system. You store node executables in the CFS the same way you would store them in the host file system. Just be sure to specify the complete pathname. For example, to load the file *nodeprog* in */cfs/bin*, issue the command,

```
% load /cfs/bin/nodeprog
```

or use the **load** system call,

```
load("/cfs/bin/nodeprog")
```

USING THE NODE SHELL

You can load the node shell on node 0. When you issue a node shell command, however, processes may run on other nodes. To load the node shell, use one of the following command:

```
% nsh                    (to load nsh on node 0 of subcube)
% nsh -s                 (to load nsh on service node)
```

To change the node shell prompt from its default of `%` to `node%`, add the following line to *.login.ipsc* in your home directory on the host file system:

```
set prompt="node% "
```

The node shell uses the files *.login.ipsc*, *.cshrc.ipsc*, and *.logout.ipsc* in the same way that the host uses *.login*, *.cshrc*, and *.logout*.

The node shell has access to both the host file system and the Concurrent File System. For example, you can copy a file from the host file system to the CFS as follows:

```
node% cp ~username/file1 /cfs/file2
```

The sequential file *file1* becomes the concurrent file *file2*.

You can also use the **star** (streaming **tar**) command. For example, to copy the file *file2* from the CFS to a tape in the host file system, issue the following command:

```
node% cd /cfs
node% star c /dev/tape file2
```

To return to the node shell, use the following command:

```
node% exit
%
```

Appendix A lists the **csh** built-in commands, the UNIX utilities, and the iPSC system NX commands supported by the node shell. The built-in commands and the UNIX utilities operate the same as those running on the host. The node shell can run any node program intended for use on a single node.

The **killcube** and **waitcube** commands running on the node do not accept the **-c** switch and require the **-p** switch. That is, these commands cannot communicate with other allocated cubes. Specifying a pid prevents you from affecting the node shell itself. For Intel386-based nodes, the node shell has its own NX pid which is one of the NX reserved pids (2 000 000 000 and above). For RX nodes, the node shell has pid 0, as do all i860 processes.

You can also run individual UNIX commands on the node without invoking the node shell. To run an individual UNIX command on a node, use the **load** command and specify the full path name of the node shell version of the command (*/usr/ipsc/bin*). For example, to have node 4 obtain a directory listing of */cfs*, the root file of the Concurrent File System, issue the following command:

```
% load 4 /usr/ipsc/bin/ls -l /cfs
```

Because RX nodes run only one process, you must allocate a separate cube to run the node shell. You should allocate at least four nodes. This cube should be the current cube when you execute **nsh -s** (or run on service nodes).

The **nsh** commands reside in the directory */usr/ipsc/bin* and */usr/i860/ipsc/bin* on the SRM. To use the node shell from a remote host, copy all the **nsh** commands to the CFS. Perform the following steps.

1. Login to the SRM.

2. For a CX cube, execute the following commands:

```
% getcube
% nsh -s
node% cd /usr/ipsc/bin
node% mkdir /cfs/bin
node% mkdir /cfs/bin/cx
node% cp /usr/ipsc/bin/[a-j]* /cfs/bin/cx
node% cp /usr/ipsc/bin/[k-z]* /cfs/bin/cx
node% exit
node% relcube
```

For an RX cube, execute the following commands:

```
% getcube
% nsh -s
node% cd /usr/i860/ipsc/bin
node% mkdir /cfs/bin
node% mkdir /cfs/bin/rx
node% cp /usr/ipsc/i860/bin/[a-j]* /cfs/bin/rx
node% cp /usr/i860/ipsc/bin/[k-z]* /cfs/bin/rx
node% exit
node% relcube
```

For a mixed cube, execute the following commands:

```
% getcube
% nsh -s
node% cd /usr/ipsc/bin
node% mkdir /cfs/bin
node% mkdir /cfs/bin/cx
node% cp /usr/ipsc/bin/[a-j]* /cfs/bin/cx
node% cp /usr/ipsc/bin/[k-z]* /cfs/bin/cx
node% cd /usr/i860/ipsc/bin
node% mkdir /cfs/bin/rx
node% cp /usr/ipsc/i860/bin/[a-j]* /cfs/bin/rx
node% cp /usr/i860/ipsc/bin/[k-z]* /cfs/bin/rx
node% exit
node% relcube
```

3. Add the following lines to the *.cshrc.ipsc* file in your home directory on the remote host:

   For a CX cube, execute the following commands:

   ```
   set path=(/cfs/bin/cx .)
   set shell=/cfs/bin/cx/csh
   ```

   For an RX cube, execute the following commands:

   ```
   set path=(/cfs/bin/rx .)
   set shell=/cfs/bin/rx/csh
   ```

You may find an improvement in performance if you copy the **nsh** commands to the CFS even if you intend to work on the SRM

REMOTE HOST ACCESS TO NSH

To use the node shell (**nsh**) from a remote host workstation:

1. Login to the SRM.

2. Perform the following sequence to copy the commands to CFS:

   ```
   getcube
   nsh
   cd /usr/ipsc/bin
   mkdir /cfs/bin
   cp /usr/ipsc/bin/[a-j]* /cfs/bin
   cp /usr/ipsc/bin/[k-z]* /cfs/bin
   exit
   relcube
   ```

Once you have copied the commands, you must add the following to the *.cshrc.ipsc* file in your home directory on the remote host:

```
set path=(/cfs/bin)
set shell=/cfs/bin/csh
```

CFS ROUTINE SYNCHRONIZATION

Some of the library routines used to access the CFS occasionally perform some synchronization (message passing). The node processes participating in the synchronization must have the same pid, and all nodes in the cube must participate. Table 6-1 lists the routines and shows when (or if) they do such synchronization.

Table 6-1. CFS Routines That Synchronize

CFS Routine	Conditions Causing the Routine to Synchronize
close()	Mode 1, 2, or 3
setiomode()	Always
lseek()	Mode 2
iseof()	Mode 2
cread()	Mode 2
cwrite()	Mode 2
iread()	Mode 2
iwrite()	Mode 2

Mode 3 is a synchronized mode. Routines **lseek()**, **iseof()**, **cread()**, **cwrite()**, **iread()**, and **iwrite()** do not perform any synchronization in Mode 3. The high performance provided by Mode 3 is attributable to this fact.

TCP/IP ON THE NODES 7

INTRODUCTION

TCP/IP is named after two of the Internet's main standards, the Transmission Control Protocol and the Internet Protocol.

Intel provides an implementation of the TCP/IP protocol running on iPSC system RX nodes. Node programs can use this software to send messages to other computers on the Ethernet running TCP/IP software. These other computers can be remote hosts, the SRM, or any other workstation on the Ethernet. The interface is intended for C programs.

The TCP/IP software runs on a PBX I/O node, also referred to as a socket node. A PBX I/O node is an I/O node without a SCSI interface. It is specially designed to operate with a Bus Interface Adapter (BIA) board. (Refer to the *iPSC®/2 and iPSC®/860 VME Interface Reference Manual.*) The purpose of the socket node is to let RX node programs access the Ethernet.

An iPSC/860 system can have several socket nodes. When the NX operating system loads a program on an RX node, it assigns a socket node to that program. That assigned socket node is called the "iphost." The NX operating system keeps track of what socket nodes it has assigned and assigns one that it has given out the least number of times.

When a program running on an RX node makes a TCP/IP call, the NX operating system sends a request to the TCP/IP software running on the iphost. The I/O node in turn communicates with an iPSC system BIA that is connected to an Ethernet controller.

This chapter describes how to use TCP/IP with an iPSC system. It assumes that you are already familiar with TCP/IP programming.

PERFORMANCE CONSIDERATIONS

Although you can use TCP/IP calls to send messages between nodes, you should use the NX message passing calls for high performance. Unlike the message passing calls, the TCP/IP calls must be routed through a PBX I/O node.

The TCP/IP mechanism is designed primarily for use between an RX node and another computer on the Ethernet, such as a remote host or the SRM.

THE IPHOST

The */etc/hosts* file on the SRM lists the iphost names and Internet addresses of the socket nodes. This information was added to the */etc/hosts* file during installation with the **ipconf** command. Each socket node has its iphost name and its own Internet address, which you assigned during installation. Any program on the Ethernet can connect to this socket node.

If a program is to use TCP/IP to communicate with a node program, this program must know the node program's iphost name. If there is more than one socket node in the system, one solution is for the node server to set a particular iphost name and override the one that was dynamically determined.

A node program can find out what iphosts are available by issuing the call **getiphosts()**. This call returns a pointer to an array of character pointers. The list is similar to **argv** and is null terminated. The node program can then look at the list and set a particular PBX I/O node to be the iphost with **setiphost()**. It can choose the iphost expected by the connector.

For example, the program in Figure 7-1 calls **getiphosts()** and puts the list of iphosts in *iphlist*. It then checks each value and makes a particular one the iphost.

The example presented later in this chapter has the node program accept the name of its iphost as a command line argument, provided when you loaded it with the **load** command. For example, assume that you load the program *node* as

```
load node iphostname
```

Then, the node program can attach to *iphostname* by executing the call

```
setiphost(argv[1]);
```

```
#include <i860/include-ipsc/CMC/sys/types.h>
#include <i860/include-ipsc/CMC/sys/socket.h>
#include <i860/include-ipsc/CMC/netinet/in.h>
#include <i860/include-ipsc/CMC/netdb.h>
#include <i860/include-ipsc/CMC/ntoh.h>
#include <cube.h>
#define NULL 0
extern char **getiphosts();

long iam;

main(argc,argv)
int argc;
char **argv;
{
char           **iphlist;
struct hostent *hp;
int            i;

   iam = mynode();
   if(argc != 2) {
      if(!iam)
         printf("Usage: %s <iphostname>\n",argv[0]);
      exit(1);
   }


   iphlist = getiphosts();
   i       = 0;
   while(iphlist[i] != NULL) {
      if(!strcmp(iphlist[i],argv[1])) {
         setiphost(iphlist[i]);
         hp = gethostbyname(iphlist[i]);
         break;
      }
      i++;
   }

   if(iphlist[i] != NULL) {
      if(!iam) printf("%d: host is %s\n",iam,hp->h_name);
   }
   else {
      if(!iam) printf("Unable to find desired host:
%s\n",argv[1]);
   }
}
```

Figure 7-1. Example Using getiphosts()

NODE TCP/IP SPECIFICS

This section describes the version of TCP/IP that is supported on the nodes.

Addressing Scheme and Socket Type

The TCP/IP that is available to node programs follows the AF_INET addressing scheme. This addressing scheme uses Internet addresses to identify the machine. Port numbers allow more than one socket on a machine. Also, the node TCP/IP uses only the SOCK_STREAM type of sockets.

The consequence is that when you create a socket with the **socket()** call, you must specify AF_INET and SOCK_STREAM. Also, be sure to use the Internet socket structure, `struct sockaddr_in`. (defined in *$(IPSC-XDEV)/i860/include-ipsc/CMC/netinet/in.h*) .

Finally, by default, sockets created for use with node TCP/IP are blocking sockets. When you issue an **accept()** call, your program blocks until someone makes a connection. You can make sockets nonblocking with **fcntl()**.

Include Files

Include files reside by default on the SRM under *$(IPSC-XDEV)/i860/include-ipsc/CMC*. The directory structure under *CMC* is as you would expect for TCP/IP. For example, if your makefile specifies the **-I** option as ***-I$(IPSC-XDEV)/i860/include-ipsc***, then to include *socket.h*, your program should contain the following line:

```
#include <CMC/sys/socket.h>
```

There is one exception, however. When you use the byte order calls (such as **htons()**), you must also include *$(IPSC-XDEV)/i860/include-ipsc/CMC/ntoh.h*. That is:

- Under the UNIX operating system, the byte order calls require only the */usr/include/sys/types.h* and */usr/include/sys/in.h* header files.
- Under the NX operating system, these calls also require *$(IPSC-XDEV)/i860/include-ipsc/CMC/ntoh.h* (in addition to */usr/include/sys/types.h* and */usr/include/sys/in.h*).

Server/Client Relationship

TCP/IP convention is to have the server do the **accept()**. The server doesn't have previous knowledge of its caller. It gets that information from the **accept()**. A client on the other hand must know from whom to request service. It must put that information in the **connect()** call.

A node can be a server or a client. However, in an iPSC system, the nodes are usually considered to be a compute server, and the host is the client. When you use TCP/IP to send data between a node and the host, the node is usually the acceptor and the host is the connector.

TCP/IP convention is to have the server **fork()** a child to handle a client request after a successful **accept()**. If you do this in an RX node, the NX operating system loads the child on another node in the cube. For this to be successful, you must have at least one node in your allocated cube that does not have a process running on it. You also might consider writing your application so that it does not **fork()** processes to handle requests.

If a name server is configured for the SRM, the node TCP/IP will use it. The calls **sethostent()** and **endhostent()** are available to open and close the */etc/hosts* file, and **gethostent()** is provided to access the file.

Also, when you use **gethostbyname()**, note that the argument is the name of the iphost, not the name of the SRM.

COMPILING A NODE PROGRAM WITH TCP/IP

The TCP/IP calls are in *libsocknode.a*. Include the switch **-lsocknode** on the linking step of your compile. For example, to compile the program *node.c* for use on an RX node with the TCP/IP libraries, issue one of the following commands on the SRM:

```
icc -o node node.c -lsocknode -node
```

If you use a makefile, the pertinent lines are as follows:

```
host: host.c
      cc -o host host.c -host
node: node.o
      icc -o node node.o -lsocknode -node
```

Note that you may want to put **-node** as part of SWITCHES. The built-in rule for generating a C object from a C source uses SWITCHES. Putting **-node** into SWITCHES would include **-node** on the compilation step. This results in the definition of the preprocessor symbol _ _NODE, which your application may find useful.

Also note that the makefile does not use a built-in rule for compiling *host.c*.

The Node TCP/IP System Calls

The node TCP/IP software provides a number of system calls for node programs. These calls include the standard TCP/IP calls, documented in the *Ethernet Node Processor ENP-100 Series Reference Guide*. For the NX operating system, node TCP/IP calls are all library calls.The calls **bcmp(3)**, **bcopy(3)**, **bzero(3)**, and **index(3)** require that you include *bsd.h*. **gettimeofday(3)** requires that you include *longnames.h*.

Table 7-1 lists the calls provided by the node TCP/IP.

Table 7-1. Node TCP/IP System Calls (1 of 3)

TCP/IP System Call	Description
accept(*s*, *addr*, *addrlen*)	Accept a connection on a socket.
bind(*s*, *name*, *namelen*)	Bind a name to a socket. The name is a `struct sockaddr_in`.
connect(*s*, *name*, *namelen*)	Initiate a connection on a socket.
gethostbyaddr(*addr*, *len*, *type*) **gethostbyname**(*name*) **sethostent**(*stayopen*) **endhostent**() **gethostent**()	Get network host entry information. The name in **gethostbyname**() is a character string.
gethostname(*name*, *namelen*)	Get name of current host. The name is a character string.
getnetbyaddr(*net*, *type*) **getnetbyname**(*name*) **getnetent**() **setnetent**(*stayopen*) **endnetent**()	Get network entry information.
getpeername(*s*, *name*, *namelen*)	Get the name of the connected peer. The name is a `struct sockaddr_in`.
getprotobyname(*name*) **getprotobynumber**(*proto*) **getprotoent**() **setprotoent**(*stayopen*) **endprotoent**()	Get protocol entry information. The name is a character string.

Table 7-1. Node TCP/IP System Calls (2 of 3)

TCP/IP System Call	Description
getservbyname(*name, proto*) **getservbyport**(*port, proto*) **getservent**() **setservent**(*stayopen*) **endservent**()	Get service entry information. The name is a character string.
getsockname(*s, name, namelen*)	Get the name of the specified socket. The name is a `struct sockaddr_in`.
getsockopt(*s, level, optname, optval, optlen*)	Get socket options.
setsockopt(*s, level, optname, optval, optlen*)	Set socket options
htonl(*hostlong*) **htons**(*hostshort*) **ntohl**(*netlong*) **ntohs**(*netshort*)	Byte swapping between host and network and vice versa.
inet_addr(*cp*) **inet_lnaof**(*in*) **inet_makeaddr**(*net, lna*) **inet_netof**(*in*) **inet_network**(*cp*)	Internet address manipulation routines.
listen(*s, backlog*)	Listen for connections on a socket.
recv(*s, buf, len, flags*) **recvfrom**(*s, buf, len, flags, from, fromlen*)	Receive messages from a socket.
select(*nfds, readfds, writefds, exceptfds, timeout*)	Synchronous I/O multiplexing.
send(*s, msg, len, flags*) **sendto**(*s, msg, len, flags, to, tolen*)	Send a message from a socket.
shutdown(*s, how*)	Shut down part of a full-duplex connection.
socket(*domain, type, protocol*)	Create a socket.

Table 7-1. Node TCP/IP System Calls (3 of 3)

Unique Node TCP/IP System Calls	
getiphosts()	Obtain a list of the names of available socket nodes.
setiphost(*iphost***)**	Specify a particular socket node as the iphost. The system dynamically chooses a default iphost based on load balancing. This call gives you the ability to override the system's decision.

How to Create a Socket, Connect to It, and Transfer Data

System Call	Environment
accept()	**node**
bind()	**node**
connect()	**node**
gethostbyaddr()	**node**
gethostbyname()	**node**
gethostname()	**node**
getiphosts()	**node**
getpeername()	**node**
htonl()	**node**
htons()	**node**
listen()	**node**
ntohl()	**node**
ntohs()	**node**
setiphost()	**node**

Consider the C version of the pi example included with your iPSC system software. This section shows how to convert that example into one using TCP/IP calls.

The source code for the original example is in */usr/ipsc/examples/c/pi* on the SRM. The host program sends two messages to all nodes. The first message is the number of participating nodes; the second message is a structure containing the integration limits and the number of points to use in the quadrature. This example does not use any TCP/IP calls.

If you use TCP/IP, it makes best sense to communicate between the host and one node (typically node 0) and then have that node broadcast to the other nodes. The host program must know the Internet name or address of the iphost and share a port number with its communicating partner, also called its peer. It is a good idea to choose a large number (greater than 5000) for the port number so that you do not conflict with any reserved ports. The node program creates the socket and blocks on an **accept()**.

AN EXAMPLE OF A NODE SERVER (pi Example)

The example is the familiar pi example described in Chapter 4. However, in this case, communication between the host and node 0 is done with TCP/IP calls. Then, node 0 broadcasts any needed information to the other nodes, using **csend()** for higher performance.

The actual calculation takes place in the node program, which is loaded with the **load** command. Node 0 blocks at an **accept()** and waits for input from a client host. A host program uses the node server to calculate pi. When the host program completes, the node server is ready for another host client.

The node program on node 0 has one command line argument, the name of an iphost. Because in this example, there is more than one socket node, the program's first task is to make this the iphost with the **setiphost()** call. Then, the node program on node 0 creates the socket *s* with **socket()**, gets its iphost information with **gethostbyname()**, binds that information to the socket with **bind()**, listens with **listen()**, and then blocks with **accept()**.

The example here arbitrarily picks a port number of 5500. Because this number goes out on the net, **htons()** must be used to get the correct byte order. The number 5500 is arbitrary, but it must be the same number used by the corresponding host program. **ntohs()** performs the opposite translation; **htonl()** and **ntohl()** operate on long integers.

Then, node 0 reads the appropriate information from the socket. It uses message passing calls to relay that information to the other participating nodes. All the nodes perform the calculation, which concludes with a **gdsum()** that puts the answer in the variable *partial_int*. Node 0 also calculates an elapsed time and writes the answer and the time to the socket. The infinite loop puts node 0 back at the **read()**.

Note that only node 0 makes the TCP/IP calls. The node program has two nested infinite loops. The outer loop accepts the connection; the inner loop performs the calculation. When the client host no longer requires service, the node program exits the inner loop. Node 0 blocks at the **accept()** in the outer loop, and the other nodes block at **crecv()**'s in the inner loop.

The node program need only accept the socket once before entering its inner infinite loop. A program's **read()** will block until its peer performs a **write()**. If it closed the socket after each cycle, the host program would have to reconnect. Note also that the socket is full duplex; the node program can both read and write to it.

In the code fragments in Figure 7-2, the TCP/IP calls are shown in **bold** typestyle. Only the pertinent lines in the node program are shown.

```
#include <$(IPSC-XDEV)/i860/include-ipsc/CMC/sys/types.h>
#include <$(IPSC-XDEV)/i860/include-ipsc/CMC/sys/socket.h>
#include <$(IPSC-XDEV)/i860/include-ipsc/CMC/netinet/in.h>
#include <$(IPSC-XDEV)/i860/include-ipsc/CMC/netdb.h>
#include <$(IPSC-XDEV)/i860/include-ipsc/CMC/ntoh.h>
•
•
•
char                myname[80];
int                 iam, s, len, news;
struct sockaddr_in  acceptor;
struct hostent      *hp;

iam = mynode();
if(iam == 0) {
    setiphost(argv[1]);
    printf("Iphost is %s\n",argv[1]);
    len = sizeof(acceptor);
    s = socket(AF_INET, SOCK_STREAM, 0);

    hp = gethostbyname(argv[1]);
    if(hp == NULL) { printf("no host\n"); exit(1);}

    memset(&acceptor,0,sizeof(acceptor));
    memcpy(&acceptor.sin_addr, hp->h_addr, hp->h_length);
    acceptor.sin_family = AF_INET;
    acceptor.sin_port = htons((short)5500);

    if(bind(s, &acceptor, len) < 0) {
       perror("bind");
       exit(1);
    }
    if (listen(s, 3) < 0) {
       perror("listen");
       exit(1);
    }
}
•
•
•
```

Figure 7-2. Example Node Server (1 of 2)

```
.
.
.
/* Begin infinite loop */
for(;;) {
   if(iam == 0) news = accept(s, &acceptor, &len);
   for(;;) {
      partial_int = 0.0;
      if(iam == 0) {
         if(read(news, &work_nodes, sizeof(work_nodes)) <= 0)
            break;
         read(news, &integral, sizeof(integral));
        /* Now use message passing to get the information
            to the other nodes. */
         csend(SIZE_TYPE,&work_nodes,sizeof(work_nodes),-1,0);
         csend(INIT_TYPE,&integral,sizeof(integral),-1,0);
      }
      else {
         crecv(SIZE_TYPE,&work_nodes,sizeof(work_nodes));
         crecv(INIT_TYPE,&integral,sizeof(integral));
      }
.
.
.
/* All nodes perform the calculation. Use a global operation to
collect the answer in partial_int. */

     gdsum(&partial_int,1,&work);


      if(iam == 0) {
         integral.a = partial_int;
         integral.points = mclock() - starttime;
         write(news, &integral, sizeof(integral));
      }
   } /*End inner infinite loop */
   close(news);
} /* End outer infinite loop */
.
.
.
```

Figure 7-2. Example Node Server (2 of 2)

Here are a few more observations about the node program.

The **memset()** call sets the socket address structure called *acceptor* to zero before it is used. Even though you may be overwriting the fields in *acceptor* later, it is essential that you zero the structure first.

The node program gets information about its host with **gethostbyname()**. **gethostname()** returns the character string representing the name of the iphost. This is the same name you chose when you installed TCP/IP on the nodes. **gethostbyname()** uses that name to get more detailed information about the iphost, which it puts in the structure *hp*. That information is written into the acceptor's socket structure called *acceptor*. It is that structure that is bound to the socket.

If you knew the Internet address of the iphost, you might have chosen to use **gethostbyaddr()** instead of **gethostbyname()**.

Note that the same socket structure (*acceptor*) is used in both the **bind()** and the **accept()**. Once the information has been bound with **bind()**, you don't need the data any more. The **accept()** fills the structure with the information about the connector. You could also get information about the connector with **getpeername()**.

AN EXAMPLE OF A HOST CLIENT (pi Example)

The host program gets information about its accepting iphost, puts that information in a socket address structure, and creates a socket. This example assumes that the name of the iphost is *iphostname*. You decided what the name was when you installed TCP/IP on the nodes.

The host program then attempts to connect to a socket described by the socket address structure it just filled. If the connection is successful, the host program asks the user for the number of participating nodes and the number of points in the quadrature. It then writes this information to the socket. Finally, it reads the socket, expecting an answer.

In the code fragments in Figure 7-3, the TCP/IP calls are shown in **bold** typestyle. Only the pertinent lines in the node program are shown.

```
#include <$(IPSC-XDEV)/i860/include-ipsc/CMC/sys/types.h>
#include <$(IPSC-XDEV)/i860/include-ipsc/CMC/sys/socket.h>
#include <$(IPSC-XDEV)/i860/include-ipsc/CMC/netinet/in.h>
#include <$(IPSC-XDEV)/i860/include-ipsc/CMC/netdb.h>
#include <$(IPSC-XDEV)/i860/include-ipsc/CMC/ntoh.h>
•
•
•
int                 s,len;
struct sockaddr_in  connector;
struct hostent      *hp;

len = sizeof(connector);
setpid(HOST_PID);
hp = gethostbyname(argv[1]);
if(!hp) exit(1);
memset(&connector, 0, sizeof(connector));
memcpy(&connector.sin_addr, hp->h_addr, hp->h_length);
connector.sin_family = AF_INET;
connector.sin_port = htons(5500);

s = socket(AF_INET, SOCK_STREAM, 0);
if(connect(s, &connector, sizeof(connector)) < 0) {
   close(s);
   perror("connect");
   exit(1);
}

for(;;) {  /* Infinite loop */
            /* Get user input. */
    if (!user_input(&msg, &size)) break;
      write(s, &size, sizeof(size));
      write(s, &msg, sizeof(msg));
      read(s, &msg, sizeof(msg));
      integral = msg.a;

/* Calculate a time interval. Display the answer and time. */
•
•
•
}   /* End infinite loop */
close(s);
}
```

Figure 7-3. Example Host Client

Again, notice that the connector's socket address structure is set to zero before use. This is essential. As before, the connection is full duplex. The host program both writes and reads the socket.

Note that the connector needed to know both its acceptor's name and the port number to be used. The port number can be agreed upon before writing the program. If you have only one iphost, you can do the same with the acceptor's name.

The example can be run as follows:

- Load the node program with the **load** command, assuming *iphostname* is the name of a socket node.

  ```
  load node iphostname
  ```

  The node server is now ready for service.

- Execute the host program, also specifying the iphost to connect to.

  ```
  host iphostname
  ```

  The node program runs and calculates pi. When you exit from the program, the host program is finished, but the node server is ready for use again.

- Use the **killcube** command to kill the node server.

THE X WINDOW SYSTEM ON THE iPSC®/860 8

INTRODUCTION

A set of X Window System client libraries is offered as an option for the iPSC/860 system. These libraries run only on i860-microprocessor-based RX nodes. The X Window System is a software standard developed during Project Athena at the Massachusetts Institute of Technology. It can be used to control the display of workstations, and can provide a standard environment for application software. This option runs on the nodes of the iPSC/860 system, and is Version 11 Release 4.0 (commonly abbreviated as X11R4) of the X Window System.

This chapter does not attempt to describe how to write X Window System applications programs. It describes only information specific to those who are using the iPSC system to write X applications.

Your workstation must have the X server software to be able to use the client libraries. Most versions of the X server software have an authorization mechanism to limit access of clients on other nodes of the network to your display. Refer to the X server documentation for your workstation for more information about authorization and security. Note that the name or Internet address for which you want to give access is the name or Internet address of the iphost (the PBX I/O node), not the SRM.

The node TCP/IP software must also be installed on your iPSC system before you can install and use the X Window software. Ensure that when TCP/IP was set up on your system that an entry was included in the SRM's */etc/hosts* file for the workstation containing your X server software. If not, you need to have the system administrator add an entry for your workstation (refer to the *iPSC®/2 and iPSC®/860 System Administrator's Guide*).

This chapter describes how to compile and link X applications. It also lists the supplied client libraries and describes how to find X resources. X Window installation is described in the *iPSC®/860 X Window Product Release Notes.*

COMPILING AND LINKING X WINDOW SYSTEM APPLICATIONS

Applications using the X Window System must be written in the C language. To compile an X Window System application, you must use the **icc** compiler driver. For example:

icc -node *filename*

Other compiler switches may be used as well. The required switch (**-node**) compiles the code to be executed on iPSC nodes (rather than on the simulator).

The X Window System has recently undergone a transition from Release 3.0 to Release 4.0. During this transition, major changes were made to the *Xaw* library. To assist in transition, the Release 4.0 *Xaw* library in this release includes backward compatability for applications written with the X Window Systems Release 3.0 *Xaw* library. To make use of this compatibility, programs must be compiled with the **-DXAW_BC** flag added to the compile line. Applications should, however, be converted to Release 4.0, because it cannot be guaranteed that backward compatibility will be a feature of future versions of this software.

To link X Window Systems applications, you must link in the client libraries of your choice with one or more of the switches shown in Tables 8-1 and 8-2. You must also include the **-lsocknode** switch to link in the necessary TCP/IP library,

There are nine X window client libraries provided. Most of these libraries (*Xlib*, *Xaw*, *Xmu*, *Xt*, and *oldX*) are documented in the X Window System manuals by O'Reilly and Associates, provided with this release. Volumes in which these libraries are documented are listed in Table 8-1.

Table 8-1. X Window Libraries

Library Name	Function	Link Switch
Xlib	core X Window System library, documented (Volumes 1 and 2 of the O'Reilly manuals)	**-lx11**
Xaw	Athena widget set, documented (Volumes 4 and 5 of the O'Reilly manuals)	**-lXaw**
Xmu	MIT miscellaneous utilities. (Volume 2 of the O'Reilly manuals)	**-lXmu**
Xt	toolkit intrinsics layer. (Volumes 4 and 5 of the O'Reilly manuals)	**-lXT**
oldX	X10 compatability library. (Volume 2 of the O'Reilly manuals)	**-loldX**

Xlib is the only library whose name on the system is different from the standard library name.

The other supplied libraries, *Xau*, *Xdmcp*, *Xext*, and *Xinput*, are typically used for advanced X Window System programming. Documentation for these libraries is supplied in troff format and is located (on an iPSC system on which the X Window System libraries are installed) in the */usr/ipsc/lib/X11/doc* directory. Specific file names for these documents are shown in Table 8-2.

Table 8-2. X Window Libraries for Advanced Applications

Library Name	Function	Link Switch
Xau	X11R4 sample Authorization Protocol (troff documentation in */usr/ipsc/lib/X11/doc/Xau)*	**-lXau**
Xdmcp	Xdm protocol library. (troff documentation in */usr/ipsc/lib/X11/doc/Xdmcp)*	**-lXdmcp**
Xext	support for the MIT shape, multibuffering, and MIT miscellaneous extensions. (troff documentation in */usr/ipsc/lib/X11/doc/Xest)*	**-Xext**
Xinput	Input Extension Library. (troff documentation in */usr/ipsc/lib/X11/doc/Xinput)*	**-lXinput**

RESOURCES

The preferred method of defining resources is to use the command **xrdb** to download resources to the X server. This method is documented in the X server documentation for your workstation. When resources are not downloaded, the X Window System will look for resource information in several places in the file system of the computer that loaded the node program.

The primary repository for application specific resource files is the following directory:

/usr/ipsc/lib/X11/app-defaults

The resource files in this directory are always used. These files normally contain the default set of resources provided by the application author. To establish an alternate directory as the default resource directory, set the environment variable **XUSERFILESSEARCHPATH** to the full pathname of the desired directory.

To establish a resource directory in addition to the default directory, set the environment variable, **XAPPLRESDIR** to the full pathname of the additional resource directory. Resources specified by **XAPPLRESDIR** take precedence over resources in the default directory (either */usr/ipsc/lib/X11/app-defaults* or the directory defined in the **XUSERFILESSEARCHPATH** statement).

User specific resource definitions should be located in the file:

$HOME/**.Xdefaults-***hostname*

where $HOME indicates the home directory of the workstation you are logged into when you are using the iPSC system. The *hostname* is the name of the iPSC/860 system on which you are running the X window application. This path of this file can be overridden by defining the environment variable **XENVIRONMENT**.

NODE CONNECTION TO THE SERVER

Only the node that calls the procedure **XOpenDisplay** has a connection to the server. As a result, you need either to ensure that one node makes all of the X Window System calls, or that each node opens its own connection to the server.

Each call to **XOpenDisplay** uses one file descriptor in the server. Most X Window System servers reside on UNIX-based systems, where the default number of file descriptors is restricted to 60 per process. The X Window System server is one process and is restricted by this limit. Although it is possible to increase this limit, it is necessary both to configure a new UNIX kernel and rebuild the X Window System (clients and server) for your workstation. If you have the necessary UNIX and X Window System sources to do this, documentation is provided with the source describing how to increase this limit.

SETTING THE DISPLAY ENVIRONMENT VARIABLE

There is no default X Window System server running on the iPSC/860 system. Therefore, you must remember to set the environment variable **DISPLAY** on your workstation. It is also possible to specify the display as a command line argument in an application program.

For example, assume that you want to set the **DISPLAY** environmental variable and that the name of your workstation is *workname*. Before loading a node program that makes X client calls, issue the command:

```
% setenv DISPLAY workname:0
```

iPSC® SYSTEM COMMANDS, SYSTEM CALLS, and ROUTINES A

Table A-1. SRM Cube Command Summary (1 of 2)

Command Invocation	Description
archcube [**-c** *cubename*]	Displays architecture type of the cube. (i860™ or Intel386™)
attachcube [**-c** *cubename*]	Make the default or specified *cubename* the currently attached cube.
bootcube [**-B** *file*] [**-b** *file*] [**-C**] [**-D***n* | **-D***m-n*] [**-d** *dim* | **-n** *nodes*] [**-f**] [**-K** *file*] [**-k** *file*] [**-L**] [**-l**] [**-o**] [**-Q** *file*] [**-R** *file*] [**-r** *file*] [**-S** *file*] [**-s** *file*] [**-T** *file*] [**-U** *file*]	Load NX/2 operating system onto nodes, and start iPSC daemons. (System Administrator only)
cubeinfo [**-a** | **-s** | **-n** | **-h** *srmname*]	Displays cube ownership information.
fsplit [*file*]	Splits a multi-routine Fortran file into individual files
getcube [**-c** *cubename*] [**-t** *cubetype*] [**-h** *srmname*]	Allocate a cube, and make it the currently attached cube.
killcube [**-c** *cubename*] [**-p** *pid*] [*node...*]	Kill node process(es).
load [**-c** *cubename*] [**-p** *pid*] [**-H**] [*node...*] *filename* [*arguments...*]	Loads a user process into the cube.
mkcfs	Construct a file system. (System Administrator only)

Table A-1. SRM Cube Command Summary (2 of 2)

Command Invocation	Description
newserver [**-c** *cubename*]	Start a new file server for the specified cube.
nsh [*arguments*]	Load C-shell onto node 0.
plogoff	Stop logging process-related information. (System Administrator only)
plogon [**-a**] [**-c**] [*pathname*]	Start logging process-related information. (System Administrator only)
rar [**-h** *srmname*] [*ar_options*] *filename...* **ras** [**-h** *srmname*] [*as_options*] *filename...* **rcc** [**-cpp**] [**-h** *srmname*] [*cc_options*] *filename...* **rf77** [**-cpp**] [**-h** *srmname*] [*f77_options*] *filename...* **rld** [**-h** *srmname*] [*ld_options*] *filename...*	Librarian, assembler, C compiler, Fortran compiler, and linker. These remote development utilities let the remote host build node-executable programs. (Available only on remote host.)
rebootcube	Load NX/2 operating system onto nodes, and start iPSC daemons. Same as a **bootcube** with defaults, but does not require root privilege.
relcube [**-c** *cubename* | **-a** | **-f** *cubeid* | **-F**]	Release one or more cubes.
showvol	Display available CFS disk drives.
startcube [**-c** *cubename*] [**-p** *pid*] [*node...*]	Start a process executing on the cube.
syslog [**-e**]	Send output of host process to file server handling I/O from nodes.
waitcube [**-c** *cubename*] [**-p** *pid*] [**-f**] [**-i**] [**-s**] [*node...*]	Wait for process(es) on node(s) to finish before proceeding.

Table A-2. Summary of System Calls for Cube Control (C Version) (1 of 2)

Call	Environment	Synopsis	Description
attachcube()	Host	**attachcube**(*cubename*) char **cubename*;	Attach to a cube and make it the current cube.
cubeinfo()	Host	long **cubeinfo**(*ct*, *numslots*, *global*) struct cubetable **ct*; long *numslots*, *global*;	Obtain information about allocated cubes.
flick()	Host, Node	**flick**()	On a node, relinquish the CPU to another process. On the host, is a no-op.
getcube()	Host	**getcube**(*cubename*, *cubetype*, *srmname*, *keep*) char **cubename*, **cubetype*, **srmname*; long *keep*;	Allocate a cube.
handler()	Node	**handler**(*etype*, *proc*) long *etype*; void(**proc*)();	Provide user-written exception handler for program failures.
killcube()	Host, Node	**killcube**(*node*, *pid*) long *node*, *pid*;	Terminate and clear node process(es).
killproc()	Host, Node	**killproc**(*node*, *pid*) long *node*, *pid*;	Terminate a node process.
killsyslog()	Host	**killsyslog**()	Terminate *syslog* process.
load()	Host, Node	**load**(*filename*, *node*, *pid*) char **filename*; long *node*, *pid*;	Load a node process.
myhost()	Host, Node	long **myhost**()	Obtain node ID of host machine.
mynode()	Host, Node	long **mynode**()	Obtain node ID of calling process.
mypid()	Host, Node	long **mypid**()	Obtain NX/2 process ID of calling process.

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

Call	Environment	Synopsis	Description
newserver()	Host	**newserver**(*cubename*) char **cubename*;	Start new file server for specified cube.
nodedim()	Host, Node	long **nodedim()**	Obtain dimension of current cube.
numnodes()	Host, Node	long **numnodes()**	Obtain number of nodes in current cube.
plogoff()	Host	**plogoff()**	Stop logging process-related information.
plogon()	Host	**plogon**(*append, alloconly, logfile*) short *append, alloconly*; char **logfile*;	Start logging process-related information.
relcube()	Host	**relcube**(*cubename*) char **cubename*;	Release specified cube.
setpid()	Host	**setpid**(*pid*) long *pid*;	Set NX/2 process ID for host program.
setsyslog()	Host	**setsyslog**(*stdfd*) long *stdfd*;	Start *syslog* process.
waitall()	Host, Node	**waitall**(*node, pid*) long *node, pid*;	Wait for all specified processes to complete.
waitone()	Host, Node	**waitone**(*node, pid, cnode, cpid, ccode*) long *node, pid*; long **cnode, *cpid, *ccode*;	Wait for specified processes to complete.

Table A-3. Summary of Routines for Cube Control (Fortran Version) (1 of 2)

Routine	Environment	Synopsis	Description
ATTACHCUBE()	Host	SUBROUTINE **ATTACHCUBE**(*cubename*) CHARACTER *cubename**(*)	Attach to a cube and make it the current cube.
CUBEINFO()	Host	INTEGER FUNCTION **CUBEINFO**(*ct, numslots, global*) CHARACTER*16 *ct*(*slotsize, numslots*) INTEGER *numslots, global*	Obtain information about allocated cubes.
FLICK()	Host, Node	SUBROUTINE **FLICK()**	On a node: Relinquish the CPU to another process. On the host: A no-op.
GETCUBE()	Host	SUBROUTINE **GETCUBE**(*cubename, cubetype, srmname, keep*) CHARACTER *cubename**(*) CHARACTER *cubetype**(*) CHARACTER *srmname**(*) INTEGER *keep*	Allocate a cube.
HANDLER()	Node	SUBROUTINE **HANDLER**(*etype, proc*) INTEGER *etype* EXTERNAL *proc* *proc* must be written in C. It has one argument: the hardware exception code.	Provide user-written exception handler for program failures.
KILLCUBE()	Host, Node	SUBROUTINE **KILLCUBE**(*node, pid*) INTEGER *node, pid*	Terminate and clear node process(es).
KILLPROC()	Host, Node	SUBROUTINE **KILLPROC**(*node, pid*) INTEGER *node, pid*	Terminate a node process.
KILLSYSLOG()	Host	SUBROUTINE **KILLSYSLOG()**	Terminate *syslog* process.
LOAD()	Host, Node	SUBROUTINE **LOAD**(*filename, node, pid*) CHARACTER *filename**(*) INTEGER *node, pid*	Load a node process.

Table A-3. Summary of Routines for Cube Control (Fortran Version) (2 of 2)

Routine	Environment	Synopsis	Description
MYHOST()	Host, Node	INTEGER FUNCTION **MYHOST()**	Obtain node ID of host machine.
MYNODE()	Host, Node	INTEGER FUNCTION **MYNODE()**	Obtain node ID of calling process.
MYPID()	Host, Node	INTEGER FUNCTION **MYPID()**	Obtain NX/2 process ID of calling process.
NEWSERVER()	Host	SUBROUTINE **NEWSERVER**(*cubename*) CHARACTER *cubename**(*)	Start new file server for specified cube.
NODEDIM()	Host, Node	INTEGER FUNCTION **NODEDIM()**	Obtain dimension of current cube.
NUMNODES()	Host, Node	INTEGER FUNCTION **NUMNODES()**	Obtain number of nodes in current cube.
PLOGOFF()	Host	SUBROUTINE **PLOGOFF()**	Stop logging process-related information.
PLOGON()	Host	SUBROUTINE **PLOGON**(*append, alloconly, logfile*) INTEGER *append, alloconly* CHARACTER *logfile**(*)	Start logging process-related information.
RELCUBE()	Host	SUBROUTINE **RELCUBE**(*cubename*) CHARACTER *cubename**(*)	Release specified cube.
SETPID()	Host	SUBROUTINE **SETPID**(*pid*) INTEGER *pid*	Set NX/2 process ID for host program.
SETSYSLOG()	Host	SUBROUTINE **SETSYSLOG**(*stdfd*) INTEGER *stdfd*	Start *syslog* process.
WAITALL()	Host, Node	SUBROUTINE **WAITALL**(*node, pid*) INTEGER *node, pid*	Wait for all specified processes to complete.
WAITONE()	Host, Node	SUBROUTINE **WAITONE**(*node, pid, cnode, cpid, ccode*) INTEGER *node, pid* INTEGER *cnode, cpid, ccode*	Wait for specified processes to complete.

Table A-4. Summary of System Calls for Message Passing (C Version) (1 of 3)

Call	Environment	Synopsis	Description
cprobe()	Host, Node	**cprobe**(*typesel*) long *typesel*;	Wait for a message to arrive.
crecv()	Host, Node	**crecv**(*typesel*, *buf*, *len*) long *typesel*; char **buf*; long *len*;	Receive a message, and wait for completion.
csend()	Host, Node	**csend**(*type*, *buf*, *len*, *node*, *pid*) long *type*; char **buf*; long *len*, *node*, *pid*;	Send a message, and wait for completion.
csendrecv()	Host, Node	long **csendrecv**(*type*, *sbuf*, *slen*, *tonode*, *topid*, *typesel*, *rbuf*, *rlen*) long *type*; char **sbuf*; long *slen*, *tonode*, *topid*, *typesel*; char **rbuf*; long *rlen*;	Simultaneously, send a message, and post a receive for the reply. Wait for completion.
flushmsg()	Host, Node	**flushmsg**(*typesel*, *node*, *pid*) long *typesel*, *node*, *pid*;	Flush specified messages from the system.
hrecv()	Node	**hrecv**(*typesel*, *buf*, *len*, *proc*) long *typesel*; char **buf*; long *len*; void(**proc*)(); proc(*type*, *count*, *node*, *pid*) long *type*, *count*, *node*, *pid*;	Provide user-written exception handler for receive traps.
hsend()	Node	**hsend**(*type*, *buf*, *len*, *node*, *pid*, *proc*) long *type*; char **buf*; long *len*, *node*, *pid*; void(**proc*)(); proc(*type*, *count*, *node*, *pid*) long *type*, *count*, *node*, *pid*;	Send a message, and set up a handler procedure to be called when the reply arrives.

Table A-4. Summary of System Calls for Message Passing (C Version) (2 of 3)

Call	Environment	Synopsis	Description
hsendrecv()	Node	**hsendrecv**(*type*, *sbuf*, *slen*, *tonode*, *topid*, *typesel*, *rbuf*, *rlen*, *proc*) long *type*; char **sbuf*; long *slen*, *tonode*, *topid*, *typesel*; char **rbuf*; long *rlen*; void(**proc*)(); proc(*type*, *count*, *node*, *pid*) long *type*, *count*, *node*, *pid*;	Simultaneously, send a message, and post a receive for the reply. Also, set up a handler procedure to be called when the reply arrives.
infocount() **infonode()** **infopid()** **infotype()**	Host, Node	long **infocount()** long **infonode()** long **infopid()** long **infotype()**	Return information about a pending or received message.
iprobe()	Host, Node	long **iprobe**(*typesel*) long *typesel*;	Determine whether a message of a selected type is pending.
irecv()	Host, Node	long **irecv**(*typesel*, *buf*, *len*) long *typesel*; char **buf*; long *len*;	Receive a message.
isend()	Host, Node	long **isend**(*type*, *buf*, *len*, *node*, *pid*) long *type*; char **buf*; long *len*, *node*, *pid*;	Send a message.
isendrecv()	Host, Node	long **isendrecv**(*type*, *sbuf*, *slen*, *tonode*, *topid*, *typesel*, *rbuf*, *rlen*) long *type*; char **sbuf*; long *slen*, *tonode*, *topid*, *typesel*; char **rbuf*; long *rlen*;	Simultaneously, send a message, and post a receive for the reply. Do not wait for completion.
masktrap()	Node	long **masktrap**(*state*) long *state*;	Enable or disable receive trap.
msgcancel()	Host, Node	**msgcancel**(*id*) long *id*;	Cancel a send or receive operation.

Table A-4. Summary of System Calls for Message Passing (C Version) (3 of 3)

Call	Environment	Synopsis	Description
msgdone()	Host, Node	long **msgdone**(*id*) long *id*;	Determine whether a send or receive operation has completed.
msgwait()	Host, Node	**msgwait**(*id*) long *id*;	Wait for completion of a send or receive operation.

Table A-5. Summary of Routines for Message Passing (Fortran Version) (1 of 3)

Routine	Environment	Synopsis	Description
CPROBE()	Host, Node	SUBROUTINE **CPROBE**(*typesel*) INTEGER *typesel*	Wait for a message to arrive.
CRECV()	Host, Node	SUBROUTINE **CRECV**(*typesel, buf, len*) INTEGER *typesel* INTEGER *buf*(*) INTEGER *len*	Receive a message, and wait for completion.
CSEND()	Host, Node	SUBROUTINE **CSEND**(*type, buf, len,node, pid*) INTEGER *type* INTEGER *buf*(*) INTEGER *len, node, pid*	Send a message, and wait for completion.
CSENDRECV()	Host, Node	INTEGER FUNCTION **CSENDRECV**(*type, sbuf, slen, tonode, topid, typesel, rbuf, rlen*) INTEGER *type* INTEGER *sbuf*(*) INTEGER *slen, tonode, topid, typesel* INTEGER *rbuf*(*) INTEGER *rlen*	Simultaneously, send a message, and post a receive for the reply. Wait for completion.
FLUSHMSG()	Host, Node	SUBROUTINE **FLUSHMSG**(*typesel, node, pid*) INTEGER *typesel, node, pid*	Flush specified messages from the system.
HRECV()	Node	SUBROUTINE **HRECV**(*typesel, buf, len, proc*) INTEGER *typesel* INTEGER *buf*(*) INTEGER *len* EXTERNAL *proc* *proc* must be written in C: proc(*type, count, node, pid*) long *type, count, node, pid*;	Provide user-written exception handler for receive traps.

Table A-5. Summary of Routines for Message Passing (Fortran Version) (2 of 3)

Routine	Environment	Synopsis	Description
HSEND()	Node	SUBROUTINE **HSEND**(*type*, *buf*, *len*, *node*, *pid*, *proc*) INTEGER *type* INTEGER *buf*(*) INTEGER *len*, *node*, *pid* EXTERNAL *proc* *proc* must be written in C: proc(*type*, *count*, *node*, *pid*) long *type*, *count*, *node*, *pid*;	Send a message, and set up a handler procedure to be called when the reply arrives.
HSENDRECV()	Node	SUBROUTINE **HSENDRECV**(*type*, *sbuf*, *slen*, *tonode*, *topid*, *typesel*, *rbuf*, *rlen*, *proc*) INTEGER *type* INTEGER *sbuf*(*) INTEGER *slen*, *tonode*, *topid*, *typesel* INTEGER *rbuf*(*) INTEGER *rlen* EXTERNAL *proc* *proc* must be written in C: proc(*type*, *count*, *node*, *pid*) long *type*, *count*, *node*, *pid*;	Simultaneously, send a message, and post a receive for the reply. Also, set up a handler procedure to be called when the reply arrives.
INFOCOUNT() **INFONODE()** **INFOPID()** **INFOTYPE()**	Host, Node	INTEGER FUNCTION **INFOCOUNT()** INTEGER FUNCTION **INFONODE()** INTEGER FUNCTION **INFOPID()** INTEGER FUNCTION **INFOTYPE()**	Return information about a pending or received message.
IPROBE()	Host, Node	INTEGER FUNCTION **IPROBE**(*typesel*) INTEGER *typesel*	Determine whether a message of a selected type is pending.
IRECV()	Host, Node	INTEGER FUNCTION **IRECV**(*typesel*, *buf*, *len*) INTEGER *typesel* INTEGER *buf*(*) INTEGER *len*	Receive a message.
ISEND()	Host, Node	INTEGER FUNCTION **ISEND**(*type*, *buf*, *len*, *node*, *pid*) INTEGER *type* INTEGER *buf*(*) INTEGER *len*, *node*, *pid*	Send a message.

Table A-5. Summary of Routines for Message Passing (Fortran Version) (3 of 3)

Routine	Environment	Synopsis	Description
ISENDRECV()	Host, Node	INTEGER FUNCTION **ISENDRECV**(*type, sbuf, slen, tonode,* *topid, typesel, rbuf, rlen*) INTEGER *type* INTEGER *sbuf*(*) INTEGER *slen, tonode, topid, typesel* INTEGER *rbuf*(*) INTEGER *rlen*	Simultaneously, send a message, and post a receive for the reply. Do not wait for completion.
MASKTRAP()	Node	INTEGER FUNCTION **MASKTRAP**(*state*) INTEGER *state*	Enable or disable a receive trap.
MSGCANCEL()	Host, Node	SUBROUTINE **MSGCANCEL**(*id*) INTEGER *id*	Cancel a send or receive operation.
MSGDONE()	Host, Node	INTEGER FUNCTION **MSGDONE**(*id*) INTEGER *id*	Determine whether a send or receive operation has completed.
MSGWAIT()	Host, Node	SUBROUTINE **MSGWAIT**(*id*) INTEGER *id*	Wait for completion of a send or receive operation.

Table A-6. Global Operations (C Version) (1 of 3)

Call	Environment	Synopsis	Description
gcol()	Node	**gcol**(*x*, *xlen*, *y*, *ylen*, *ncnt*) char *x*[]; long *xlen*; char *y*[]; long *ylen*; long **ncnt*;	Global concatenation operation.
gcolx()	Node	**gcolx**(*x*, *xlens*, *y*) char *x*[]; long *xlens*; char *y*[];	Global concatenation operation for contributions of known length.
gdhigh()	Node	**gdhigh**(*x*, *n*, *work*) double *x*[]; long *n*; double *work*[];	Global vector double precision MAX operation.
gdlow()	Node	**gdlow**(*x*, *n*, *work*) double *x*[]; long *n*; double *work*[];	Global vector double precision MIN operation.
gdprod()	Node	**gdprod**(*x*, *n*, *work*) double *x*[]; long *n*; double *work*[];	Global vector double precision MULTIPLY operation.
gdsum()	Node	**gdsum**(*x*, *n*, *work*) double *x*[]; long *n*; double *work*[];	Global vector double precision SUM operation.
giand()	Node	**giand**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Global vector integer bitwise AND operation.
gihigh()	Node	**gihigh**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Global vector integer MAX operation.

Table A-6. Global Operations (C Version) (2 of 3)

Call	Environment	Synopsis	Description
gilow()	Node	**gilow**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Global vector integer MIN operation.
gior()	Node	**gior**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Global vector integer bitwise OR operation.
giprod()	Node	**giprod**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Global vector integer MULTIPLY operation.
gisum()	Node	**gisum**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Global vector integer SUM operation.
gixor()	Node	**gixor**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Global vector integer bitwise exclusive OR operation.
gland()	Node	**gland**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Global vector logical AND operation.
glor()	Node	**glor**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Global vector logical inclusive OR operation.
glxor()	Node	**glxor**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Global vector logical exclusive OR operation.

Table A-6. Global Operations (C Version) (3 of 3)

Call	Environment	Synopsis	Description
gopf()	Node	**gopf**(*x*, *xlen*, *work*, *f*) char *x*[]; long *xlen*; char *work*[]; long (**f*)();	Arbitrary commutative function.
gsendx()	Node	**gsendx**(*type*, *x*, *xlen*, *nodenums*, *nlen*) long *type*; char *x*[]; long *xlen*; long *nodenums*[]; long *nlen*;	Send a vector to a list of nodes.
gshigh()	Node	**gshigh**(*x*, *n*, *work*) float *x*[]; long *n*; float *work*[];	Global vector real MAX operation.
gslow()	Node	**gslow**(*x*, *n*, *work*) float *x*[]; long *n*; float *work*[];	Global vector real MIN operation.
gsprod()	Node	**gsprod**(*x*, *n*, *work*) float *x*[]; long *n*; float *work*[];	Global vector real MULTIPLY operation.
gssum()	Node	**gssum**(*x*, *n*, *work*) float *x*[]; long *n*; float *work*[];	Global vector real SUM operation.
gsync()	Node	**gsync()**	Global synchronization operation.

Table A-7. Global Operations (Fortran Version) (1 of 3)

Routine	Environment	Synopsis	Description
GCOL()	Node	SUBROUTINE **GCOL**(*x, xlen, y, ylen, ncnt*) INTEGER *x*(*) INTEGER *xlen* INTEGER *y*(*) INTEGER *ylen* INTEGER *ncnt*	Global concatenation operation.
GCOLX()	Node	SUBROUTINE **GCOLX**(*x, xlens, y*) INTEGER *x*(*) INTEGER *xlen* INTEGER *y*(*)	Global concatenation operation for contributions of known length.
GDHIGH()	Node	SUBROUTINE **GDHIGH**(*x, n, work*) DOUBLE PRECISION *x*(*) INTEGER *n* DOUBLE PRECISION *work*(*)	Global vector double precision MAX operation.
GDLOW()	Node	SUBROUTINE **GDLOW**(*x, n, work*) DOUBLE PRECISION *x*(*) INTEGER *n* DOUBLE PRECISION *work*(*)	Global vector double precision MIN operation.
GDPROD()	Node	SUBROUTINE **GDPROD**(*x, n, work*) DOUBLE PRECISION *x*(*) INTEGER *n* DOUBLE PRECISION *work*(*)	Global vector double precision MULTIPLY operation.
GDSUM()	Node	SUBROUTINE **GDSUM**(*x, n, work*) DOUBLE PRECISION *x*(*) INTEGER *n* DOUBLE PRECISION *work*(*)	Global vector double precision SUM operation.
GIAND()	Node	SUBROUTINE **GIAND**(*x, n, work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Global vector integer bitwise AND operation.
GIHIGH()	Node	SUBROUTINE **GIHIGH**(*x, n, work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Global vector integer MAX operation.

Table A-7. Global Operations (Fortran Version) (2 of 3)

Call	Environment	Synopsis	Description
GILOW()	Node	SUBROUTINE **GILOW**(*x*, *n*, *work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Global vector integer MIN operation.
GIOR()	Node	SUBROUTINE **GIOR**(*x*, *n*, *work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Global vector integer bitwise OR operation.
GIPROD()	Node	SUBROUTINE **GIPROD**(*x*, *n*, *work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Global vector integer MULTIPLY operation.
GISUM()	Node	SUBROUTINE **GISUM**(*x*, *n*, *work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Global vector integer SUM operation.
GIXOR()	Node	SUBROUTINE **GIXOR**(*x*, *n*, *work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Global vector integer bitwise exclusive OR operation.
GLAND()	Node	SUBROUTINE **GLAND**(*x*, *n*, *work*) LOGICAL *x*(*) INTEGER *n* LOGICAL *work*(*)	Global vector logical AND operation.
GLOR()	Node	SUBROUTINE **GLOR**(*x*, *n*, *work*) LOGICAL *x*(*) INTEGER *n* LOGICAL *work*(*)	Global vector logical inclusive OR operation.
GLXOR()	Node	SUBROUTINE **GLXOR**(*x*, *n*, *work*) LOGICAL *x*(*) INTEGER *n* LOGICAL *work*(*)	Global vector logical exclusive OR operation.

Table A-7. Global Operations (Fortran Version) (3 of 3)

Call	Environment	Synopsis	Description
GOPF()	Node	SUBROUTINE **GOPF**(*x, xlen, work, f*) INTEGER *x*(*) INTEGER *xlen* INTEGER *work*(*) EXTERNAL *f*	Arbitrary commutative function.
GSENDX()	Node	SUBROUTINE **GSENDX**(*type, x, xlen, nodenums, nlen*) INTEGER *type* INTEGER *x*(*) INTEGER *xlen* INTEGER *nodenums*(*) INTEGER *nlen*	Send a vector to a list of nodes.
GSHIGH()	Node	SUBROUTINE **GSHIGH**(*x, n, work*) REAL *x*(*) INTEGER *n* REAL *work*(*)	Global vector real MAX operation.
GSLOW()	Node	SUBROUTINE **GSLOW**(*x, n, work*) REAL *x*(*) INTEGER *n* REAL *work*(*)	Global vector real MIN operation.
GSPROD()	Node	SUBROUTINE **GSPROD**(*x, n, work*) REAL *x*(*) INTEGER *n* REAL *work*(*)	Global vector real MULTIPLY operation.
GSSUM()	Node	SUBROUTINE **GSSUM**(*x, n, work*) REAL *x*(*) INTEGER *n* REAL *work*(*)	Global vector real SUM operation.
GSYNC()	Node	SUBROUTINE **GSYNC()**	Global synchronization operation.

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

Call	Environment	Synopsis	Description
createstruc()	Host	**createstruc**(*ptr*) char **ptr*;	Initialize array used when byte-swapping structures.
CTOHC()	Host	**CTOHC**(*sv, n*) unsigned char **sv*; long *n*;	Used when byte-swapping structures with chars, cube to host.
CTOHD()	Host	**CTOHD**(*sv, n*) double **sv*; long *n*;	Byte-swap cube to host double.
CTOHF()	Host	**CTOHF**(*sv, n*) float **sv*; long *n*;	Byte-swap cube to host float.
CTOHL()	Host	**CTOHL**(*sv, n*) long **sv*; long *n*;	Byte-swap cube to host long.
CTOHS()	Host	**CTOHS**(*sv, n*) short **sv*; long *n*;	Byte-swap cube to host short.
HTOCC()	Host	**HTOCC**(*sv, n*) unsigned char **sv*; long *n*;	Used when byte-swapping structures with chars, cube to host.
HTOCD()	Host	**HTOCD**(*sv, n*) double **sv*; long *n*;	Byte-swap host to cube double.
HTOCF()	Host	**HTOCF**(*sv, n*) float **sv*; long *n*;	Byte-swap host to cube float.
HTOCL()	Host	**HTOCL**(*sv, n*) long **sv*; long *n*;	Byte-swap host to cube long.
HTOCS()	Host	**HTOCS**(*sv, n*) short **sv*; long *n*;	Byte-swap host to cube short.
relstruc()	Host	unsigned long **relstruc**()	Release array used when byte-swapping structures.

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

Routine	Environment	Synopsis	Description
CTOHD()	Host	SUBROUTINE **CTOHD**(*sv*, *n*) DOUBLE PRECISION *sv*(*) INTEGER *n*	Byte-swap cube to host double precision.
CTOHF()	Host	SUBROUTINE **CTOHF**(*sv*, *n*) REAL *sv*(*) INTEGER *n*	Byte-swap cube to host real.
CTOHL()	Host	SUBROUTINE **CTOHL**(*sv*, *n*) INTEGER*4 *sv*(*) INTEGER *n*	Byte-swap cube to host integer.
CTOHS()	Host	SUBROUTINE **CTOHS**(*sv*, *n*) INTEGER*2 *sv*(*) INTEGER *n*	Byte-swap cube to host short integer.
HTOCD()	Host	SUBROUTINE **HTOCD**(*sv*, *n*) DOUBLE PRECISION *sv*(*) INTEGER *n*	Byte-swap host to cube double precision.
HTOCF()	Host	SUBROUTINE **HTOCF**(*sv*, *n*) REAL *sv*(*) INTEGER *n*	Byte-swap host to cube real.
HTOCL()	Host	SUBROUTINE **HTOCL**(*sv*, *n*) INTEGER*4 *sv*(*) INTEGER *n*	Byte-swap host to cube integer.
HTOCS()	Host	SUBROUTINE **HTOCS**(*sv*, *n*) INTEGER*2 *sv*(*) INTEGER *n*	Byte-swap host to cube short integer.

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

Call	Environment	Synopsis	Description
dclock()	Node	double **dclock()**	Returns time in seconds since booting the cube.
ginv()	Host, Node	long **ginv**(*j*) long *j*;	Return the position of an element in the binary-reflected gray code sequence. Inverse of **gray()**.
gray()	Host, Node	long **gray**(*j*) long *j*;	Return the binary-reflected gray code for an integer.
hwclock()	Node	void **hwclock**(*hwtime*) esize_t **hwtime*;	Places the current value of the hardware counter in the 64-bit unsigned integer.
led()	Node	void **led**(*lstate*) long *lstate*;	Turn the node's green LED on or off.
mclock()	Host, Node	unsigned long **mclock()**	Return the time (in milliseconds).

Table A-11. Miscellaneous Routines (Fortran Version)

Routine	Environment	Synopsis	Description
DCLOCK()	Node	DOUBLE PRECISION **DCLOCK()**	Returns time in seconds since booting the cube.
GINV()	Host, Node	INTEGER FUNCTION **GINV**(*j*) INTEGER *j*	Return the position of an element in the binary-reflected gray code sequence. Inverse of **GRAY()**.
GRAY()	Host, Node	INTEGER FUNCTION **GRAY**(*j*) INTEGER *j*	Return the binary-reflected gray code for an integer.
HWCLOCK()	Node	SUBROUTINE **HWCLOCK**(*hwtime*) INTEGER *hwtime*(2)	Places the current value of the hardware counter in the 64-bit unsigned integer.
LED()	Node	SUBROUTINE **LED**(*lstate*) INTEGER *lstate*	Turn the node's green LED on or off.
MCLOCK()	Node	INTEGER FUNCTION **MCLOCK()**	Return elapsed time (in milliseconds).

Table A-12. Summary of System Calls for Concurrent File I/O (C Version) (1 of 2)

Call	Environment	Synopsis	Description
cread()	Node	**cread**(*fildes*, *buf*, *len*) long *fildes*; char **buf*; long *len*;	Read from a file, and wait for completion.
cwrite()	Node	**cwrite**(*fildes*, *buf*, *len*) long *fildes*; char **buf*; long *len*;	Write to a file, and wait for completion.
eseek()	Node	esize_t **eseek**(*fildes*, *offset*, *whence*) long *fildes*; esize_t *offset*; long *whence*;	Move file pointer in extended file.
esize()	Node	esize_t **esize**(*fildes*, *offset*, *whence*) long *fildes*; esize_t *offset*; long *whence*;	Increase size of extended file.
estat()	Node	long **estat**(*path*, *buf*) char **path*; struct estat **buf*;	Get status of extended file.
festat()	Node	long **festat**(*fildes*, *buf*) long *fildes*; struct estat **buf*;	Get status of extended file from file descriptor.
iodone()	Node	long **iodone**(*id*) long *id*;	Determine whether an asynchronous I/O read or write operation is complete.
iomode()	Node	long **iomode**(*fildes*) long *fildes*;	Return the current I/O mode for a file.
iowait()	Node	**iowait**(*id*) long *id*;	Wait for completion of an asynchronous I/O read or write operation.
iread()	Node	long **iread**(*fildes*, *buf*, *len*) long *fildes*; char **buf*; long *len*;	Asynchronous read from a file. (Do not wait for completion.)

Table A-12. Summary of System Calls for Concurrent File I/O (C Version) (2 of 2)

Call	Environment	Synopsis	Description
iseof()	Node	long **iseof**(*fildes*) long *fildes*;	Test for end-of-file.
iwrite()	Node	long **iwrite**(*fildes*, *buf*, *len*) long *fildes*; char **buf*; long *len*;	Asynchronous write to a file. (Do not wait for completion.)
lsize()	Node	long **lsize**(*fildes*, *offset*, *whence*) long *fildes*, *offset*, *whence*;	Increase size of a file.
restrictvol()	Node	long **restrictvol**(*fildes*, *nvol*, *vollist*) long *fildes*, *nvol*, *vollist*[];	Restrict or show the disk volumes to which a file can be allocated.
setiomode()	Node	**setiomode**(*fildes*, *mode*) long *fildes*, *mode*;	Set the I/O mode for a file.

Table A-13. Summary of Routines for Concurrent File I/O (Fortran Version) (1 of 2)

Routine	Environment	Synopsis	Description
CREAD()	Node	SUBROUTINE **CREAD**(*unit*, *buf*, *len*) INTEGER *unit* INTEGER *buf*(*) INTEGER *len*	Read from a file, and wait for completion.
CWRITE()	Node	SUBROUTINE **CWRITE**(*unit*, *buf*, *len*) INTEGER *unit* INTEGER *buf*(*) INTEGER *len*	Write to a file, and wait for completion.
ESEEK()	Node	SUBROUTINE **ESEEK**(*unit*, *offset*, *whence*, *newpos*) INTEGER *unit* INTEGER *offset*(2) INTEGER *whence* INTEGER *newpos*(2)	Move file pointer in large file.
ESIZE()	Node	SUBROUTINE **ESIZE**(*unit*, *offset*, *whence*, *newsize*) INTEGER *unit* INTEGER *offset*(2) INTEGER *whence* INTEGER *newsize*(2)	Increase size of large file.
FORCEFLUSH()	Node	SUBROUTINE **FORCEFLUSH()**	Flush all buffered I/O when an exception occurs.
FORFLUSH()	Node	SUBROUTINE **FORFLUSH**(*iunit*) INTEGER *iunit*	Complete all buffered I/O on the specified unit.
IODONE()	Node	INTEGER FUNCTION **IODONE**(*id*) INTEGER *id*	Determine whether an asynchronous I/O read or write operation is complete.
IOMODE()	Node	INTEGER FUNCTION **IOMODE**(*unit*) INTEGER *unit*	Return the current I/O mode for a file.
IOWAIT()	Node	SUBROUTINE **IOWAIT**(*id*) INTEGER *id*	Wait for completion of an asynchronous I/O read or write operation.

Table A-13. Summary of Routines for Concurrent File I/O (Fortran Version) (2 of 2)

Routine	Environment	Synopsis	Description
IREAD()	Node	INTEGER FUNCTION **IREAD**(*unit*, *buf*, *len*) INTEGER *unit* INTEGER *buf*(*) INTEGER *len*	Asynchronous read from a file. (Do not wait for completion.)
ISEOF()	Node	INTEGER FUNCTION **ISEOF**(*unit*) INTEGER *unit*	Test for end-of-file.
IWRITE()	Node	INTEGER FUNCTION **IWRITE**(*unit*, *buf*, *len*) INTEGER *unit* INTEGER *buf*(*) INTEGER *len*	Asynchronous write to a a file. (Do not wait for completion.)
LSEEK()	Node	INTEGER FUNCTION **LSEEK**(*unit*, *offset*, *whence*) INTEGER *unit* INTEGER *offset* INTEGER *whence*	Move the read/write file pointer.
LSIZE()	Node	INTEGER FUNCTION **LSIZE**(*unit*, *offset*, *whence*) INTEGER *unit* INTEGER *offset* INTEGER *whence*	Increase size of a file.
RESTRICTVOL()	Node	INTEGER FUNCTION **RESTRICTVOL**(*unit*, *nvol*, *vollist*) INTEGER *unit* INTEGER *nvol* INTEGER *vollist*(*)	Restrict or show the disk volumes to which a file can be allocated.
SETIOMODE()	Node	SUBROUTINE **SETIOMODE**(*unit*, *mode*) INTEGER *unit* INTEGER *mode*	Set the I/O mode for a file.

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

Call	Environment	Synopsis	Description
eadd()	Node	esize_t **eadd**(*e1*, *e2*) esize_t *e1*; esize_t *e2*;	Add two extended numbers.
ecmp()	Node	long **ecmp**(*e1*, *e2*) esize_t *e1*; esize_t *e2*;	Compare two extended numbers.
ediv()	Node	long **ediv**(*e*, *n*) esize_t *e*; long *n*;	Divide extended number by integer.
emod()	Node	long **emod**(*e*, *n*) esize_t *e*; long *n*;	Display remainder of **ediv()**.
emul()	Node	esize_t **emul**(*e*, *n*) esize_t *e*; long *n*;	Multiply extended number by integer.
esub()	Node	esize_t **esub**(*e1*, *e2*) esize_t *e1*; esize_t *e2*;	Subtract two extended numbers.
etos()	Node	**etos**(*e*, *s*) esize_t *e*; char **s*;	Convert extended number to string.
stoe()	Node	esize_t **stoe**(*s*) char **s*;	Convert string to extended number.

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

Routine	Environment	Synopsis	Description
EADD()	Node	SUBROUTINE **EADD**(*e1*, *e2*, *eresult*) INTEGER *e1*(2) INTEGER *e2*(2) INTEGER *eresult*(2)	Add two extended numbers.
ECMP()	Node	INTEGER FUNCTION **ECMP**(*e1*, *e2*) INTEGER *e1*(2) INTEGER *e2*(2)	Compare two extended numbers.
EDIV()	Node	SUBROUTINE **EDIV**(*e*, *n*, *result*) INTEGER *e*(2) INTEGER *n* INTEGER *result*	Divide extended number by integer.
EMOD()	Node	SUBROUTINE **EMOD**(*e*, *n*, *result*) INTEGER *e*(2) INTEGER *n* INTEGER *result*	Display remainder of **EDIV()**.
EMUL()	Node	SUBROUTINE **EMUL**(*e*, *n*, *eresult*) INTEGER *e*(2) INTEGER *n* INTEGER *eresult*(2)	Multiply extended number by integer.
ESUB()	Node	SUBROUTINE **ESUB**(*e1*, *e2*, *eresult*) INTEGER *e1*(2) INTEGER *e2*(2) INTEGER *eresult*(2)	Subtract two extended numbers.
ETOS()	Node	SUBROUTINE **ETOS**(*e*, *s*) INTEGER *e*(2) CHARACTER *s*(*)	Convert extended number to string.
FPSETMASK()	Node	SUBROUTINE **FPSETMASK**(*mask*) INTEGER *mask*	Set 387™ exception mask.
STOE()	Node	SUBROUTINE **STOE**(*s*, *e*) CHARACTER *s*(*) INTEGER *e*(2)	Convert string to extended number.

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

C-Shell Built-Ins		UNIX Utilities	
alias	login	awk	ln
break	logout	cat	ls[1]
breaksw	newgrp	chgrp	lsize
case	nice	chmod	mkdir
cd	nohup	chown	mv
chdir	onintr	cmp	pr
continue	rehash	cp	ps
default	repeat	csh	pwd
echo	set	dd	rm
else	sentenv	df	rmdir
end	shift	diff	sed
endif	source	diffh	star
endsw	switch	du	stream
exec	time	egrep	sync
exit	umask	env	tail
foreach	unalias	fgrep	tar
glob	unhash	file	touch
goto	unset	grep	wc
hashstat	unsetenv	kill	
history	wait		
if	while		

1. Does not show . and ..

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

access	execvp	lseek	sigpause
alarm	exit	mkdir	sigrelse
brk	fork	nice[2]	sigset
chdir	fstat	open	stat
chmod	fstatfs	pause	stafs
chown	getegid	pipe	sync
close	geteuid	read	sysi86[3]
creat	getgid	rmdir	time
dup	getpid	sbrk	times
excel	getuid	setgid	umask
execle	gtty	setuid	unlink
execlp	ioctl[1]	sighold	utime
execv	kill	sigignore	wait
execve	link	signal	write

1. Supports only the following requests on terminal devices:

   TCGETA
   TCSETA
   TCSETAW
   TCSETAF
   TCSBRK
   RCXONC
   TCFLSH

2. No operation.
3. Supports only the SI86FPHW and the SI86MEM commands.

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

a641	fpgetsticky	getw	printf	strdup
abort	fprintf	gmtime	putchar	strlen
abs	fpsetmask	gsignal	putenv	strncat
asctime	fpsetround	hcreate	putpwent	strncmp
atof	fpsetsticky	hdestroy	puts	strncpy
atoi	fputc	hsearch	putw	strpbrk
atol	fputs	isatty	qsort	strrchr
bsearch	fread	isnand	rand	strspn
calloc	free	jrand48	readdir	strtod
clearerr	freopen	l3tol	realloc	strtok
closedir	frexp	l64a	rewind	strtol
crypt	fscanf	lcong48	rewinddir	swab
ctermid	fseek	ldexp	scanf	tdelete
ctime	ftell	lfind	seed48	telldir
cuserid	ftok	localtime	seekdir	tempnam
drand48	ftw	longjmp	setbuf	tfind
ecvt	fwrite	lrand48	setgrent	tmpfile
encrypt	gcvt	lsearch	setjmp	tmpnam
endgrent	getchar	ltol3	setkey	tolower
endpwent	getcwd	malloc	setpwent	toupper
erand48	getenv	memccpy	setvbuf	tsearch
fclose	getgrent	memchr	sleep	ttyname
fcvt	getgrgid	memcmp	sprintf	ttyslot
fdopen	getgrnam	memcpy	srand	twalk
fflush	getlogin	memset	srand48	tzset
fgetc	getopt	mktemp	sscanf	ungetc
fgetgrent	getpass	modf	ssignal	vfprintf
fgetpwent	getpw	mrand48	strcat	vprintf
fgets	getpwent	nlist	strchr	vsprintf
fopen	getpwnam	nrand48	strcmp	
fpgetmask	getpwuid	opendir	strcpy	
fpgetround	gets	perror	strcspn	

Table A-19. Summary of SRM UNIX Extensions

Command Invocation	Description
cptape [*-i* | *-o* | *-t*] [*-f devicename*] *filename...*	Provide interface to **cpio** command.
head [*-n*] [*filename...*]	Display first few lines of specified file(s).
less [**-dstwcCeEmMqQuU**] [**-h** *N*] [**-b[fp]** *N*] [**-x** *N*] [**-[z]** *N*] [**-P[mM]** *string*] [**-[lL]** *logfile*] [+ *cmd*] [*filename...*]	Similar to **more**, but with additional features.
star [-]c[wfb] *starfile blocksize filename...* **star [-]r[wb]** *starfile blocksize* [*filename...*] **star [-]t[f]** *starfile* **star [-]u[wb]** *starfile blocksize* [*filename...*] **star [-]x[lmopwf]** *starfile* [*filename...*]	Create new tape archive. Add to existing tape archive. List contents of tape archive. Update existing tape archive. Extract files from tape archive.
stream [*KBytes_of_buffer*] [*source*] [*destination*]	Transfer large amounts of data to or from tape.

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

Command Invocation	Description
archcube [**-c** *cubename*]	Displays architecture type of the current cube (i860 or Intel386).
cbackup [**-s** *mediasize*] [**-a**] [*volume...*] [*filename*]	Back up CFS disk drive(s).
crestore [*volume...*] [*filename*]	Restore backed up CFS disk image.
killcube -p *pid* [*node...*]	Kill node process(es).
load -p *pid* [*node...*] *filename* [*arguments...*]	Loads a user process into the cube.
mkdev *volume devicename*	Allocates a file that represents a tape managed by CFS.
mt [**-f** *tapename*] *command* [*count*]	A magnetic tape manipulating program.
showvol	Display information about available drives.
tapemode [*switches*] *devicename...*	Changes operating mode for tape drives.
waitcube -p *pid* [**-f**] [**-i**] [*node...*]	Wait for process(es) on node(s) to finish before proceeding.

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

Command Invocation	Description
getiphosts()	Obtain a list of the names of available socket nodes.
setiphost(*iphost*)	Specify a particular socket node as the iphost. The system dynamically chooses a default iphost based on load balancing. This call gives you the ability to override the system's decision.

iPSC® SYSTEM FEATURES B

Table B-1. iPSC® System Features

Topic	iPSC®/2 and iPSC®/860 Systems
Node Processor	iPSC/2 system has Intel386 microprocessor with Intel387 numeric coprocessor. iPSC/860 system has i860 microprocessor.
Message Passing	Messages can be sent directly from any node processor to any other node processor in the system via the Direct-Connect Module.
Remote Workstations	Multiple remote workstations can share the system and the software development tools on the system.
Numeric Accelerators	iPSC/2 system has VX Vector Processor SX Scalar Processor iPSC/860 has on-chip scalar and vector processing.
Cube Sharing	Multiple users can share the cube.
Node Memory Options	Memory available per board: 1M byte (iPSC/2 system only) 4M bytes (iPSC/2 system only) 8M bytes 16M bytes(iPSC/2 system only) 32M bytes 64M bytes
Cube Manager/SRM	Intel386 microprocessor based AT bus UNIX V.3
Host File System	Node programs have access to host file system. Standard C and Fortran calls can be made from node programs.
Network Access	Integrated Ethernet LAN controller with TCP/IP. Optional Network File System (NFS) support for SRM files.
Node File System	Optional Concurrent File System (CFS).

iPSC® SYSTEM SPECIFICATIONS C

SYSTEM SPECIFICATIONS

iPSC® SYSTEM	d3	d4	d5	d6	d7
Number of Nodes	8	16	32	64	128
Aggregate Memory (M bytes)					
Basic & SX[1] System					
1M byte per node[1]	8	16	32	64	128
4M bytes per node[1]	16	64	128	256	512
8M bytes per node	64	128	256	512	1024
16M bytes per node[1]	128	256	512	1024	—
VX System[1]					
1M bytes per node and 1M-byte VX	16	32	64	128	—
4M bytes per node and 1M-byte VX	40	80	160	320	—
8M bytes per node and 1M-byte VX	72	144	288	576	—

1. Applicable to iPSC/2 only.

iPSC®/860 RX COMPUTE NODE

Node Processor	Intel i860 microprocessor
Memory	8M-byte module, one wait state on write 8M-byte daughter card expansion
Communication	Direct-Connect Routing, implements a fully-connected network, variable length messages support peak data rates of 2.8M bytes per second on 8 bidirectional connections
Expansion Port	Full master/slave access to node
Indicators	Red, green, and amber
Operating System	NX/860, the Node eXecutive providing process management and message passing
Size	Eurocard 2x4 (9.2"x11")

iPSC®/2 CX COMPUTE NODE

Node Processor	Intel 386 microprocessor, native mode execution
Memory	1, 4, 8, 16M-byte modules, 64K Byte zero-wait-state cache 16M-byte module requires an adjacent slot
Communication	Direct-Connect Routing, implements a fully-connected network, variable length messages support peak data rates of 2.8M bytes per second on 8 bidirectional connections
Expansion Port	iLBX™-II interface to adjacent slot
Indicators	Red, green, and amber
Operating System	NX/2, the Node eXecutive providing process management and message passing
Size	Eurocard 2x4 (9.2"x11")

iPSC® SYSTEM I/O NODE

Node Processor	Intel386 microprocessor, native mode execution
Numeric Coprocessor	Intel387 arithmetic coprocessor 32-, 64-, 80-bit floating-point (IEEE 754)
Memory	4, 8, 16M-byte modules, 64K Byte zero-wait-state cache 16M-byte module requires an adjacent slot
Communication	Direct-Connect Routing, implements a fully-connected network, variable length messages support peak data rates of 2.8M bytes per second on 1 bidirectional connection
Expansion Port	PBX interface to adjacent slot
Indicators	Red, green, and amber
Operating System	NX/2, the Node eXecutive providing process management and message passing
Size	Eurocard 2x4 (9.2"x11")
I/O Port	Single-ended SCSI, 4M bytes per second peak transfer rate

SYSTEM RESOURCE MANAGER

Central Processing Unit	Intel 80386, native mode execution
Numeric Processing Unit	Intel 80387, arithmetic coprocessor 32-, 64-, 80-bit floating-point (IEEE 754)
Memory	8.5M bytes
Cube Communication	Direct-Connect Routing, 2.8M bytes per sec bandwidth RS 422 diagnostic channel
External Communication	Ethernet TCP/IP local area network port Optional Network File System (NFS) support
Peripherals	140/380M-byte hard disk 1.2M-byte 5-1/4" floppy 60M-byte 1/4" cartridge tape
Operating System	AT&T UNIX, Version V, Release 3.2, Version 2.1

ELECTRICAL AND ENVIRONMENTAL

Electrical

	Compact Cabinet Unit	SRM	Monitor/ Keyboard	Full-Size Cabinet Unit
AC Voltage	230 VAC ±15%	115/230 VAC ±10%	115/230 VAC ±10%	230 VAC ±15%
AC Current	16 amps	5/3 amps	0.5/0.25 amps	24 amps
Frequency	50/60 Hz ±5%	50/60 Hz ±5%	50/60 Hz ±5%	50/60 Hz ±5%
Power	3366 watts	314 watts	50 watts	5049 watts

Safety/RFI/EMI Standards System Is Designed To Meet

Compact Cabinet Unit	SRM	Monitor/ Keyboard	Full-Size Cabinet Unit
UL 478	UL 478	UL 478	UL 478
CSA C22.2 No. 154	CSA 22.2	CSA 22.2	CSA C22.2 No. 220
IEC 380	IEC 435	IEC 435	IEC 950
VDE 0806			EN60 950
VDE 0871 Class A	VDE 0871 Class A		VDE 0871 Class A
FCC 15 CFRJ Class A	FCC 15 CFRJ Class A	FCC 15 CFRJ Class B	FCC 15 CFRJ Class A
CRC C.1374	CRC C.1374	CRC C.1374	CRC C.1374

Environmental

	Compact Cabinet Unit	SRM	Monitor/ Keyboard	Full-Size Cabinet Unit
Temperature	10-35° C	17-32° C	5-40° C	10-35° C
Humidity	85%, maximum non-condensing	5-85%	10-90%	85%, maximum non-condensing
Altitude	0-8,000 ft	0-7,000 ft	0-8,000 ft	0-8,000 ft
Acoustical	55 dBA, max			55 dBA, max

Physical

	Compact Cabinet Unit	SRM	Monitor/ Keyboard	Full-Size Cabinet Unit
Dimensions	16"x16"x49"	22"x18"x6"	(monitor) 14"x15"x14" (keyboard) 2"x18"x8"	21-3/8"x25-1/2"x61-5/8"
Weight	215 lbs	50 lbs	(monitor) 21 lbs	200–450 lbs

TCP/IP SYSTEM CALLS D

Table D-1. Node TCP/IP System Calls (1 of 2)

accept(*s*, *addr*, *addrlen*)	**getservbyname**(*name*, *proto*) **getservbyport**(*port*, *proto*) **getservent**() **setservent**(*stayopen*) **endservent**()
bind(*s*, *name*, *namelen*)	**getsockname**(*s*, *name*, *namelen*)
connect(*s*, *name*, *namelen*)	**getsockopt**(*s*, *level*, *optname*, *optval*, *optlen*)
gethostbyaddr(*addr*, *len*, *type*) **gethostbyname**(*name*) **sethostent**(*stayopen*) **endhostent**()	**setsockopt**(*s*, *level*, *optname*, *optval*, *optlen*)
gethostname(*name*, *namelen*)	**htonl**(*hostlong*) **htons**(*hostshort*) **ntohl**(*netlong*) **ntohs**(*netshort*)
getnetbyaddr(*net*, *type*) **getnetbyname**(*name*) **getnetent**() **setnetent**(*stayopen*) **endnetent**()	**inet_addr**(*cp*) **inet_lnaof**(*in*) **inet_makeaddr**(*net*, *lna*) **inet_netof**(*in*) **inet_network**(*cp*)
getpeername(*s*, *name*, *namelen*)	**listen**(*s*, *backlog*)
getprotobyname(*name*) **getprotobynumber**(*proto*) **getprotoent**() **setprotoent**(*stayopen*) **endprotoent**()	

Table D-1. Node TCP/IP System Calls (2 of 2)

recv(*s, buf, len, flags*) **recvfrom**(*s, buf, len, flags, from, fromlen*) **select**(*nfds, readfds, writefds, exceptflds, timeout*) **send**(*s, msg, len, flags*) **sendto**(*s, msg, len, flags, to, tolen*) **shutdown**(*s, how*) **socket**(*domain, type, protocol*)	**Unique Node TCP/IP System Calls** **getiphosts**() **setiphost**(*iphost*)

INDEX

K

L

M

N

O

P