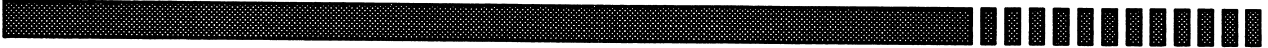
December 1991
Order Number: 311534-004



iPSC® SYSTEM SIMULATOR MANUAL

intel® Corporation

Copyright ©1991 by Intel Supercomputer Systems Division, Beaverton, Oregon. All rights reserved. No part of this work may be reproduced or copied in any form or by any means...graphic, electronic, or mechanical including photocopying, taping, or information storage and retrieval systems...without the express written consent of Intel Corporation. The information in this document is subject to change without notice.

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

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

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

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

286
287
4-SITE
Above
BITBUS
COMMputer
Concurrent File System
Concurrent Workbench
CREDIT
Data Pipeline
Direct-Connect Module
FASTPATH
GENIUS
i
i^{2}ICE
i386
i387
i486
i487
i860
ICE
iCEL
iCS
iDBP
iDIS
iLBX
im
Im
iMDDX
iMMX
Insite
int$_e$l
int$_e$lBOS
Intelevision
int$_e$ligent Identifier
int$_e$ligent Programming
Intel
Intel386
Intel387
Intel486
Intel487
Intellec
Intellink
iOSP
iPDS
iPSC
iRMX
iSBC
iSBX
iSDM
iSXM
KEPROM
Library Manager
MAP-NET
MCS
Megachassis
MICROMAINFRAME
MULTI CHANNEL
MULTIMODULE
ONCE
OpenNET
OTP
PC BUBBLE
Plug-A-Bubble
PROMPT
Promware
QUEST
QueX
Quick-Pulse Programming
Ripplemode
RMX/80
RUPI
Seamless
SLD
SugarCube
UPI
VLSiCEL

Ada is a registered trademark of the U.S. Government, Ada Joint Program Office
APSO is a service mark of Verdix Corporation
Ethernet is a registered trademark of XEROX Corporation
Excelan is a trademark of Excelan Corporation
EXOS is a trademark or equipment designator of Excelan Corporation
FORGE is a trademark of Pacific-Sierra Research Corporation
Green Hills Software, C-386, and FORTRAN-386 are trademarks of Green Hills Software, Inc.
GVAS is a trademark of Verdix Corporation
IBM and IBM/VS are registered trademarks of International Business Machines
Lucid and Lucid Common Lisp are trademarks of Lucid, Inc.
NFS is a trademark of Sun Microsystems
ParaSoft is a trademark of ParaSoft Corporation
Sun Microsystems and the combination of Sun and a numeric suffix are trademarks of Sun Microsystems
The X Window System is a trademark of Massachusetts Institute of Technology
UNIX is a trademark of AT&T
VADS and Verdix are registered trademarks of Verdix Corporation
VAST2 is a registered trademark of Pacific-Sierra Research Corporation
VMS and VAX are trademarks of Digital Equipment Corporation
VP/ix is a trademark of INTERACTIVE Systems Corporation and Phoenix Technologies, Ltd.
XENIX is a trademark of Microsoft Corporation

REV.	REVISION HISTORY	DATE
-004	Original Issue	12/91

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, 2200 Mission College Drive, Santa Clara, California 95052-8126.

Preface

Introduction

This manual describes the iPSC® Simulator. It is assumed that you are an application programmer proficient in the C and Fortran programming languages and the UNIX operating system. This manual provides you with enough detail to begin using your iPSC system.

The Simulator is a software program that mirrors the actions of the Intel iPSC system. It allows the development of iPSC programs in a controlled environment prior to execution on the actual iPSC system. Using the Simulator significantly accelerates the program development cycle and provides facilities for locating program errors, however, the Simulator will cause cube programs to run slower than they do on the actual nodes.

NOTE

In this manual, the term "iPSC system(s)" refers to any or all of the following SSD products: iPSC®/1, iPSC®/2, iPSC®/2S, iPSC®/860, and iPSC®/860S.

Organization

Chapter 1 "Introduction", provides an overview and guidelines for using the Simulator. This chapter also lists some differences between the Simulator and a real iPSC system.

Chapter 2 "Installing the Simulator", describes the procedure for installing the Simulator.

Chapter 3 "Preparing a Simulator Program", contains information about how to write, compile and link an application program for the Simulator.

Chapter 4	"Using the Simulator", contains a step-by-step procedure showing how to run one of the provided examples under the Simulator. A list of Simulator commands is provided.
Chapter 5	"The Simulator Commands", contains all of the Simulator's commands in manual page format.
Chapter 6	"Simulator Operation", explains how the Simulator is implemented. This is a source code description that can be the starting point for your own Simulator port.
Appendix A	"Commands, C System Calls and Fortran Routines" contains information about the NX/2 system calls and routines which can be used in writing parallel applications. This appendix contains a set of tables, by function, that lists a synopsis and a brief description of the calls and routines which are recognized by the Simulator.
Appendix B	"Error Messages" contains a list of Simulator error messages.

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

(NEW) *iPSC® System Technical Documentation Guide*
312026-002
Describes the technical documentation that supports the iPSC System and tells how to use the various documents.

(REV) *iPSC®/2 and iPSC®/860 C Commands and Routines Quick Reference*
311610-004
Summarizes all C routines and commands for the iPSC system.

iPSC®/2 and iPSC®/860 C Language Reference Manual
311567-004
Describes the Green Hills C compiler for the iPSC/2 and iPSC/860 systems.

(REV) *iPSC®/2 and iPSC®/860 Fortran Commands and Routines Quick Reference*
311615-004
Summarizes all Fortran routines and commands and for the iPSC system.

(REV) *iPSC®/2 and iPSC®/860 Programmer's Reference Manual*
311708-004
(Replaces 311071-003, 311019-003, and 311831-001)
Provides detailed information on all C and Fortran routines and commands for the iPSC system.

iPSC®/2 and iPSC®/860 User's Guide
311532-007
Overviews the iPSC system, including hardware and software architectures. Tells how to develop and run programs.

iPSC®/2 Fortran Language Reference Manual
311020-004
Describes the Green Hills Fortran compiler for the iPSC/2 system.

Intel® Manuals

UNIX Release R3.2 Manuals Literature Kit, UNXSYS386R3.2

Consists of the following documents:

(NEW) *UNIX System V Integrated Software Development Guide*
465274-001
Supplies information needed to write application software and installable drivers for new hardware additions for UNIX.

(NEW) *UNIX System V Introduction to UNIX System V*
465273-001
Introduces you to UNIX System V Release 3.2 on PC AT compatible computers using Intel386™ and Intel486™ microprocessors.

(NEW) *UNIX System V Network Programmer's Guide*
465282-001
Describes the UNIX System network programming environment, and provides detailed descriptions of programming tools.

(NEW) *UNIX System V Programmer's Guide, Vol. I*
465277-001
Describes the UNIX System programming environment, and provides detailed descriptions of programming tools.

(NEW) *UNIX System V Programmer's Guide, Vol. II*
465278-001
Describes the UNIX System programming environment, and provides detailed descriptions of programming tools.

(NEW) *UNIX System V Programmer's Reference Manual*
465276-001
Contains descriptions of commands, system calls, subroutines, libraries, file formats, macro packages, and character set tables.

(NEW) *UNIX System V Software Development Set*
465255-001
Provides UNIX System V Release 3.2 release notes.

(NEW) *UNIX System V Streams Primer*
465283-001
Introduces the UNIX System streams programming environment.

(NEW) *UNIX System V Streams Programmer's Guide*
465279-001
Describes the UNIX System streams programming environment, and provides detailed descriptions of programming tools.

(NEW) *UNIX System V System Administrator's Guide*
465280-001
Describes system maintenance tasks performed on the System Resource Manager under UNIX.

(NEW) *UNIX System V System Administrator's Reference Manual*
465281-001
Describes the UNIX System commands used by system administrators.

(NEW) *UNIX System V User's Guide*
465275-001
Provides a general description of UNIX.

Other Manuals

C: A Reference Manual - Harbison and Steele
480628-001
Describes the C programming language.

(NEW) *Effective Fortran 77* - Michael Metcalf
312201-001
Describes the Fortran 77 programming language.

The C Programming Language - Kernighan and Ritchie
122008-002
Describes the C programming language.

UNIX V - The Quick Reference Guide
311533-003
Summarizes UNIX commands, buzzwords, C shell hints and standard directory layout.Supercomputer Systems Division

Table of Contents

Chapter 1
Introduction

Overview 1-1

Differences Between an iPSC® System and the Simulator 1-2

Chapter 2
Installing the Simulator

Introduction 2-1

Installing the Files from the Source Media 2-1

Building and Installing the Executables and Library 2-3

Summary of Installed Files 2-3

Chapter 3
Preparing a Simulator Program

Introduction 3-1

Compiling and Linking Programs 3-1

Preparing UNIX V.3 or V.2 AT&T Programs 3-2

Preparing UNIX 4.2 BSD Programs 3-2

Preparing XENIX Programs 3-3

Chapter 4
Using the Simulator

A Simulator Example 4-1

Starting the Simulator 4-1

Interrupting the Simulator 4-2

Example of Using a Script File for Command Input 4-2

Commands 4-3

Walking Through a Simulator Session 4-4

Preparing the Example Source Code 4-4

Starting the Simulator 4-6

Loading a Program and Issuing Simulator Commands 4-7

Chapter 5
Simulator Commands

Introduction 5-1
ASIM, BSIM, XSIM 5-3
BOOTCUBE 5-4
CUBELOG 5-5
CUBEMAN 5-6
GETCUBE 5-7
HELP 5-8
KILLCUBE 5-9
LOAD 5-10
QUIT 5-11
RELCUBE 5-12
START 5-13
STARTCUBE 5-14
STATUS 5-15
SYSTEM 5-17
TRACE 5-18
WAITCUBE 5-20

Chapter 6
Simulator Operation

Introduction 6-1
Implementation Overview 6-1
Components of the Simulator 6-2
Requestor/Server Protocol Description 6-3
Description of the Modules 6-4
Interprocess Implementation of the NX/2 Library 6-7

Detailed Description of the Modules 6-10
General Definitions 6-10
Queue Management 6-10
Pipe Management 6-10
NX/2 Library 6-11
Fortran NX/2 Interface 6-12
Request Management 6-12
Process Management 6-15
Simulator 6-17

Appendix A
iPSC® System Commands, System Calls, and Routines

C System Call Summary A-1

Fortran System Call Summary A-8

Appendix B
Error Messages

Error Messages B-1

List of Illustrations

Figure 4-1. Examples Directory Tree 4-5
Figure 6-1. Access Hierarchy of Simulator Modules 6-6

List of Tables

Table 4-1. Summary of Simulator Commands 4-3
Table 5-1. Summary of Simulator Commands 5-1
Table A-1. C System Calls for Cube Control A-1
Table A-2. C System Calls for Message Passing A-3
Table A-3. C System Calls for Global Operations A-5
Table A-4. C System Calls for Miscellaneous Operations A-7
Table A-5. Fortran Routines for Cube Control A-8
Table A-6. Fortran Routines for Message Passing A-9
Table A-7. Fortran Routines for Global Operations A-12
Table A-8. Fortran Routines for Miscellaneous Operations A-15

Introduction 1

Overview

The Simulator package consists of a Simulator program and a set of libraries. The Simulator program is an interactive interface that simulates iPSC commands and system calls. It runs on either the UNIX V.3/V.2 AT&T Operating System, the UNIX 4.2 BSD Operating System or the XENIX Operating System. The Simulator libraries mirror the actions of the iPSC Node Executive (NX/2) routines.

The Simulator provides a vehicle for developing your iPSC application in a controlled environment prior to execution on an iPSC system. Although the Simulator gives you the flavor of a parallel system and helps you develop parallel prototypes, it does not give you true parallel performance. Expect your application to run much slower on the Simulator than on the actual nodes.

An iPSC application consists of a node program that runs on a group of allocated nodes and/or a host program that runs on either a local or remote host. Often the node program consists of just one process that runs in parallel on the nodes. A node program running under the Simulator does not run as a parallel program. The Simulator creates one process for each simulated node or host process, which execute sequentially.

Both C and Fortran languages are supported as on the iPSC system. Host programs are started within the Simulator and not from the UNIX or XENIX shell, as they would be with the actual System Resource Manager. Generally, programs executing on the Simulator can be transferred to the iPSC without modification to the source code. However, there are some routines which have no meaning in the simulated environment or are unsupported. Also, as there is only a single library supporting both host and node programs for the Simulator, a host application will succeed in calling a node-only system routine. This would fail if linked with the iPSC libraries. These types of differences are described in a subsequent section.

The Simulator has some built-in debugging capabilities to assist in developing programs. It performs interrupt handling enabling you to check the program status and system message queues and then continues from the point of interruption.

Differences Between an iPSC® System and the Simulator

Although the Simulator gives you the flavor of a parallel system and lets you perform some preliminary development, it is not intended to provide the power, speed, and flexibility of an iPSC system. The following list constitutes some of the differences between an iPSC system and the Simulator.

- The nodes are simulated as individual processes. There is no true parallel computation. The speed of execution is dramatically slower and the size of the simulation is operating system bound rather than hardware bound.

  The XENIX and UNIX operating systems limit the number of processes that the Simulator can create and thus limit the size of a simulation. The number of processes allowed varies between operating systems. For example, the XENIX operating system allows 14 processes to be created; the UNIX operating system allows 23 processes to be created.

  Thus, since a cube is allocated in powers of two, a simulated cube has a maximum of 16 nodes on a UNIX system and 8 nodes on a XENIX system.

- The Simulator does not support the NX/2 **handler()** system call. Although your program may link successfully, it will not run as expected if an exception occurs.

- The Simulator does not support the **plogon()** and **plogoff()** subroutines as well as the **dclock()** and **hwclock()** timing routines .

- The Simulator does not support the Concurrent File System™. If your program contains CFS calls, it will not successfully link.

- All timing uses a common time base. Because the simulated processes are executed in sequential time slices by the operating system, the values returned from **mclock()** are very different from those obtained on the actual nodes.

- In an iPSC system, different users can allocate separate cubes. With the Simulator, only one user can allocate a cube at a time.

- A simulated host program cannot parse the command line nor be used with UNIX pipes for input/output redirection. Once the Simulator has been invoked, subsequent host program execution is under Simulator control, and the host programs are not allowed to access command line arguments nor to use UNIX pipes to redirect standard input or output.

- File descriptors 9, 10, 11, and 12 are used by the Simulator and therefore cannot be used in applications.

- SIGUSR1 is reserved for use by the Simulator. Refer to */usr/include/sys/signal.h* for the specific number.

- The Simulator does not distinguish between host and node system calls. The iPSC system reserves some system calls for host programs and others for node programs. The Simulator does not catch host-only calls made by a node program or vice versa. Therefore it is possible to write programs for the Simulator that will not run on the iPSC system. On an iPSC system, the improper mixing of host and node calls will be displayed as unresolved references when host and node programs are linked with their respective libraries.
- The Simulator does not support the remote host. It assumes that host programs are running on the System Resource Manager.
- Compilers provided with XENIX systems may support the huge memory model. Programs compiled under the huge model will not run under the Simulator.

Installing the Simulator 2

Introduction

The Simulator product is shipped as a collection of source files which are then built and installed for the version of the operating system running on your computer. This chapter contains the instructions necessary to install the Simulator on any one of three different operating systems. A makefile is included to simplify the installation process.

Installing the Files from the Source Media

1. Login as *root.*

2. You can copy the Simulator files anywhere you desire. The installation instructions in this chapter assume that you loaded the Simulator files into a directory called */usr/ipsc/src/sim.* If you wish to follow along on a step-b -step installation process and you do not already have this directory, create it and make it your current directory. For example:

```
# mkdir /usr/ipsc/src/sim
# cd /usr/ipsc/src/sim
```

3. Now copy the tape or diskettes. The exact syntax of the command you use depends on your system. If your operating system is UNIX System V/386 Release 3.2, skip to instruction 3A. If your operating system is the UNIX System V/386 but its release is less than 3.2, skip to instruction 3B. If your operating system is the XENIX operating system, skip to instruction 3C. If your system cannot read the **tar** or **installpkg** format diskettes, skip to instruction 3D and use the magnetic tape.

   A. If you are running the UNIX System V/386 Release 3.2 operating system, use the high density UNIX diskettes in the **installpkg** format. Enter:

   ```
   # installpkg
   ```

Your system monitor will display the following:

```
Are you installing from tape (y/n)?
```

Enter ***n***.

Your system monitor will display the following:

```
If the program installation requires more than one floppy
disk, be sure to insert the disks in the proper order,
starting with disk number 1.

After the first floppy disk, instructions will be provided
for inserting the remaining floppy disks.

     Strike ENTER when ready
     or ESC to stop.
```

Insert the diskette and press **<Enter>**.

B. If you are running the UNIX System V/386 operating system, but its release is less than 3.2, use the two low density, XENIX **tar** format diskettes. Note that a UNIX V/386 operating system whose release is less than 3.2 uses XENIX diskettes. Insert "Disk 1 of 2". Then enter the command,

```
# tar xvf /dev/fdlo
```

When the diskette has been copied and the diskette access light has gone out, insert "Disk 2 of 2" and enter

```
# tar xvf /dev/fdlo
```

C. If you are running the XENIX operating system, use the two low density, XENIX **tar** format diskettes. Insert "Disk 1 of 2." Then enter the command,

```
# tar xvf /dev/dvf0
```

When the diskette has been copied and the diskette access light has gone out, insert "Disk 2 of 2" and enter

```
# tar xvf /dev/dvf0
```

D. Use the magnetic tape if your system cannot read the **tar** or **installpkg** format diskettes. Assuming that your tape device name is **rmt0**, enter the command,

```
# tar xvf /dev/rmt0
```

After the files are loaded, a message of "60 records in, 60 records out" is displayed indicating a successful load.

Building and Installing the Executables and Library

1. Create an executable version of the Simulator by issuing the **make** command. Choose the target appropriate for the operating system on which you will be using the Simulator.

# **make att52**	*UNIX System V Release 2.0 operating system*
# **make att53**	*UNIX System V Release 3.x operating system*
# **make xenix**	*XENIX operating system*
# **make bsd**	*Berkeley UNIX 4.2 operating system*

   The makefile also has the ability to provide definitions for CASSERT and DEBUG. These definitions conditionally compile certain debugging and assertion procedures into the Simulator code.

2. Now use the **make** command again to set the permissions, the owner, and the group, and place the files in the appropriate directories. The default is to put the executable in */usr/bin* and the library in */usr/lib*. On some systems, */usr* may be mounted as read-only. If that is true, you can either unmount */usr* and remount it as read-write, or you can choose to place the Simulator executable in a different directory. If you choose the latter, modify the macro definitions BINDIR and LIBDIR in the file *Makefile* prior to executing the following **make** command.

# **make instlatt**	*UNIX System V Release 2.0 operating system* *UNIX System V Release 3.x operating system*
# **make instlxenix**	*XENIX operating system*
# **make instlbsd**	*Berkeley UNIX 4.2 operating system*

Summary of Installed Files

After the Simulator has been installed, it consists of files in */usr/lib*, */usr/bin*, and */usr/ipsc/src/sim* directories.

Depending on the operating system chosen for installation, one of the following libraries is installed in */usr/lib*.

xsimlib.a	XENIX Simulator library for C and RM Fortran programs
bsimlib.a	UNIX 4.2BSD Simulator library for C and Fortran programs
asimlib.a	UNIX V.3/V.2 AT&T Simulator library for C and Fortran programs
rmfort_c.o	XENIX Simulator RM Fortran interface module

Depending on the operating system chosen for installation, one of the following executable files is installed in */usr/bin.*

xsim	XENIX Simulator program
bsim	UNIX 4.2BSD Simulator program
asim	UNIX V.3/V.2 AT&T Simulator program

The include files remain in the *src* directory, */usr/ipsc/src/sim.*

cube.h	include file for C language host and node programs
fcube.h	include file for Fortran language host and node programs

In addition, the following definition files are provided in the */usr/ipsc/src/sim* directory for iPSC/1 programs.

chost.def	definitions for C language host programs
cnode.def	definitions for C language node programs
rmfhost.def	definitions for RM Fortran host programs
rmfnode.def	definitions for RM Fortran node programs
fhost.def	definitions for UNIX Fortran host programs
fnode.def	definitions for UNIX Fortran node programs

NOTE

Only the source, include and definition files are supplied on the media. The others are created when you make and install the Simulator.

Preparing a Simulator Program 3

Introduction

Now that the Simulator is installed, you can begin to write your C and Fortran programs just as you would for an actual iPSC system. To avoid conflicts with the Simulator, adhere to the following programming conventions:

- Do not use XENIX/UNIX file descriptors 9, 10, 11, and 12. These descriptors are used by the Simulator library.
- Do not use SIGUSR1 because it is used by the Simulator. Refer to */usr/include/sys/signal.h* for the specific signal number.

Compiling and Linking Programs

This section describes how to compile and link programs for execution on the Simulator in the UNIX V.3/V.2 AT&T, XENIX, and UNIX 4.2BSD environments.

NOTE

In the following examples, and throughout this manual, it is assumed that you have installed the Simulator executable in */usr/bin* and the Simulator library in */usr/lib*.

Preparing UNIX V.3 or V.2 AT&T Programs

UNIX V.3/V.2 AT&T based C and Fortran programs are prepared by a simple compile and linking to the AT&T version of the Simulator Library. The following commands should be used when preparing C host and node programs for the Simulator. It is assumed in these examples that there is a single source file named *xxx.c*. If there are several modules in your application, add them in where the *xxx.c* and *xxx.o* appear. For example:

```
% cc -c xxx.c
% cc -o xxx xxx.o /usr/lib/asimlib.a
```

The following commands should be used when preparing Fortran host and node programs for the Simulator. It is assumed in these examples that there is a single source file named *xxx.f*. If there are several modules in your application, add them in where the *xxx.f* and *xxx.o* appear. For example:

```
% f77 -c xxx.f
% f77 -o xxx xxx.o /usr/lib/asimlib.a
```

Preparing UNIX 4.2 BSD Programs

UNIX 4.2 BSD based C and Fortran programs are prepared in a straightforward manner. The following commands should be used when preparing C host and node programs for the Simulator. It is assumed in these examples that there is a single source file named *xxx.c*. If there are several modules in your application, add them in where the *xxx.c* and *xxx.o* appear. For example:

```
% cc -c xxx.c
% cc -o xxx xxx.o /usr/lib/bsimlib.a
```

The following commands should be used when preparing Fortran host and node programs for the Simulator. It is assumed in these examples that there is a single source file named *xxx.f*. If there are several modules in your application, simply add them in where the *xxx.f* and *xxx.o* appear. For example:

```
% f77 -c xxx.f
% f77 -o xxx xxx.o /usr/lib/bsimlib.a
```

Preparing XENIX Programs

XENIX based C programs should use the large memory model (not the huge memory model), which is specified with the option **-Ml**. The following commands should be used when preparing C host and node programs for the Simulator. It is assumed in these examples that there is a single source file named *xxx.c*. If there are several modules in your application, add them in where the *xxx.c* and *xxx.o* appear.

```
% cc  -Ml  -c  xxx.c
% cc  -Ml  -o  xxx  xxx.o  /usr/lib/xsimlib.a  -lx -lf
```

If you are using math functions, then add the suffix **-lm**. Thus,

```
% cc  -Ml  -o  xxx  xxx.o /usr/lib/xsimlib.a  -lx -lf -lm
```

The following commands should be used when preparing Ryan-McFarland Fortran host and node programs for the Simulator. It is assumed in this example that there is a single source file named *xxx.f*. If there are several modules in your application, add them in where the *xxx.f* and *xxx.o* appear. For example:

```
% rmfort  xxx.f
% cc  -Ml  -o  xxx  xxx.o  /usr/lib/rmfort_c.o
  /usr/lib/xsimlib.a  -lx  -lf
```

When using Ryan-McFarland Fortran on a XENIX system, the integer parameters to the iPSC/1 calls are of type INTEGER*2. Because Ryan-McFarland Fortran integers default to INTEGER*4 when INTEGER is used, you must explicitly use INTEGER*2 in these calls.

Using the Simulator 4

A Simulator Example

Now that your application program has been compiled and linked, you will need to invoke the Simulator. This chapter describes the procedures used to start, stop and interrupt program execution using the Simulator.

To help explain how the Simulator can be used, an example of building a script file for command input along with a table of available commands is provided. Then, a step-by-step example of how to make, load, and run the Fortran pi example is shown.

Starting the Simulator

To invoke the Simulator enter one of the following commands, depending on your particular operating system.

`% asim`	*on the UNIX V.3/V.2 AT&T system*
`% bsim`	*on UNIX BSD systems*
`% xsim`	*on XENIX systems*

Interrupting the Simulator

When you send an interrupt signal during a simulation, the simulation pauses and the Simulator's prompt is displayed on the terminal. The interrupt key is usually the `<Del>` key, but on some systems the interrupt signal may be assigned to the `<Ctrl-C>` key. Once the simulation is interrupted, you can, for example, check on the Simulator's progress using the **status** command. To resume the simulation, use the **start** or **startcube** command.

NOTE

Be sure to send the interrupt signal when the program is running. You may not be able to restart the program if you send an interrupt signal while the program is waiting for screen input. To restart the program, use the **killcube** command to kill the process and then reload and restart the program.

Example of Using a Script File for Command Input

The Simulator's commands can be placed into a script file which is supplied to the Simulator's standard input. For example, the following commands could be placed into a script file called *runprog*:

```
getcube -t d2
load nodeprog
cubeman hostprog
trace on
start
quit
```

where *nodeprog* and *hostprog* are executable programs. This file would then be used as input to the Simulator, as follows:

```
asim < runprog > simout
```

Note that the output, including tracing output, is saved in the file *simout* in this example.

Commands

Most of the commands used in the Simulator mirror the commands that you type at the System Resource Manager's console to control the cube, however, a few commands are unique to the Simulator.

When using a command, either enter the name of the command or an abbreviation (the Simulator expects either one to be in lower case). Table 4-1 summarizes the Simulator commands.

Table 4-1. Summary of Simulator Commands

Command	Abbreviation	Description
bootcube	**bc**	simulates the **bootcube** command
cubelog	**log**	manages the Simulator's log file for the iPSC/1 interface
cubeman	**m**	loads programs into simulated System Resource Manager
getcube	**gc**	simulates the **getcube** command
help	**h or ?**	prints summary of Simulator commands and their use
killcube	**kc**	simulates **killcube** command
load	**ld**	simulates **load** command
quit	**q or exit**	terminates simulation and exits the Simulator
relcube	**rc**	simulates the **relcube** command
start	**s**	starts the simulation after cube and host processes are loaded
startcube	**sc**	simulates **startcube** command
status	**st**	checks the status of an application
system	**!**	passes commands through the Simulator to the shell
trace	**t**	enables or disables tracing of node operating system (NX) calls
waitcube	**wc**	simulates the **waitcube** command

Walking Through a Simulator Session

The Simulator includes a number of examples in the *examples* directory. The locations of the C and Fortran examples are illustrated in Figure 4-1.

In the following example, it is assumed that you have a UNIX V.3 system and that the Simulator software was installed in the */usr/ipsc/src/sim* directory. This example is a complete Simulator session, beginning with preparing the example source code, invoking the Simulator and running the Fortran pi example provided in the */usr/ipsc/src/sim/examples/f/pi* directory. The Fortran pi example calculates the value of pi by integrating $4/(1+x^2)$ from 0 to 1. Each node process performs a portion of the integral. The answer is the sum of the portions from each node.

Preparing the Example Source Code

Create your own directory to build the source file for the Fortran pi example. Copy the files from */usr/ipsc/src/sim/examples/f/pi* into that directory. For example:

```
% mkdir ~you/sim_ex
% cd ~you/sim_ex
% cp /usr/ipsc/src/sim/examples/f/pi/* .
```

A makefile is included to link in the library *asimlib.a* to both host and node programs. This makefile assumes a UNIX V.3/V.2 system. If you are not using a UNIX V.3/V.2 system, you must modify the makefile to contain the appropriate library. The following examples identify the library name modifications to the makefile for each type of operating system.

/usr/lib/asimlib.a	UNIX V.3/V.2 system
/usr/lib/bsimlib.a	UNIX BSD system
/usr/lib/xsimlib.a	XENIX system
/usr/lib/rmfort_c.o	XENIX system with Ryan-McFarland Fortran

Now issue the **make** command.

```
% make
        f77   -c host.f
host.f:
        f77   -c prompt.f
prompt.f:
        f77 -o host host.o prompt.o /usr/lib/asimlib.a

        f77 -o node node.f fx.f /usr/lib/asimlib.a

node.f:

fx.f:
```

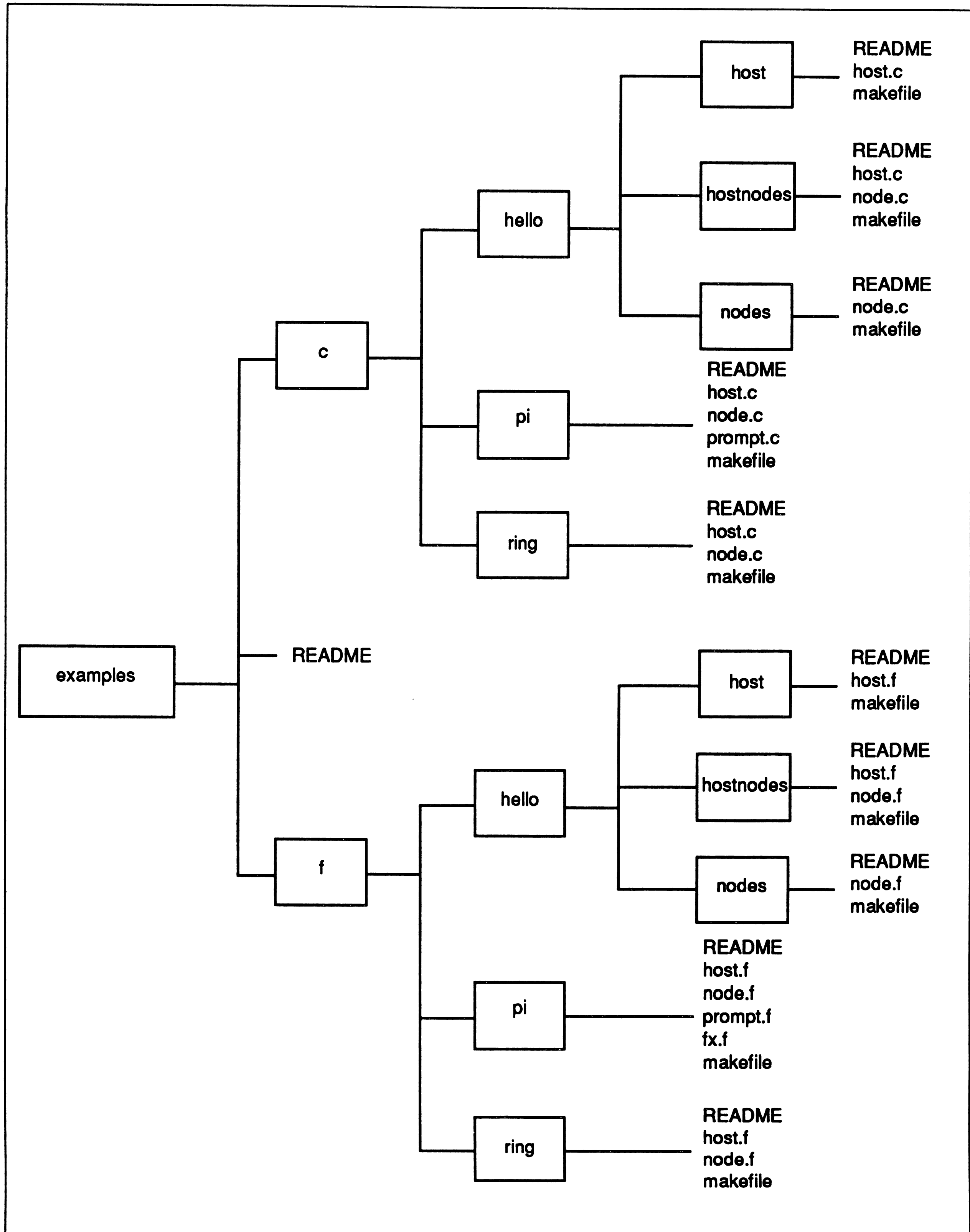


Figure 4-1. Examples Directory Tree

Starting the Simulator

Enter the name of the Simulator executable file. For a UNIX V.3/V.2 system, this is **asim**. The makefile installs the Simulator executable in */usr/bin*. This example assumes that this directory is in your PATH environment variable.

```
% asim
Welcome to the Intel iPSC/2 Simulator (Version 3.0).
Type help for assistance.
iPSC/2 sim>
```

The **help** command lists the available Simulator commands and their abbreviations.

```
iPSC/2 sim> help
The simulator has the following commands:

{bootcube | bc} [-d dim | -n nodes]
{cubeman | m} [-H] filename
{getcube | gc} [-t cubetype]
{help | h | ?}
{killcube | kc} [-p pid] [node ...]
{load | ld} [-p pid] [-H] [node ...] file
{quit | q | exit}
{relcube | rc}
{start | s}
{startcube | sc} [-p pid] [node ...]
{status | st} [-a] [-m] [-p] [-r] [-t]
{system | !} cmd parameters...
{trace | t} [on | off]
{waitcube | wc} [-p pid] [-f] [node ...]

In addition, you can hit the interrupt key to stop
the simulation and return to the command interpreter.
See the user's guide for more information.
```

Loading a Program and Issuing Simulator Commands

The Simulator assigns a default cube of 16 nodes (UNIX System) or 8 nodes (XENIX System). Use the **getcube** command to allocate a smaller cube. This example allocates four nodes:

```
iPSC/2 sim> getcube -t4
```

The **cubeman** command loads a host program and starts it running. The host part of the Fortran pi example contains a **load()** system call which loads the node program. The host program runs until that location in its code. You must then execute the Simulator **start** command to load and start the node program.

```
iPSC/2 sim> cubeman host
```

The **trace** command turns on tracing. When tracing is enabled, the Simulator prints a message on the screen whenever the program being simulated encounters an NX/2 system call. As you can see in the following example, tracing quickly generates a lot of data. Note that user input is shown in bold.

```
iPSC/2 sim> trace                                                User
checks state of trace
Tracing is off.

iPSC/2 sim> trace on                                             User
turns tracing on
Tracing was off.
iPSC/2 sim> start                                 User starts simulation
6/2/89 13:25 NODE: 4 UPID: 249 :setpid: for pid 100
 LOADING THE CUBE ...
6/2/89 13:25 NODE: 4 UPID: 249 :load: for process 0, node -1
6/2/89 13:25 NODE: 4 UPID: 249 : load (cont'd): path name is node
6/2/89 13:25 NODE: 4 UPID: 249 : numnodes

 ***************** 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)?
6/2/89 13:25 NODE: 3 UPID: 253 : myhost
6/2/89 13:25 NODE: 2 UPID: 252 : myhost
6/2/89 13:25 NODE: 1 UPID: 251 : myhost
6/2/89 13:25 NODE: 0 UPID: 250 : myhost
6/2/89 13:25 NODE: 3 UPID: 253 : mypid
6/2/89 13:25 NODE: 2 UPID: 252 : mypid
6/2/89 13:25 NODE: 1 UPID: 251 : mypid
6/2/89 13:25 NODE: 0 UPID: 250 : mypid

6/2/89 13:25 NODE: 3 UPID: 253 : mynode
6/2/89 13:25 NODE: 2 UPID: 252 : mynode
6/2/89 13:25 NODE: 1 UPID: 251 : mynode
6/2/89 13:25 NODE: 0 UPID: 250 : mynode
6/2/89 13:25 NODE: 3 UPID: 253 : *recv: for type=0
6/2/89 13:25 NODE: 3 UPID: 253 : *recv (cont'd): msg not found;
holding req
6/2/89 13:25 NODE: 2 UPID: 252 : *recv: for type=0
6/2/89 13:25 NODE: 2 UPID: 252 : *recv (cont'd): msg not found;
holding req
6/2/89 13:25 NODE: 1 UPID: 251 : *recv: for type=0
6/2/89 13:25 NODE: 1 UPID: 251 : *recv (cont'd): msg not found;
holding req
6/2/89 13:25 NODE: 0 UPID: 250 : *recv: for type=0
6/2/89 13:25 NODE: 0 UPID: 250 : *recv (cont'd): msg not found;
holding req
100                                                User requests 100 points
 What cube size (1 - 4) should I use?
4                                                  User requests cube size of 4
6/2/89 13:25 NODE: 4 UPID: 249 : mynode
6/2/89 13:25 NODE: 4 UPID: 249 : numnodes
6/2/89 13:25 NODE: 4 UPID: 249 : *send: for type=0 tnode=0 tpid=0

      .
      .
```

```
6/2/89 13:26 NODE: 0 UPID: 250 : mynode
6/2/89 13:26 NODE: 3 UPID: 253 : *recv: for type=0
6/2/89 13:26 NODE: 3 UPID: 253 : *recv (cont'd): msg not found;
holding req
6/2/89 13:26 NODE: 0 UPID: 250 : numnodes
6/2/89 13:26 NODE: 0 UPID: 250 : *send: for type=20 tnode=4 tpid=100
6/2/89 13:26 NODE: 0 UPID: 250 : *send (cont'd): deliver msg to
waiting receiver
  pi is approximately : 3.1416009869231245
6/2/89 13:26 NODE: 0 UPID: 250 : *recv: for type=0
6/2/89 13:26 NODE: 0 UPID: 250 : *recv (cont'd): msg not found;
holding req
  elapsed time =    0 min.  0.004 sec.
6/2/89 13:26 NODE: 4 UPID: 249 : numnodes

 ***************** 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                                         User input to quit simulation
 PLEASE WAIT WHILE I CLEAN OUT THE CUBE ...
6/2/89 13:26 NODE: 4 UPID: 249 : kill op: for node -1, process 0
sim: simulation done
```

To run the simulation again, load the host program again with the **cubeman** command. This time turn off the tracing, but interrupt the program while it is executing and look at the Simulator status with the **status** command.

To interrupt a simulating program, send an interrupt signal. This is usually the **<Del>** key, but sometimes it is **<Ctrl-C>**.

NOTE

Be sure to send the interrupt signal when the program is running, not when it is waiting for screen input. For example, if you sent the interrupt signal when the example was waiting for you to type the number of points or the number of nodes, you would not be able to restart it.

```
iPSC/2 sim> m host                                  User loads the host program

iPSC/2 sim> trace                                   User checks state of trace
Tracing is on.

iPSC/2 sim> trace off                               User disables tracing
Tracing was on.

iPSC/2 sim> start                                   User starts the simulation
 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)?
100000                                              User Input
 What cube size (1-  4) should I use ?
4                                                   User Input
<Del>                                               User sends an interrupt signal
sim: interrupt occurred
 iPSC/2 sim> status                                 User requests process status

Process status: (cube dimension = 2)

node     Unix pid        user pid        status          name
----     --------        --------        ------          ----
host     254             100             running         host
0        255             0               running         node
1        256             0               running         node
2        257             0               running         node
3        258             0               blocked         node

iPSC/2 sim> status -a                               User requests status tables

Process status: (cube dimension = 2)

node     Unix pid        user pid        status          name
----     --------        --------        ------          ----
host     254             100             running         host
0        255             0               running         node
1        256             0               running         node
2        257             0               running         node
3        258             0               blocked         node
```

```
Outstanding Messages:

target  target  source  source  type    length
 node    pid     node    pid
------  ------  ------  ------  ----    ------

Outstanding Receive Requests:

node    process type    buffer  Unix
          id            length  pid
----    ------- ----    ------  ----
3       0       5       20      258

Summary of Message Traffic:

node    # msgs out      # msgs in       # msgs recvd
----    ----------      ---------       ------------
host    15              1               1
0       3               6               6
1       2               6               6
2       2               6               6
3       2               5               5

Total # messages sent: 24
 Average distance for node to node messages: 1

iPSC/2 sim> start                                User re-starts the simulation
<Del>                                            User sends an interrupt
sim: interrupt occurred

iPSC/2 sim> status                               User requests process status

Process status: (cube dimension = 2)

node    Unix pid        user pid        status          name
----    --------        --------        ------          ----
host    254             100             blocked         host
0       255             0               blocked         node
1       256             0               blocked         node
2       257             0               running         node
3       258             0               running         node

iPSC/2 sim> start                                User re-starts the simulation
  pi is approximately : 3.1415926535981260
  elapsed time =    0 min.  0.056 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                                                          User Input
 PLEASE WAIT WHILE I CLEAN OUT THE CUBE ...
sim: simulation done

iPSC/2 sim> quit                                 User quits the simulation
%
```

Simulator Commands 5

Introduction

Most of the Simulator's commands mirror the commands that you use at the System Resource Manager's console to control the cube. However, a few commands are unique to the Simulator.

When using a command, enter either the name of the command or an abbreviation (the Simulator expects either one to be in lower case). Table 5-1 identifies the Simulator commands along with their abbreviations:

Table 5-1. Summary of Simulator Commands (1 of 2)

Command	Abbreviation	Description
bootcube	bc	simulates the **bootcube** command
cubelog	log	manages the Simulator's log file for the iPSC/1 interface
cubeman	m	loads programs into simulated System Resource Manager
getcube	gc	simulates the **getcube** command
help	h or ?	prints summary of Simulator commands and their use
killcube	kc	simulates **killcube** command
load	ld	simulates **load** command
quit	q or exit	terminates simulation and exits the Simulator
relcube	rc	simulates the **relcube** command

Table 5-1. Summary of Simulator Commands (2 of 2)

Command	Abbreviation	Description
start	s	starts the simulation after cube and host processes are loaded
startcube	sc	simulates **startcube** command
status	st	checks the status of an application.
system	!	passes commands through the Simulator to the shell
trace	t	enables or disables tracing of node operating system (NX) calls
waitcube	wc	simulates the **waitcube** command

ASIM, BSIM, XSIM — asim, bsim, xsim

These commands are used to invoke the Simulator. Use the **asim** command to invoke the Simulator on a UNIX V.3/V.2 AT&T system. Use the **bsim** command to invoke the Simulator on a UNIX BSD system. Use the **xsim** command to invoke the Simulator on a XENIX system.

Syntax

asim *<exec_file> output_file*
bsim
xsim

Arguments

exec_file — A file containing Simulator commands to be executed immediately upon invocation of the Simulator.

output_file — The log file to which all output from the Simulator is written (including the Simulator prompt and the application output.)

Description

If all output is sent to a logfile, there will be no start-up message or prompt displayed on the terminal to indicate when input (command or program) is required. This option is best used in conjunction with the *exec_file*. Be sure to end the *exec_file* with a **quit** command when sending to an output file, or else the Simulator may appear to hang.

Examples

1. Invoke the Simulator on a UNIX V.3/V.2 AT&T system causing it to execute all the commands in the file command_file and saving the output in the file sim-log.

```
% asim < command_file > sim_log
```

2. Invoke the Simulator on a UNIX BSD system causing it to execute some commands to start-up the Simulator session:

```
% bsim < startup_file
```

BOOTCUBE bc

This command simulates the **bootcube** command and is used to reset the nodes. It removes all outstanding processes, messages, and receive requests.

Syntax

bootcube [**-d** *dim* | **-n** *nodes*]

Arguments

-d *dim* Sets the maximum size of the cube. A cube with dimension 2 has four nodes. A cube with dimension 3 has eight nodes, etc.

-n *nodes* Sets the maximum size of the cube. *nodes* is the number of nodes.

Description

You can set the size of the cube, either by specifying the dimension or the number of nodes. The Simulator will accept the other switches from the iPSC bootcube, but they are ignored.

Examples

1. Load the NX/2 operating system into entire cube:

```
iPSC/2 sim> # bootcube
```

2. Reset and load NX/2 into a 8-node cube:

```
iPSC/2 sim> # bootcube -n8
```

3. Load an alternative operating system:

```
iPSC/2 sim> # bootcube -K /usr/joelc/my_os
```

CUBELOG

log

This command simulates the cube managers **cubelog** command and is used to manage the Simulator's log file. This command is used for the iPSC/1 interface.

Syntax

cubelog [-n] [-l*log_filename*| -lstdout |-l]

Arguments

-n Specifies that the system log file is not to be used any longer. Events can be logged again to the default log file by typing **cubelog** without any arguments.

-l*log_filename* Specifies that the Simulator is to maintain a log in the specified log file.

-lstdout The Simulator supports a special log file named *stdout* which causes all logging output to be directed to the standard output, which is, by default, sent to the user's terminal.

If no log file name is supplied, then the default log file, called *LOGFILE*, is used.

Description

Simulated processes use the log file name that is in effect at the time they are loaded.

CUBEMAN m

This command loads programs into the simulated System Resource Manager.

Syntax

cubeman [**-H**] *object_file*

Arguments

-H Wait for a **start** command to start the process; do not start the process immediately.

In either case, with or without **-H**, the actual simulation will not run until the **start** command is entered. If the **-H** option is not used, the process will block at the first call to a node library routine. The **-H** option is very useful for interactive host programs. By delaying the execution of the host (via **-H**) until the simulation is started, you avoid the race condition of both the Simulator interface and host program competing for input from the terminal (*stdin*).

object_file pathname of the executable program to be loaded. Arguments cannot be supplied to *object_file* within the **cubeman** command as they would be from the shell in the actual System Resource Manager.

Description

This command can be executed multiple times to load more than one program into the simulated System Resource Manager. With or without the **-H** option, node simulation will start only after a **start** command has been entered. Until then, no node library routines can be executed by the host process.

Examples

```
iPSC/2 sim> cubeman host
```

GETCUBE

gc

This command simulates the **getcube** command and is used to allocate a cube.

Syntax

getcube [-t *cubetype*]

Arguments

-t *cubetype* — Allows you to specify the size and type of the cube. Type is meaningless in the Simulator and will be ignored. Size can be either:

dn	where **d** is dimension and *n* is the size of the dimension (for example, d3)
n	where *n* is the number of nodes when no dimension is given (for example, 8)

Description

If no size is given, the largest size is assigned. If you start a simulation without issuing a getcube, the Simulator allocates a cube with the maximum number of nodes.

Examples

1. Allocate the first available cube of any size and type, and name it "defaultname":

```
iPSC/2 sim> getcube
```

2. Allocate a four-node cube:

```
iPSC/2 sim> getcube - t4
```

HELP h or ?

This command prints a summary of the Simulator's commands and their use in order to give you quick assistance.

Syntax

help

Examples

```
iPSC/2 sim> help
```

KILLCUBE kc

This command simulates the **killcube** command and lets you terminate node processes running on your simulated nodes.

Syntax

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

Arguments

-p *pid* — *pid* is the process id of the process you want to kill. This is the pid assigned by the Simulator, not the UNIX/XENIX pid. If **-p** is not used, all processes will be killed.

node... — the number(s) of the node(s) in which processes are to be killed. If no node number is specified, all *pid* processes on all nodes are killed.

Description

If no parameters are supplied, **killcube** kills all simulated node and host processes and cleans out all messages awaiting delivery. The **-c** switch, for specifying cube name, is accepted but ignored.

The **killcube** command has the same function as the iPSC/1 **loadkill** command. Use **killcube** to terminate node processes and flush messages related to those processes.

Before loading another application (with the **load** command) or releasing the cube (with the **relcube** command), you should use **killcube** to ensure that no processes remain active and all message buffers are empty.

Examples

1. Kill all processes on all nodes in the cube named "alpha."

```
iPSC/2 sim> killcube
```

2. Kill all processes on the first three nodes of the current cube.

```
iPSC/2 sim> killcube 0 1 2
```

LOAD ld

This command simulates the **load** command. It loads programs onto the simulated nodes.

Syntax

load [**-p** *pid*] [**-H**] [*node...*] *file*

Arguments

-p *pid* — *pid* is the process id to be assigned to the new process being loaded. If **-p** is not used, *pid* will be zero.

-H — Wait for a **startcube** or **start** command to start the process; do not start the process immediately.

In either case, with or without **-H**, the actual simulation will not run until the **start** or **startcube** is entered. If the **-H** option is not used, the process will block at the first NX/2 system call.

node... — Specifies the numbers of the node(s) to be loaded. If no node number is specified, all nodes in the simulated cube are loaded.

file — Specifies the pathname of the file to be loaded into the simulated nodes.

Description

This command may be executed multiple times to load more than one program into the simulated nodes. The Simulator will ignore any other arguments. Note that processes loaded with the **load** command must have unique process ids in every node.

Examples

1. The following example loads the file */usr/tom/proc3* on all nodes of the current cube:

```
iPSC/2 sim> load /usr/tom/proc3
```

2. The following example loads the file named *node_pgm* (which requires the two arguments "input_file" and "1000") on nodes 3 and 4 of the cube:

```
iPSC/2 sim> load 3 4 node_pgm input_file 1000
```

QUIT **q** or **exit**

This command terminates the simulation and exits the Simulator. The Simulator removes all node and host processes before exiting.

Syntax

quit

Examples

```
iPSC/2 sim> quit
```

RELCUBE rc

This command simulates the **relcube** command and is used to release currently allocated cubes.

Syntax

relcube

Description

Use **relcube** to release one or all cubes that you own. Switches are accepted but ignored.

Examples

- Release the current cube:

```
iPSC/2 sim> relcube
```

- Release all cubes owned by you:

```
iPSC/2 sim> relcube -a
```

START
S

This command is used to start the simulation after all cube and host processes have been loaded with **cubeman** or **load.**

Syntax

start

Description

The actual XENIX/UNIX processes that were previously loaded by **cubeman** or **load** using the **-H** option are created only after **start** is invoked. In this case, if too many processes have been previously loaded, you will be notified at this time.

This command terminates when either all simulated processes have terminated or the interrupt key is pressed at the terminal.

STARTCUBE sc

This command simulates the **startcube** command and is used to start processes which have been loaded with **load -H** and starts the simulation running.

Syntax

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

Arguments

-p *pid*	*pid* is the process to be started. If **-p** is not used, all processes will be started.
node...	the number(s) of the node(s) in which processes are to be started.

Description

This command is used to start processes that were loaded with the **load** command using the **-H** switch. It has no effect on processes that are already running.

If too many processes have been previously loaded with **load -H**, you will be notified at this time.

Note that **startcube** does not start processes loaded with **cubeman**. Switches are accepted but ignored.

This command terminates when either all simulated processes have terminated or the interrupt key is pressed at the terminal.

The **startcube** command has the same function as the iPSC/1 **loadstart** command.

Examples

1. Start all processes (on the current attached cube) loaded with the **-H** argument.

```
iPSC/2 sim> startcube
```

2. Start process 3 on nodes 5 and 6 in the cube.

```
iPSC/2 sim> startcube -p 3 5 6
```

STATUS st

This command lets you check the status of your application.

Syntax

status [-a] [-m] [-p] [-r] [-t]

Arguments

-a All status tables are displayed.

-m Display only message status.

-p Display only process status.

-r Display only request status.

-t Display only message traffic status.

Description

If no option is supplied, then only the process status is displayed. A common use is first to load the simulated nodes and start your process running, then to press the interrupt key and enter the **status** command. At this point, you can peruse the various status tables that are available.

Examples

```
iPSC/2 sim> status -a                                    User requests status tables

Process status: (cube dimension = 2)

node    Unix pid        user pid        status          name
----    --------        --------        ------          ----
host    254             100             running         host
0       255             0               running         node
1       256             0               running         node
2       257             0               running         node
3       258             0               blocked         node
```

STATUS *(cont.)* st *(cont.)*

```
Outstanding Messages:

target  target  source  source  type    length
 node    pid     node    pid
------  ------  ------  ------  ----    ------

Outstanding Receive Requests:

node    process type    buffer  Unix
          id            length  pid
----    ------- ----    ------  ----
3       0       5       20      258

Summary of Message Traffic:

node    # msgs out      # msgs in       # msgs recvd
----    ----------      ---------       ------------
host    15              1               1
0       3               6               6
1       2               6               6
2       2               6               6
3       2               5               5

Total # messages sent: 24
 Average distance for node to node messages: 1

iPSC/2 sim> status
```

User requests process status

```
Process status: (cube dimension = 2)

node    Unix pid        user pid        status          name
----    --------        --------        ------          ----
host    254             100             blocked         host
0       255             0               blocked         node
1       256             0               blocked         node
2       257             0               running         node
3       258             0               running         node
```

SYSTEM !

This command lets you pass commands to the shell.

Syntax

system *cmd parameters*

Arguments

cmd parameters Use any command allowed in the UNIX/XENIX shell.

Description

Note that the abbreviated command **!** can be used with or without a space between the **!** and the command.

TRACE t

This command enables or disables the tracing of NX/2 library calls.

Syntax

trace [on | off]

Arguments

on	Turns tracing on.
off	Turns tracing off.

Description

With no arguments, the command reports whether tracing is currently enabled or disabled. When enabled, the Simulator prints a message on the screen when a process executes an NX/2 command. This helps you trace the progress of the simulation and locate bugs.

Examples

Trace messages are generated when tracing is enabled. The format for these messages is

```
mm/dd/yy: hh:mm NODE: nn UPID: uu : <string>
```

where:

mm/dd/yy	is the current date.
hh:mm	is the current time.
nn	is the node that generated the trace message.
uu	is the UNIX process that generated the trace message.
<string>	is one of the message strings.

TRACE *(cont.)* — t *(cont.)*

In the trace messages, an asterisk (*) indicates that the message applies to all commands with the given base name. For example:

*recv refers to **crecv()**, **hrecv()**, and **irecv()**

*send refers to the **send()**, **sendw()**, **sendmsg()**, **csend()**, **isend()**, and **hsend()** commands. The symbol <*int*> or <*long*> indicates that a short or long integer value is placed in the message at this location.

iPSC®/1 Trace Messages

```
"send*: for chan=<int> type=<int> tnode=<int> tpid=<int>"
"send* (cont'd): deliver msg to waiting receiver"
"send* (cont'd): receiver not waiting; msg enqueued"
"recv*: for chan=<int> type= int>"
"recv*: (cont'd): msg found from node=<int>, pid=<int> of
type=<int>"
"recv* (cont'd): msg not found; holding req"
"status: for channel <int>"
"cubedim"
"mynode"
"probe: for chan=<int> type=<int>"
"probe: msg of length <int> found for chan=<int> type=<int>"
"probemsg: for pid=<int>"
"probemsg: msg of length <int> found for pid=<int>"
"copen: for process <int>"
"cclose: for channel <int>"
"clock"
"load: for process <int>, node <int>"
"load (cont'd): path name is <string>"
"kill op: for node <int>, process <int>"
"wait op: for node <int>, process <int>"
```

WAITCUBE wc

This command simulates the **waitcube** command, which starts the simulated node processes and waits for a process to complete.

Syntax

waitcube [**-p** *pid*] [**-f**] [*node...*]

Arguments

-p *pid*	*pid* is the process id of the specific process to wait for. If **-p** is not used, **waitcube** waits for any process.
-f	Causes **waitcube** to return when the first process that meets the requirements specified by the other switches is completed. The default is to wait for all specified processes.
node...	the number(s) of the node(s) in which processes are to be found. If no node number is specified, **waitcube** waits for the specified process in any node.

Description

Switches are accepted but ignored.

If you use **waitcube** with no arguments, it will wait for a process to complete anywhere in the simulated cube. After each process completes, **waitcube** prints the following message:

```
Node xx PID yy completed, code xx (no meaning)
```

Unlike the real system, the return code for **waitcube** in the message is always zero and has no significance.

The **waitcube** command has the same function as the iPSC/1 **loadwait** command.

Examples

1. Wait for all processes on all nodes of attached cube to be completed:

```
iPSC/2 sim> waitcube
```

Simulator Operation 6

Introduction

This chapter describes the implementation of the iPSC System Simulator and assists you in understanding the Simulator's source code so that you can modify it or port it to another environment. You should be familiar with the C programming language, the system calls and subroutines of the UNIX operating system and the iPSC System Node Executive (NX/2).

The Simulator also supports iPSC/1 calls which are shown in this chapter as ***bold italic***.

Implementation Overview

The Simulator is implemented as a set of modules. The specification for each module is in the file *module_name.h*, where *module_name* specifies the module's name, and its implementation is in the file *module_name.c*. For example, the Pipe Management module has the module name *pipemgt*; its specification is in *pipemgt.h*, and its implementation is in *pipemgt.c*.

The Simulator implements each simulated node and host process as a separate UNIX/XENIX process (hereafter referred to as just a UNIX process) and communicates with these processes using pipes and signals. A protocol is used to share a fixed number of pipes. The simulator process mediates all message passing between the simulated nodes and host processes. By centralizing all interprocess iPSC System communication functions in one UNIX process, debugging and tracing information are easily obtained. Also, only a fixed number of UNIX pipes are required, and the only limitation to the size of a simulation is the number of UNIX processes that can be spawned. This limitation differs for various UNIX systems.

For conciseness, the simulated node and host processes are referred to as *requestors*, and the simulator process is referred to as the *server*.

Components of the Simulator

The Simulator consists of the following source files:

commutl.h	specification for communication utilities
commutl.c	implementation of communication utilities
cube.h	include file for C language host and node programs
fcube.h	include file for Fortran language host and node programs
flib.c	implementation of UNIX Fortran and Green Hills Fortran NX/2 interface
gendef.h	specification for general definitions
hslib.h	specification for C version of NX/2 library
hslib.c	implementation of NX/2 library
pipemgt.h	specification for pipe management
pipemgt.c	implementation of pipe management
pcsmgt.h	specification for process management
pcsmgt.c	implementation of process management
qmgt.h	specification for queue management
qmgt.c	implementation of queue management
reqmgt.h	specification for request management
reqmgt.c	implementation of request management
rmfort_c.f	implementation of RM/Fortran NX/2 interface on the XENIX operating system
sim.c	implementation of the Simulator
simerr.h	Simulator error list
version.h	specification for version management
version.c	implementation of version management
xrmflib.c	implementation of C part of RM/Fortran NX/2 interface on the XENIX OS
Makefile	make file for the Simulator

In addition, the Simulator includes the following set of definition files for iPSC/1 programs:

chost.def	definitions for C language host programs
cnode.def	definitions for C language node programs
fhost.def	definitions for Fortran language host programs
fnode.def	definitions for Fortran language node programs
rmfhost.def	definitions for RM Fortran language host programs
rmfnode.def	definitions for RM Fortran language node programs

Requestor/Server Protocol Description

All NX/2 library calls are implemented as remote procedure calls to the server using the UNIX pipes for communication. This section describes the protocol by which the requestors communicate with the server to implement the NX/2 subprograms. This protocol assures that only one simulated node or host process uses the UNIX pipes at a time and that the pipes are left in a consistent state for other processes.

In this protocol, the server and requestor exchange messages over the pipes. The protocol implements the NX/2 commands which allow you to pass messages between processes. The messages you set up in your application are referred to as ***user messages*** in this chapter. This distinguishes them from the underlying messages of the implementation.

Three UNIX pipes are used for communication.

- A ***request pipe*** into which the simulated node and host processes place requests.
- A ***data in*** pipe with which the simulated processes send messages to the server.
- A ***data out*** pipe with which the server delivers replies and messages to the simulated processes.

In addition, a UNIX signal called SIG_REPLY is used to wake up requestors who are blocked awaiting service.

The protocol between the requestors and the server proceeds as follows:

1. The requestors construct request messages and write them one at a time into the request pipe using the UNIX **write()** call. All request messages have a common message header, and parameters associated with the requests are included in the request messages. The requestors block awaiting the SIG_REPLY signal from the server once their requests have been written.

2. The server repeatedly reads request messages from the request pipe and processes the requests. The server first reads a fixed size request header. The contents of the request header, namely the requested operation, indicate the size of the complete request message and, thus, the number of bytes remaining to be read.

3. The server replies to the requestor with a reply message that is particular to the requested operation. The requestor and server exchange messages on the data pipes according to the requested operation.

4. The requestor terminates the transaction by writing a final message on the *data out* pipe. The server awaits this message before reading the next request. This ensures that the data pipes are always empty when the server accepts a new request.

It is important to note that the server replies to all requests that supply user parameters (for example, **msgdone()**); that is, tacit replies are avoided. This guarantees that the requestor does not terminate itself and possibly the simulation before the server is done.

The server can also initiate a transaction to one of the requestors. This mechanism is used to deliver messages to processes. The protocol proceeds as follows:

1. Between the serving of requests, the server writes a specific message into the *data out* pipe and then issues SIG_REPLY to the requestor.

2. The requestor reads in the server's message and identifies that it is a server message and not a reply to a request. The requestor then engages the server in an exchange of messages in the same manner as the above protocol.

Description of the Modules

The directed acyclic graph shown in Figure 6-1 shows the access relationships between modules. Starting from the bottom of the access hierarchy, the modules comprising the Simulator are briefly described as follows:

1. **General Definitions.** This module contains generally useful definitions that are used by the other modules. Its module name is *gendef.*

2. **Queue Management.** This module provides general purpose queueing functions. Its module name is *qmgt.*

3. **Pipe Management.** This module manages the reading and writing of UNIX pipes that connect the server and the requestors. It provides primitives for sending large messages across the pipes in small packets. It defines the message formats used in the requestor/server protocol. Its module name is *pipemgt.*

4. **NX/2 Library**. This module provides the NX/2 subprogram library used by the requestors. This library implements the requestor's side of the requestor/server protocol, and it accesses the Pipe Management module in order to communicate with the server. The specification portion is provided in the file *cube.h* for consistency with other node software. The module's implementation file name is *hslib.c*. The *commutl.c* contains applications utilities which are part of the library.

5. **Fortran NX/2 Interface**. This module provides a Fortran interface to the above module. The specification portion is provided in the file *fcube.h* for consistency with other node software. Its UNIX BSD and Green Hills Fortran / UNIX AT&T implementations are the same and provided in the file *flib.c*. Its XENIX implementation is found in the files *xrmflib.c* and *rmfort_c.f* for Ryan-McFarland Fortran.

6. **Request Management**. This module is part of the server and manages requests for NX/2 services from the requestors; it implements the server's side of the requestor/server protocol. It also buffers messages that are awaiting delivery. It accesses Pipe Management in order to communicate with the requestors. Its module name is *reqmgt*.

7. **Process Management**. This module manages the creation and destruction of the requestor processes. Its module name is *pcsmgt*.

8. **Simulator**. This module implements the user interface to the Simulator and manages the simulation. It initializes the other modules and directly accesses Request Management and Process Management. Its module name is *sim*.

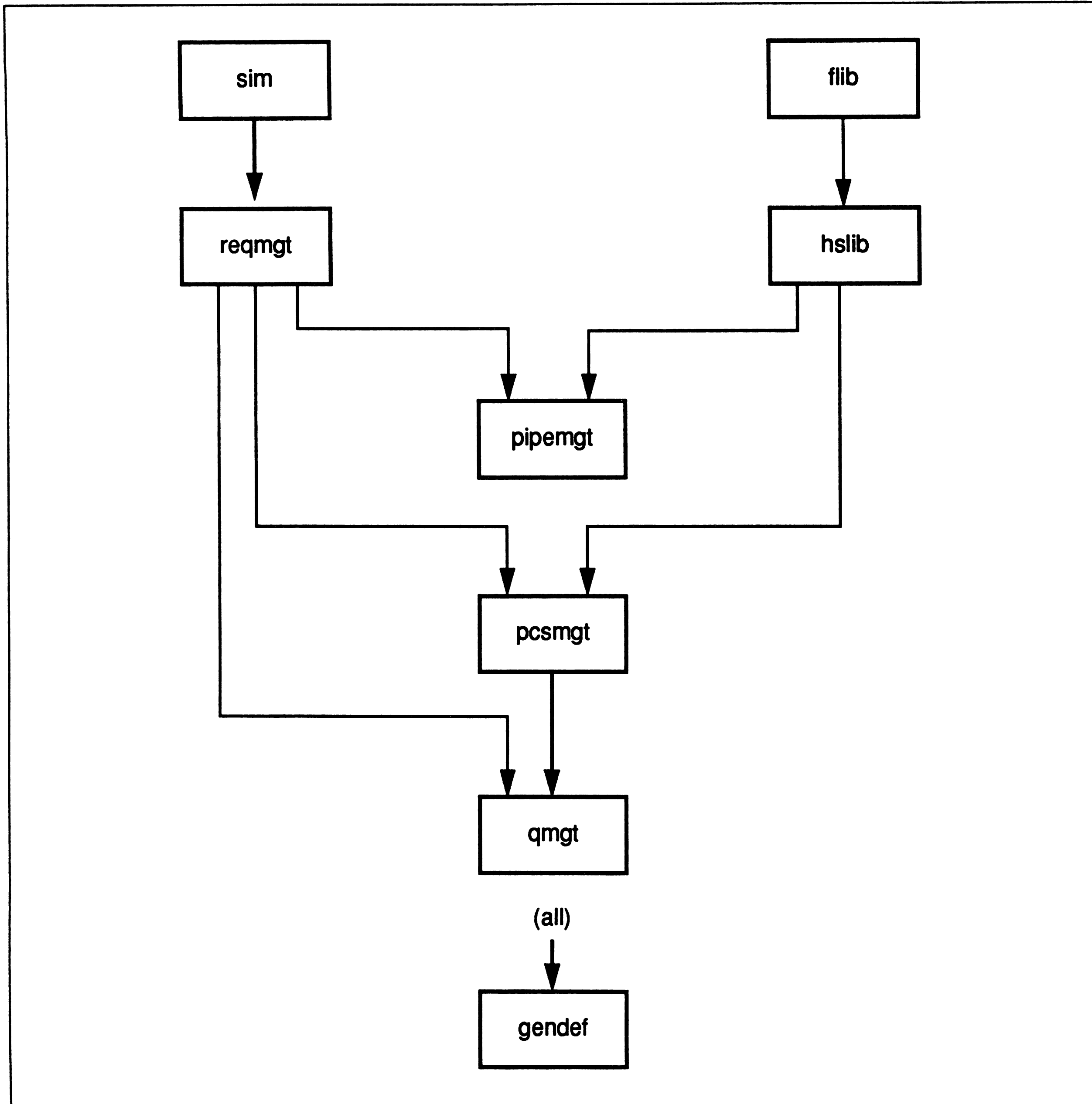


Figure 6-1. Access Hierarchy of Simulator Modules

Interprocess Implementation of the NX/2 Library

This section describes how the NX/2 library calls are implemented using the requestor/server protocol. All NX/2 calls except **syslog()** are implemented as requests to the server. The exact exchange of messages is described below for each NX/2 call; request and reply codes as defined in Pipe Management are indicated. All requests are issued using request messages containing the actual parameters of the call. Illegal parameters supplied by the requestor are treated as fatal errors and terminate the simulation. All iPSC/1 calls are shown in ***bold italics***.

1. **csend()**, **isend()**, **hsend()**, ***send()***, ***sendw()***, ***sendmsg()***. Upon receiving a SEND_REQ request, the server replies with a positive acknowledgment in a SEND_REPLY message. In this implementation, the server never rejects send requests. The requestor then sends the user's message to the server folloSupercomputer Systems Divisionwed by an acknowledgment message. Reception of the acknowledgment message terminates the transaction with the requestor. Note that all six versions of send are implemented almost in the same manner.

   If the server determines that another requestor is awaiting the message from a previous receive call, then the server issues a MSG_DELIVERY or ASYNC_MSG_DELIVERY message to the receiver depending on whether or not the requestor is blocked awaiting the message. This message contains the parameters of the earlier receive request (including the pointers supplied by the user); note that the requestor need not locally maintain this information. The server then sends the user's message to be delivered and awaits an acknowledgment.

   In case the call is **csend()** or **isend()**, the server saves the information to be used by the **info...()** calls. Also, if the receiver request was made by **hrecv()**, the server delivers the signal to execute the user-defined handler routine if the mask is clear. If the mask is set, the server saves the information and a pointer to the user handler routine in PCSDATA until the mask is cleared. If a receiver is not available, the server then buffers the message for later delivery. Also, if the call is **csend()** or **isend()**, the server searches for pending probe requests to awaken its requestor, if available. If the call is **hsend()** and the mask is clear, it executes the user handler routine provided by the user; otherwise, it saves the information until the mask is cleared.

2. **crecv()**, **hrecv()**, **irecv()**, ***recv()***, ***recvw()***, ***recvmsg()***. Upon receiving a RECV_REQ request, the server determines whether a user message is available for delivery to the requestor. It replies with a RECV_REPLY message that indicates if a user message is available and its parameters. The server then sends the user message to the requestor, if available, and awaits acknowledgment from the requestor. If the call is **crecv()** or **irecv()**, information is saved for use by **info...()** calls.

   If the call is **hrecv()** and the mask is clear, the server delivers the signal to execute the user handler routine if a message has arrived to satisfy the request. If the mask is set then the server saves the information and the pointer to the user-specified handler routine in the PCSDATA until the mask is cleared.

For efficiency purposes in ***recvw()*** or ***recv()*** or ***recvmsg()***, the requestors locally note when a channel is busy awaiting the reception of a message. If the user attempts to use a busy channel or if a ***recvw()*** or ***recvmsg()*** call is made, then the requestor blocks awaiting the delivery of a user message for the appropriate channel. This local information avoids the need to issue a request to determine this information.

3. **csendrecv(), hsendrecv(), isendrecv()**. These are implemented by calling a combination of NX/2 send and receive requests.

4. **masktrap()**. Upon receiving a MASKTRAPS_OP request, the server determines whether a mask should be set or cleared. If the mask is cleared, the server delivers the pending signals. In either case, the requestor sends an acknowledgment to the server that terminates the transaction.

5. **numnodes(), msgdone(), myhost(), info...(), iprobe(), status(), cubedim(), mynode(),** ***probe(), probemsg(), copen(), cclose()***. These calls are issued using their specific request codes; note that some calls share message formats with other calls. The server determines the appropriate reply information, issues a reply message, and awaits acknowledgment.

6. **setpid()**. Upon receiving a SETPID_OP request, the server sets the host pid.

7. **cprobe()**. Upon receiving a CPROBE_OP request, the server determines if a message with the selected type is available. Positive or negative indication is returned to the requestor depending upon whether the message is available or not. In either case, the requestor sends an acknowledgment to the server that terminates the transaction.

   If the message is not available, the requestor blocks execution and the server records the request. Whenever a message arrives, a CPROBE_COMPLETE message is sent to the requestor; this message unblocks the requestor.

8. **msgwait()**. Upon receiving a MSGWAIT_OP request, the server determines if an **irecv()** with specific transaction ID is completed. Positive or negative indication is returned to the requestor depending upon whether the transaction is completed or not. In either case, the requestor sends an acknowledgment to the server that terminates the transaction.

   If the transaction is not completed, the requestor blocks execution and the server records the request. Whenever the transaction is completed (that is, a message has arrived), a MSG_DELIVERY message is sent to the requestor; this message unblocks the requestor.

9. **msgcancel()**. Upon receiving a MSGCANCEL_OP request, the server cancels the specified transaction by setting its channel status to CH_IDLE. The requestor then sends an acknowledgment to the server which terminates the transaction.

10. **load()**. Upon receiving a LOAD_REQ request, the server determines the length of the string specifying the program name to be executed and replies to the requestor. After receiving a reply, the requestor then sends the program name and an acknowledgment to the server. The server starts a new UNIX process to execute this program.

11. **waitone()**, ***lwait()***. Upon receiving an LWAIT_REQ request, the server checks to see if the specified process has completed execution. If so, the completion data are sent to the requestor. Otherwise, a negative indication is returned to the requestor. In either case, the requestor sends an acknowledgment to the server which terminates the transaction.

    If the awaited process has not completed execution, the requestor blocks execution and the server records the request. Whenever a process completes execution, the list of pending ***lwait()*** requests is checked. If a request can be satisfied, an LWAIT_COMPLETE message is sent to the (blocked) requestor; this unblocks the requestor and returns the completion data.

12. ***lwaitall()***. The server handles an LWAITALL_REQ request in the same manner as it handles an LWAIT_REQ request, except that it checks for the completion of all selected processes.

13. **killcube()**, **killproc()**, **flushmsg()**, ***lkill()***. Upon receiving an LKILL_REQ, the server determines whether it is an ***lkill()***, **killcube()**, **killproc()**, or **flushmsg()** request. If it is an ***lkill()*** or **killcube()**, the server kills the specified process(es) and cleans up the cube. If it is a **killproc()**, the server kills the specified process(es). If it is a **flushmsg()** request, the server flushes specified messages for the specified process(es). The server then replies to the requestor, which returns an acknowledgment. As the UNIX processes exit, pending ***lwait()*** requests are checked.

In addition, the Simulator supports the Communication Utility calls such as **gopf()**, **gray()**, **ginv()**, and **gsync()**. For the complete list, refer to the *iPSC®/2 and iPSC®/860 Programmer's Reference Manual*. These calls are implemented at the library level.

There are two versions of every request call, one of which is distinguished by a leading underscore, for example, **csend()** and **_csend()**. These versions determine how errors are handled. In processing requests, the server checks for user errors. If an error occurs, the server informs the requestor by sending an error code. If the request call does not have a leading underscore, the implementation prints an appropriate error message. It then initiates a BRKTRANS_REQ transaction to inform the server that an error has occurred; this causes the server to interrupt execution and return control to the user. The user process then exits. Otherwise (in the case that the request call has a leading underscore), the implementation returns `-1` to the caller and continues processing requests.

Detailed Description of the Modules

This section provides a detailed description of the Simulator's modules.

General Definitions

This module contains useful definitions. The type STATUS is used by many subprograms to indicate to their callers whether or not the subprogram executed successfully. Several normal return codes and error codes are defined. As a general rule, errors are propagated upwards to the point at which an error message and corrective action are meaningful to the user.

This module also contains two definitions of procedures which are useful in debugging the Simulator. The ASSERT procedure tests the value of a supplied predicate; if the predicate evaluates to FALSE, then the supplied message is printed and the program terminates. Assertions are placed in key places in the code to trap situations that indicate the presence of a bug. These procedures can be conditionally compiled into the code by providing a definition for CASSERT.

The PRINTMSG procedure prints a message to the standard output. It is used to trace execution of the Simulator's procedures and to print out values for arguments and variables. It is conditionally compiled into the code by supplying a definition for DEBUG.

Queue Management

This module provides general purpose queueing functions using doubly linked lists, which allow for easy insertion and deletion of queue items. All structures to be queued are required to have an entry of type LINK as the first entry of the structure. The module provides routines to initialize links and queues before their initial use.

Pipe Management

This module provides facilities of the requestor/server protocol that are shared by both requestors and servers.

This module defines the file descriptors for the pipes used in the protocol which are determined when the pipes are opened. The requestors' file descriptors are represented as manifest constants since they must be accessible by the requestors when they are spawned via the UNIX **fork()** and **exec()** calls. The *Init_reqsvr()* routine of this module opens the pipes and sets up the file descriptors.

The module also defines the message formats which are exchanged in the requestor/server protocol. All messages start with a code field that identifies the message's type to its receiver. The remainder of the message formats are devoted to the parameters specific to the type. All message codes are defined here as manifest constants.

Primitives are provided for conveniently sending and receiving data over the pipes. These primitives are described in the module's specification.

NX/2 Library

This module provides the user with the standard NX/2 calls and is linked to the user's programs. For the iPSC®/860 interface, calls are defined in the *iPSC®/2 and iPSC®/860 Programmer's Reference Manual.*

As described earlier in this chapter, the NX/2 calls are implemented as remote procedure calls to the server. Facilities required by the requestor are implemented in this module. Since requests must be written in the request pipe using a single UNIX **write()** call (for atomicity between requestors), data structures are provided internally which combine the general request header (of type PREQUEST) with each of the specific request messages. Also, a signal handling routine called *Handle_psig()* is provided within the implementation. This routine is invoked whenever the SIG_REPLY signal is received from the server. It distinguishes between replies to requests and the delivery of user messages based on the code field in the message.

There is a subtle timing relationship between the above signal handler and the main code of the requestor process. This relationship must be properly understood to avoid deadlocks in the requestor. The code in this module, which implements the requestor's portion of the NX/2 calls, blocks the process when it is awaiting:

- a reply to a request.
- the arrival of a message within a ***recvw()***, ***recvmsg()***, **crecv()**, or **csendrecv()**.
- the arrival of a message within a **msgwait()**.
- the arrival of a message within a **cprobe()**.
- the completion of a process within an ***lwait()***, ***lwaitall()***, **waitone()**, or **waitall()**.
- the availability of a busy channel within a send or receive call for the iPSC/1 interface.

Blocking is achieved using a standard UNIX call. The signal handler re-awakens a blocked process upon the reception of the SIG_REPLY signal from the server. A potential race condition exists between the main code and the signal handler since signal handling is asynchronous to the main code. The integer *rpy_code* and the boolean *pcs_waiting* are used to avoid this race condition and thus ensure that the main code never misses a signal and blocks indefinitely. The use of these variables and the placement of the assignments is critical to this function.

Fortran NX/2 Interface

This module provides an interface to the Simulator's NX/2 library for Fortran programs. The implementation differs for the UNIX BSD or UNIX AT&T and XENIX versions due to the differing conventions in these two environments. The UNIX BSD or UNIX AT&T version consists of a C language file (*flib.c*) which is called by the user's Fortran program. This module matches the calling conventions of Fortran to those of C. The XENIX version requires a C language file (*xrmflib.c*) and an additional Fortran file (*rmfort_c.f*) to match the calling conventions of the two languages. The user calls the Fortran file, which in turn calls the C file.

The XENIX version of this module, supporting the Ryan-McFarland Fortran compiler, is generated using Ryan-McFarland's Fortran compiler. This module (*xrmflib.c* and *rmfort_c.f*) and the above NX/2 library are linked to the user's Fortran program using **ld**.

Request Management

This module implements the server's portion of the requestor/server protocol (as driven by an execution loop in the Simulator module). It provides a routine for the module's initialization, called *Init_chan()*. It also provides a routine, called *Handle_req()*, which handles all of the various requests from the requestors, as determined from the code field in the general request header. The requested functions are the NX/2 library calls, such as **mynode()**, **isend()**, and **crecv()**. The Simulator module reads the general request header from the request pipe and then calls this routine to read and handle the remainder of the request.

The implementation of this module is based on four global data structures as follow:

Each channel in the simulated nodes is represented by a structure of type CHANDATA. This structure indicates the state of the channel and is used to manage channels awaiting user messages. The Simulator maintains a fixed number of channel structures, which are allocated to channels when they are requested. A channel is either open or closed; an open channel is either idle or receiving.

- Idle. The channel is not in use.
- Receiving. A receive call has been issued and a message has not yet arrived.

The channel structure maintains this state information and is queued onto one of the two queues according to its state.

- *freeq*. This is the queue of closed (unallocated) channels;
- *waitq*. This is the queue of receiving channels. One of these queues exists for each node.

Idle channels are not queued.

The second data structure, of type MSGBUF, holds parameters for user messages pending delivery and the actual messages themselves. A message buffer is created when the server receives a user message from a requestor, and the buffer is removed when the message is delivered. It is placed in the queue for its destination node (called *rmsgq*).

The next data structure, of type PPROBE, holds pending requests to the NX/2 **cprobe()** call. Each request has a type, which selects a message to wait for. These **cprobe()** requests are queued on the queue *pprobeq*. Whenever a message arrives, the list is scanned to see if the message matches one of the selected messages. If so, then the blocked process is informed, and the **cprobe()** request is discarded.

The final data structure, of type PLWAITREQ, holds pending (that is, unsatisfied) requests to the **waitone()**, **waitall()**, ***lwait()***, and ***lwaitall()*** calls. Each request has a node and process ID, which select one or more processes on one or more nodes that the requesting process awaits. The request also indicates which process is blocked awaiting the completion of the request. These requests are queued on the queue *lwaitq*. Whenever a process completes, the list is scanned to see if the completed process matches one of the selected processes. If there is a match, then the blocked process is informed, and the request is discarded.

The Request Management module's subprograms implement the NX/2 library calls using the above data structures. A service routine is provided for each library call, and its name corresponds to that of the library call prefixed with *Req_* (for example, *Req_send()*). These routines are implemented as follows:

- The routine *Req_copen()* handles ***copen()*** requests. It removes a channel structure from the *freeq* and initializes its state.

- The routine *Req_send()* handles all versions of send requests. It allocates a message buffer and buffers the user's message. If it finds a waiting ***recv()*** request at the destination node's *waitq*, then it delivers the message to the receiver and de-allocates the message buffer. If the receiver request was made by **hrecv()** call and the mask is clear, it delivers the signal to execute the user handler routine. But if the mask is set, it saves the information and the pointer to the user-specified handler routine in the PCSDATA until the mask is cleared. If there is no pending ***recv()***, then it looks for pending **cprobe()** requests if the request is for **csend()** or **isend()**. If it finds a waiting **cprobe()**, it unblocks the waiting process and saves the information. Otherwise it enqueues the message buffer at the destination node's *rmsgq*.

- The routine *Req_recv()* handles all versions of ***recv()*** requests. If it finds a message in the requestor's *msgq*, then it delivers the message and saves the information for the **info...()** calls if the request is for **crecv()** or **irecv()**. If the request is for the **hrecv()** call and the mask is clear, it delivers the signal to execute the user handler routine if the message has arrived to match the request. If the mask is set, it saves the information and the pointer to the user specified handler routine in the PCSDATA until the mask is cleared. If the message is not available, it places the requestor's parameters in the specified channel structure and enqueues the channel in the requestor's *waitq*.

- The *Req_masktraps()* routine sets a new mask and delivers the pending signals if the mask is cleared.

- The *Req_cclose()* routine places the specified channel back on the *freeq* and resets its state.
- The *Req_status()* routine handles **status(), cubedim(), mynode(), myhost(), numnodes(), infotype(), infocount(), infonode(), infopid(), msgdone(),** and **mypid()** requests. It simply reads the data structures to determine requested information.
- The *Req_probe()* routine handles probe requests by reading the data structures to determine the requested information.
- The *Req_clock()* routine handles ***clock()*** and **mclock()** requests by obtaining the current system time from the UNIX or XENIX operating systems. To keep the values for time as small as possible, they are referenced to the system time when the Simulator is started. Note that all timing uses a common time base. Because the simulated processes are executed in sequential time slices by the UNIX operating system, the value from ***clock()*** or **mclock()** are very different from those obtained on the actual nodes.
- The *Req_setpid()* routine sets the pid for the host process.
- The *Req_ciprobe()* routine handles **cprobe()** and **iprobe()** requests by reading the data structures to determine the request information. It checks whether a message of a given type has arrived. If the request is for **cprobe()** and the message has not arrived, it builds a pending **probe()** request (of type PPROBE) and enqueues it on the *pprobeq* queue.
- The *Req_msgwait()* routine handles **msgwait()** requests. It determines if the requested transaction is completed. If not, it blocks the process.
- The *Req_msgcancel()* routine handles **msgcancel()** requests. It cancels the specified transaction.
- The routine *Req_load()* handles **load()** requests. It checks that legal process and node numbers are supplied and that the supplied process ID is unique. After receiving the path name for the process to be executed, this routine calls the *Cubeload()* routine within the Process Management module to create and start the process.
- The *Req_lkill()* routine handles **killcube(), killproc(), flushmsg(),** and ***lkill()*** requests by calling the *Cubekill()* routine in Process management. The cleanup behind a killed process is performed when the Simulator receives the UNIX signal that the process has exited.
- The *Req_lwait()* routine handles **waitone(), waitall(),** ***lwait()***, and ***lwaitall()*** requests by calling the *Check_completed()* routine in Process Management to determine if one of the specified processes has exited. As explained below, Process Management maintains a list of completed processes. If a process has completed, then the node, process ID, and return code are returned to the requestor. Otherwise, a negative indication is returned, and the requestor blocks. In this case, the server builds a pending ***lwait()*** request (of type PLWAITREQ) and enqueues it on the *lwaitq* queue. When the specified process completes at a later time, the requestor is signaled and unblocked by means of the queued ***lwait()*** request.

The implementation of **waitall()** or ***lwaitall()*** incrementally awaits completed processes. That is, when a completed process is encountered, its process data are removed by the routine *Check_completed()* even if another selected process is subsequently found to be active.

In addition to the above routines, Request Management provides a routine called *Check_lwaitreq()* for checking the status of pending ***lwait()*** requests. This routine is called (by the Simulator module) whenever a child process completes execution. It determines whether the completed process was awaited by another process and, if so, it signals the waiting process.

Process Management

This module provides the functions needed to create and destroy simulated node and host processes. It also maintains state information for these processes.

The implementation of this module is based on the state information for each process, which is recorded in a data structure of type PCSDATA. The state information includes process identifiers and the process's current status, as explained below. This structure is linked into the queue *hs_pdata* for the node associated with the process (numbered from one to the maximum number of nodes); host processes are queued on the zeroth queue.

Node processes have three process IDs associated with them.

- The process ID assigned by the Simulator when it spawns the process as a result of a **load()** call or the **load** command. This is often referred to as the specified process ID, abbreviated ***spid***, in the source code.
- The process ID assigned when the process opens a communications channel with ***copen()***. This ID is known only to the Request Management module; it is not known to Process Management.
- The UNIX process ID for the Simulator's invocation of the process. These process IDs are only relevant to the Simulator.

The first type of process ID is supplied for the cube processes when a node process is loaded. The host process uses **setpid()** to set its assigned process ID.

The Simulator assigns a process ID of -1 (named NULLSPID in the source code) to a host program loaded with **cubeman**. If you load another host program, it also gets an assigned pid of -1. If your host program forks a child, that child also has an assigned pid of -1. Assigned process IDs of -1 are not forced to be unique.

State information is maintained for each process and is encoded in the process's UNIX pid and the boolean variable *pd_active* within the process data. A process is in one of the following states:

- Spawned by the user but not started, as indicated by a null UNIX process ID and *pd_active* set to TRUE.
- Started, as indicated by a non-null UNIX process ID and *pd_active* set to TRUE.

- Completed execution and available as the target of an *lwait()* call, as indicated by a non-null UNIX process ID and *pd_active* set to FALSE.

User processes are often referred to by their node number and specified process ID. Many of the routines in Process Management use this notation. All the processes in one node can be referred to as a group by supplying -1 (the constant NULLSPID) for the process ID. Likewise, a specific process in all nodes can be referred to by supplying -1 (the constant NULLNODE) for the node number. Finally, all processes can be referred to by supplying -1 for both parameters.

In PCSDATA, a pointer is provided for extended data, which is used by *reqmgt* to save information for the **info...()** calls. *Get_extdatap()* and *Set_extdatap()* routines are provided to access the extended data pointer.

The routines *Cubeload()* and *Cubestart()* are used to create and start simulated node and host processes. The first routine creates state information for the specified process(es) and optionally starts the associated UNIX process(es). In either case, the boolean *pd_active* is set to TRUE for the newly created process(es). The second routine starts the specified process(es), whose state information was previously created with *Cubeload()*.

Note that the interrupt signal is temporarily ignored when a process is started so that the newly started process will ignore interrupts. This prevents simulated processes from inadvertently terminating when the user attempts to interrupt the simulation.

The routine *Cubekill()* kills the specified process(es) and marks them completed by setting the boolean *pd_active* to FALSE. The state information for the killed process(es) can be optionally retained in case the completion of the process(es) is subsequently awaited by another process.

Processes that exit normally are marked as completed using the routine *Mark_completed()*; this routine is intended to be called whenever a process exits. It determines which processes have completed and so marks them. This is accomplished by sending to the process the UNIX signal denoted SIG_TESTP in the module. This signal is set to a UNIX signal which is most likely not to be used by a program (the SIG_CHILD signal was selected in this implementation). If the process does not in fact exist, then the UNIX system call **signal()** returns an error.

The *Check_completed()* routine checks to see if one of the specified process(es) has completed. If such a process is found, then its process data are removed, and the completion information is returned to the caller of *Check_completed()*.

The routines *Alloc_pdata()* and *Remove_pdata()* respectively allocate and remove state information for processes. The first call is needed to make the Simulator aware of processes that are spawned using the UNIX **fork()** call by the user's simulated host processes. The second call is used within Process Management to remove state information for completed processes.

Simulator

This module provides the main program for the Simulator. Since it is not referenced by another module, it has no specification file. The main program first initializes the other modules and then enters the main loop, which repeatedly reads and executes the user's commands. The Simulator exits when an end-of-file condition exists on the standard input.

This module uses the Process Management module to create and destroy simulated host and node processes. When a **start, startcube**, or **waitcube** command is issued, the module calls the routine *Run_sim()*. This routine enters an inner loop which drives the server portion of the requestor/server protocol. The loop repeatedly reads requests from the request pipe and then invokes the *Handle_req()* call in the Request Management module. This routine exits either when a single simulated process exits or when all simulated processes have exited, as selected by its boolean parameter. The first use allows **waitcube** to check for completion after each child process exits. The second use terminates execution of the commands **start** and **startcube.**

The *Run_sim()* routine checks for the completion of the user's pending **waitone(), waitall(),** ***lwait(),*** and ***lwaitall()*** requests whenever a child process exits. It does this by first calling *Mark_completed()* in the Process Management module to mark all completed processes. It then calls *Check_lwaitreq()* in Request Management to check the pending requests. This implementation was chosen to simplify signal handling at the cost of extra execution time in this relatively infrequent case.

This module provides a signal handler for interrupts. This routine allows the user to interrupt the execution of a simulation and return control to the user interface. It also provides a handler to catch the UNIX signal (denoted SIG_CHILD in the module) which indicates that a child process has terminated.

As described above for the NX/2 library module, there is a subtle timing relationship between the signal handlers and the main line code. In this case, care is taken to avoid missing a signal when blocking to read a request from the request pipe or the user interface. This is accomplished using the booleans *child_exited, interrupt*, and *pcs_waiting*. The use of these variables and the placement of the assignments is critical to this function.

iPSC® System Commands, System Calls, and Routines

A

This appendix contains summaries of the iPSC system calls and routines which can be used by applications being run on the Simulator. The C and Fortran system call summaries are in separate sections.

C System Call Summary

This section summarizes the C versions of the system calls. See the *iPSC®/2 and iPSC®/860 Programmer's Reference Manual* for more information on these calls.

Table A-1. C System Calls for Cube Control (1 of 2)

Call	Synopsis	Environment	Description
attachcube()	**attachcube**(*cubename*) char **cubename*;	Host	Attach to a cube and make it the current cube.
cubeinfo()	long **cubeinfo**(*ct, numslots, global*) struct cubetable **ct*; long *numslots, global*;	Host	Obtain information about allocated cubes.
flick()	**flick()**	Host, Node	On a node, relinquish the CPU to another process. On the host, is a no-op.
getcube()	**getcube**(*cubename, cubetype, srmname, keep*) char **cubename, *cubetype, *srmname*; long *keep*;	Host	Allocate a cube.
killcube()	**killcube**(*node, pid*) long *node, pid*;	Host, Node	Terminate and clear node process(es).
killproc()	**killproc**(*node, pid*) long *node, pid*;	Host, Node	Terminate a node process.

Table A-1. C System Calls for Cube Control (2 of 2)

Call	Synopsis	Environment	Description
killsyslog()	**killsyslog()**	Host	Terminate *syslog* process.
load()	**load**(*filename*, *node*, *pid*) char **filename*; long *node*, *pid*;	Host, Node	Load a node process.
myhost()	long **myhost()**	Host, Node	Obtain node ID of host machine.
mynode()	long **mynode()**	Host, Node	Obtain node ID of calling process.
mypid()	long **mypid()**	Host, Node	Obtain NX/2 process ID of calling process.
newserver()	**newserver**(*cubename*) char **cubename*;	Host	Start new file server for specified cube.
nodedim()	long **nodedim()**	Host, Node	Obtain dimension of current cube.
numnodes()	long **numnodes()**	Host, Node	Obtain number of nodes in current cube.
relcube()	**relcube**(*cubename*) char **cubename*;	Host	Release specified cube.
setpid()	**setpid**(*pid*) long *pid*;	Host	Set NX/2 process ID for host program.
setsyslog()	**setsyslog**(*stdfd*) long *stdfd*;	Host	Start *syslog* process.
waitall()	**waitall**(*node*, *pid*) long *node*, *pid*;	Host, Node	Wait for all specified processes to complete.
waitone()	**waitone**(*node*, *pid*, *cnode*, *cpid*, *ccode*) long *node*, *pid*; long **cnode*, **cpid*, **ccode*;	Host, Node	Wait for specified processes to complete.

Table A-2. C System Calls for Message Passing (1 of 2)

Call	Synopsis	Environment	Description
cprobe()	**cprobe**(*typesel*) long *typesel*;	Host, Node	Wait for a message to arrive. Blocks calling process.
crecv()	**crecv**(*typesel*, *buf*, *len*) long *typesel*; char **buf*; long *len*;	Host, Node	Receive a message, blocks until receipt.
csend()	**csend**(*type*, *buf*, *len*, *node*, *pid*) long *type*; char **buf*; long *len*, *node*, *pid*;	Host, Node	Send a message, and wait for completion.
csendrecv()	long **csendrecv**(*type*, *sbuf*, *slen*, *tonode*, *topid*, *typesel*, *rbuf*, *rlen*) long *type*; char **sbuf*; long *slen*, *tonode*, *topid*, *typesel*; char **rbuf*; long *rlen*;	Host, Node	Simultaneously, send a message, and Wait for completion.
flushmsg()	**flushmsg**(*typesel*, *node*, *pid*) long *typesel*, *node*, *pid*;	Host, Node	Flush specified messages from the system.
hrecv()	**hrecv**(*typesel*, *buf*, *len*, *name*) long *typesel*; char **buf*; long *len*; void(**name*)(); *name*(*type*, *count*, *node*, *pid*) long *type*, *count*, *node*, *pid*;	Node	Post a receive; when complete, invoke a user-written handler.
hsend()	**hsend**(*type*, *buf*, *len*, *node*, *pid*, *name*) long *type*; char **buf*; long *len*, *node*, *pid*; void(**name*)(); *name*(*type*, *count*, *node*, *pid*) long *type*, *count*, *node*, *pid*;	Node	Send a message and set up a handler procedure to be called when the reply arrives.

Table A-2. C System Calls for Message Passing (2 of 2)

Call	Synopsis	Environment	Description
hsendrecv()	**hsendrecv**(*type*, *sbuf*, *slen*, *tonode*, *topid*, *typesel*, *rbuf*, *rlen*, *name*) long *type*; char **sbuf*; long *slen*, *tonode*, *topid*, *typesel*; char **rbuf*; long *rlen*; void(**name*)(); *name*(*type*, *count*, *node*, *pid*) long *type*, *count*, *node*, *pid*;	Node	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()**	long **infocount()** long **infonode()** long **infopid()** long **infotype()**	Host, Node	Return information about a pending or received message.
iprobe()	long **iprobe**(*typesel*) long *typesel*;	Host, Node	Determine whether a message of a selected type is pending.
irecv()	long **irecv**(*typesel*, *buf*, *len*) long *typesel*; char **buf*; long *len*;	Host, Node	Post a request to receive a message.
isend()	long **isend**(*type*, *buf*, *len*, *node*, *pid*) long *type*; char **buf*; long *len*, *node*, *pid*;	Host, Node	Send a non-blocking message.
isendrecv()	long **isendrecv**(*type*, *sbuf*, *slen*, *tonode*, *topid*, *typesel*, *rbuf*, *rlen*) long *type*; char **sbuf*; long *slen*, *tonode*, *topid*, *typesel*; char **rbuf*; long *rlen*;	Host, Node	Simultaneously, send a message, and post a receive for the reply. Do not wait for completion.
masktrap()	long **masktrap**(*state*) long *state*;	Node	Enable or disable send/receive trap.
msgcancel()	**msgcancel**(*id*) long *id*;	Host, Node	Cancel an asynchronous send or receive operation.
msgdone()	long **msgdone**(*id*) long *id*;	Host, Node	Determine if an asynchronous send or receive is complete.
msgwait()	**msgwait**(*id*) long *id*;	Host, Node	Wait for completion of an asynchronous send or receive.

Table A-3. C System Calls for Global Operations (1 of 3)

Call	Synopsis	Environment	Description
gcol()	**gcol**(*x*, *xlen*, *y*, *ylen*, *ncnt*) char *x*[]; long *xlen*; char *y*[]; long *ylen*; long **ncnt*;	Node	Global concatenation operation.
gcolx()	**gcolx**(*x*, *xlens*, *y*) char *x*[]; long *xlens*; char *y*[];	Node	Global concatenation operation for contributions of known length.
gdhigh()	**gdhigh**(*x*, *n*, *work*) double *x*[]; long *n*; double *work*[];	Node	Global vector double precision MAX operation.
gdlow()	**gdlow**(*x*, *n*, *work*) double *x*[]; long *n*; double *work*[];	Node	Global vector double precision MIN operation.
gdprod()	**gdprod**(*x*, *n*, *work*) double *x*[]; long *n*; double *work*[];	Node	Global vector double precision MULTIPLY operation.
gdsum()	**gdsum**(*x*, *n*, *work*) double *x*[]; long *n*; double *work*[];	Node	Global vector double precision SUM operation.
giand()	**giand**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Node	Global vector integer bitwise AND operation.
gihigh()	**gihigh**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Node	Global vector integer MAX operation.
gilow()	**gilow**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Node	Global vector integer MIN operation.

Table A-3. C System Calls for Global Operations (2 of 3)

Call	Synopsis	Environment	Description
gior()	**gior**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Node	Global vector integer bitwise OR operation.
giprod()	**giprod**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Node	Global vector integer MULTIPLY operation.
gisum()	**gisum**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Node	Global vector integer SUM operation.
gixor()	**gixor**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Node	Global vector integer bitwise exclusive OR operation.
gland()	**gland**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Node	Global vector logical AND operation.
glor()	**glor**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Node	Global vector logical inclusive OR operation.
glxor()	**glxor**(*x*, *n*, *work*) long *x*[]; long *n*; long *work*[];	Node	Global vector logical exclusive OR operation.
gopf()	**gopf**(*x*, *xlen*, *work*, *f*) char *x*[]; long *xlen*; char *work*[]; long (*f)();	Node	Make a global operation of a user-defined function.
gsendx()	**gsendx**(*type*, *x*, *xlen*, *nodenums*, *nlen*) long *type*; char *x*[]; long *xlen*; long *nodenums*[]; long *nlen*;	Node	Send a vector to a list of nodes.

Table A-3. C System Calls for Global Operations (3 of 3)

Call	Synopsis	Environment	Description
gshigh()	**gshigh**(*x*, *n*, *work*) float *x*[]; long *n*; float *work*[];	Node	Global vector float MAX operation.
gslow()	**gslow**(*x*, *n*, *work*) float *x*[]; long *n*; float *work*[];	Node	Global vector float MIN operation.
gsprod()	**gsprod**(*x*, *n*, *work*) float *x*[]; long *n*; float *work*[];	Node	Global vector float MULTIPLY operation.
gssum()	**gssum**(*x*, *n*, *work*) float *x*[]; long *n*; float *work*[];	Node	Global vector float SUM operation.
gsync()	**gsync()**	Node	Global synchronization operation.

Table A-4. C System Calls for Miscellaneous Operations

Call	Synopsis	Environment	Description
ginv()	long **ginv**(*j*) long *j*;	Host, Node	Return the position of an element in the binary-reflected gray code sequence. Inverse of **gray()**.
gray()	long **gray**(*j*) long *j*;	Host, Node	Return the binary-reflected gray code for an integer.
led()	void **led**(*lstate*) long *lstate*;	Node	Turn the node's green LED on or off.
mclock()	unsigned long **mclock()**	Host, Node	Return elapsed time since node boot (in milliseconds).

Fortran System Call Summary

This section summarizes the Fortran versions of the system calls. See the *iPSC®/2 and iPSC®/860 Programmer's Reference Manual* for more information on these calls.

Table A-5. Fortran Routines for Cube Control (1 of 2)

Routine	Synopsis	Environment	Description
ATTACHCUBE()	SUBROUTINE **ATTACHCUBE**(*cubename*) CHARACTER *cubename**(*)	Host	Attach to a cube and make it the current cube.
CUBEINFO()	INTEGER FUNCTION **CUBEINFO**(*ct, numslots, global*) CHARACTER*16 *ct(slotsize, numslots)* INTEGER *numslots, global*	Host	Obtain information about allocated cubes.
FLICK()	SUBROUTINE **FLICK**()	Host, Node	On a node: Relinquish the CPU to another process. On the host: A no-op.
GETCUBE()	SUBROUTINE **GETCUBE**(*cubename, cubetype, srmname, keep*) CHARACTER *cubename**(*) CHARACTER *cubetype**(*) CHARACTER *srmname**(*) INTEGER *keep*	Host	Allocate a cube.
KILLCUBE()	SUBROUTINE **KILLCUBE**(*node, pid*) INTEGER *node, pid*	Host, Node	Terminate and clear node process(es).
KILLPROC()	SUBROUTINE **KILLPROC**(*node, pid*) INTEGER *node, pid*	Host, Node	Terminate a node process.
KILLSYSLOG()	SUBROUTINE **KILLSYSLOG**()	Host	Terminate *syslog* process.
LOAD()	SUBROUTINE **LOAD**(*filename, node, pid*) CHARACTER *filename**(*) INTEGER *node, pid*	Host, Node	Load a node process.
MYHOST()	INTEGER FUNCTION **MYHOST**()	Host, Node	Obtain node ID of host machine.
MYNODE()	INTEGER FUNCTION **MYNODE**()	Host, Node	Obtain node ID of calling process.

Table A-5. Fortran Routines for Cube Control (2 of 2)

Routine	Synopsis	Environment	Description
MYPID()	INTEGER FUNCTION **MYPID()**	Host, Node	Obtain NX/2 process ID of calling process.
NEWSERVER()	SUBROUTINE **NEWSERVER**(*cubename*) CHARACTER *cubename**(*)	Host	Start new file server for specified cube.
NODEDIM()	INTEGER FUNCTION **NODEDIM()**	Host, Node	Obtain dimension of current cube.
NUMNODES()	INTEGER FUNCTION **NUMNODES()**	Host, Node	Obtain number of nodes in current cube.
RELCUBE()	SUBROUTINE **RELCUBE**(*cubename*) CHARACTER *cubename**(*)	Host	Release specified cube.
SETPID()	SUBROUTINE **SETPID**(*pid*) INTEGER *pid*	Host	Set NX/2 process ID for host program.
SETSYSLOG()	SUBROUTINE **SETSYSLOG**(*stdfd*) INTEGER *stdfd*	Host	Start *syslog* process.
WAITALL()	SUBROUTINE **WAITALL**(*node, pid*) INTEGER *node, pid*	Host, Node	Wait for all specified processes to complete.
WAITONE()	SUBROUTINE **WAITONE**(*node, pid, cnode, cpid, ccode*) INTEGER *node, pid* INTEGER *cnode, cpid, ccode*	Host, Node	Wait for specified processes to complete.

Table A-6. Fortran Routines for Message Passing (1 of 4)

Routine	Synopsis	Environment	Description
CPROBE()	SUBROUTINE **CPROBE**(*typesel*) INTEGER *typesel*	Host, Node	Wait for a message to arrive.
CRECV()	SUBROUTINE **CRECV**(*typesel, buf, len*) INTEGER *typesel* INTEGER *buf*(*) INTEGER *len*	Host, Node	Receive a message, and wait for completion.

Table A-6. Fortran Routines for Message Passing (2 of 4)

Routine	Synopsis	Environment	Description
CSEND()	SUBROUTINE **CSEND**(*type, buf, len,node, pid*) INTEGER *type* INTEGER *buf*(*) INTEGER *len, node, pid*	Host, Node	Send a message, and wait for completion.
CSENDRECV()	INTEGER FUNCTION **CSENDRECV**(*type, sbuf, slen, tonode, topid, typesel, rbuf, rlen*) INTEGER *type* INTEGER *sbuf*(*) INTEGER *slen, tonode, topid, typesel* INTEGER *rbuf*(*) INTEGER *rlen*	Host, Node	Simultaneously, send a message, and post a receive for the reply. Wait for completion.
FLUSHMSG()	SUBROUTINE **FLUSHMSG**(*typesel, node, pid*) INTEGER *typesel, node, pid*	Host, Node	Flush specified messages from the system.
HRECV()	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*;	Node	Provide user-written exception handler for receive traps.
HSEND()	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*;	Node	Send a message, and set up a handler procedure to be called when the reply arrives.

Table A-6. Fortran Routines for Message Passing (3 of 4)

Routine	Synopsis	Environment	Description
HSENDRECV()	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*;	Node	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()**	INTEGER FUNCTION **INFOCOUNT()** INTEGER FUNCTION **INFONODE()** INTEGER FUNCTION **INFOPID()** INTEGER FUNCTION **INFOTYPE()**	Host, Node	Return information about a pending or received message.
IPROBE()	INTEGER FUNCTION **IPROBE**(*typesel*) INTEGER *typesel*	Host, Node	Determine whether a message of a selected type is pending.
IRECV()	INTEGER FUNCTION **IRECV**(*typesel, buf, len*) INTEGER *typesel* INTEGER *buf*(*) INTEGER *len*	Host, Node	Receive a message.
ISEND()	INTEGER FUNCTION **ISEND**(*type, buf, len, node, pid*) INTEGER *type* INTEGER *buf*(*) INTEGER *len, node, pid*	Host, Node	Send a message.
ISENDRECV()	INTEGER FUNCTION **ISENDRECV**(*type, sbuf, slen, tonode, topid, typesel, rbuf, rlen*) INTEGER *type* INTEGER *sbuf*(*) INTEGER *slen, tonode, topid, typesel* INTEGER *rbuf*(*) INTEGER *rlen*	Host, Node	Simultaneously, send a message, and post a receive for the reply. Do not wait for completion.
MASKTRAP()	INTEGER FUNCTION **MASKTRAP**(*state*) INTEGER *state*	Node	Enable or disable a receive trap.

Table A-6. Fortran Routines for Message Passing (4 of 4)

Routine	Synopsis	Environment	Description
MSGCANCEL()	SUBROUTINE **MSGCANCEL**(*id*) INTEGER *id*	Host, Node	Cancel a send or receive operation.
MSGDONE()	INTEGER FUNCTION **MSGDONE**(*id*) INTEGER *id*	Host, Node	Determine whether a send or receive operation has completed.
MSGWAIT()	SUBROUTINE **MSGWAIT**(*id*) INTEGER *id*	Host, Node	Wait for completion of a send or receive operation.

Table A-7. Fortran Routines for Global Operations (1 of 3)

Routine	Synopsis	Environment	Description
GCOL()	SUBROUTINE **GCOL**(*x, xlen, y, ylen, ncnt*) INTEGER *x*(*) INTEGER *xlen* INTEGER *y*(*) INTEGER *ylen* INTEGER *ncnt*	Node	Global concatenation operation.
GCOLX()	SUBROUTINE **GCOLX**(*x, xlens, y*) INTEGER *x*(*) INTEGER *xlen* INTEGER *y*(*)	Node	Global concatenation operation for contributions of known length.
GDHIGH()	SUBROUTINE **GDHIGH**(*x, n, work*) DOUBLE PRECISION *x*(*) INTEGER *n* DOUBLE PRECISION *work*(*)	Node	Global vector double precision MAX operation.
GDLOW()	SUBROUTINE **GDLOW**(*x, n, work*) DOUBLE PRECISION *x*(*) INTEGER *n* DOUBLE PRECISION *work*(*)	Node	Global vector double precision MIN operation.
GDPROD()	SUBROUTINE **GDPROD**(*x, n, work*) DOUBLE PRECISION *x*(*) INTEGER *n* DOUBLE PRECISION *work*(*)	Node	Global vector double precision MULTIPLY operation.

Table A-7. Fortran Routines for Global Operations (2 of 3)

Routine	Synopsis	Environment	Description
GDSUM()	SUBROUTINE **GDSUM**(*x*, *n*, *work*) DOUBLE PRECISION *x*(*) INTEGER *n* DOUBLE PRECISION *work*(*)	Node	Global vector double precision SUM operation.
GIAND()	SUBROUTINE **GIAND**(*x*, *n*, *work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Node	Global vector integer bitwise AND operation.
GIHIGH()	SUBROUTINE **GIHIGH**(*x*, *n*, *work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Node	Global vector integer MAX operation.
GILOW()	SUBROUTINE **GILOW**(*x*, *n*, *work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Node	Global vector integer MIN operation.
GIOR()	SUBROUTINE **GIOR**(*x*, *n*, *work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Node	Global vector integer bitwise OR operation.
GIPROD()	SUBROUTINE **GIPROD**(*x*, *n*, *work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Node	Global vector integer MULTIPLY operation.
GISUM()	SUBROUTINE **GISUM**(*x*, *n*, *work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Node	Global vector integer SUM operation.
GIXOR()	SUBROUTINE **GIXOR**(*x*, *n*, *work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Node	Global vector integer bitwise exclusive OR operation.
GLAND()	SUBROUTINE **GLAND**(*x*, *n*, *work*) LOGICAL *x*(*) INTEGER *n* LOGICAL *work*(*)	Node	Global vector logical AND operation.

Table A-7. Fortran Routines for Global Operations (3 of 3)

Routine	Synopsis	Environment	Description
GLOR()	SUBROUTINE **GLOR**(*x*, *n*, *work*) LOGICAL *x*(*) INTEGER *n* LOGICAL *work*(*)	Node	Global vector logical inclusive OR operation.
GLXOR()	SUBROUTINE **GLXOR**(*x*, *n*, *work*) LOGICAL *x*(*) INTEGER *n* LOGICAL *work*(*)	Node	Global vector logical exclusive OR operation.
GOPF()	SUBROUTINE **GOPF**(*x*, *xlen*, *work*,*f*) INTEGER *x*(*) INTEGER *xlen* INTEGER *work*(*) EXTERNAL *f*	Node	Make a global operation of a user-defined function.
GSENDX()	SUBROUTINE **GSENDX**(*type*, *x*, *xlen*, *nodenums*, *nlen*) INTEGER *type* INTEGER *x*(*) INTEGER *xlen* INTEGER *nodenums*(*) INTEGER *nlen*	Node	Send a vector to a list of nodes.
GSHIGH()	SUBROUTINE **GSHIGH**(*x*, *n*, *work*) REAL *x*(*) INTEGER *n* REAL *work*(*)	Node	Global vector real MAX operation.
GSLOW()	SUBROUTINE **GSLOW**(*x*, *n*, *work*) REAL *x*(*) INTEGER *n* REAL *work*(*)	Node	Global vector real MIN operation.
GSPROD()	SUBROUTINE **GSPROD**(*x*, *n*, *work*) REAL *x*(*) INTEGER *n* REAL *work*(*)	Node	Global vector real MULTIPLY operation.
GSSUM()	SUBROUTINE **GSSUM**(*x*, *n*, *work*) REAL *x*(*) INTEGER *n* REAL *work*(*)	Node	Global vector real SUM operation.
GSYNC()	SUBROUTINE **GSYNC**()	Node	Global synchronization operation.

Table A-8. Fortran Routines for Miscellaneous Operations

Routine	Synopsis	Environment	Description
GINV()	INTEGER FUNCTION **GINV(*j*)** INTEGER *j*	Host, Node	Return the position of an element in the binary-reflected gray code sequence. Inverse of **GRAY()**.
GRAY()	INTEGER FUNCTION **GRAY(*j*)** INTEGER *j*	Host, Node	Return the binary-reflected gray code for an integer.
LED()	SUBROUTINE **LED(*lstate*)** INTEGER *lstate*	Node	Turn the node's green LED on or off.
MCLOCK()	INTEGER FUNCTION **MCLOCK()**	Host, Node	Return elapsed time since node boot (in milliseconds).

Error Messages B

Error Messages

The following error messages are generated by the Simulator's user interface. They are directed to the standard error, which is by default sent to the user's terminal.

The symbol <string> indicates that a string is placed in the message at this location.

"sim: error in opening log file <string>"

"sim: unrecognized command: <string>"

"sim: missing parameters"

"sim: missing node number after -n"

"sim: illegal value for node: <string>"

"sim: illegal dimension: <string>"

"sim: node specified not compatible with dimension"

"sim: missing file name"

"sim: file <string> not found"

"sim: too many processes; start failed."

"sim: missing filename after -l"

"sim: unknown option: <string>"

"sim: illegal argument: <string> ; use "on" or "off"."

"sim: missing process id after -p"

"sim: illegal value for process id: <string>"

"sim: missing argument after -S"

"sim: missing argument after -R"

"sim: missing argument after -Q"

"sim: missing argument after -B"

"sim: missing argument after -K"

"sim: illegal arguments list"

"sim: no processes to start"

"sim: process to be started not found"

"sim: illegal load option <string>"

"sim: too many processes; load failed"

"sim: process id in use; load failed"

"sim: no node process is loaded"

"sim: PID <string> not loaded"

"sim: unknown user command"

"sim: user error occurred in process <string>"

"sim: internal error 1"

"sim: internal error 2"

"sim: internal error 3"

"sim: internal error 4"

"sim: internal error 5"

"sim: internal error 6"

"sim: internal error 7"

"sim: internal error 8"

"sim: internal error 9"

The following informatory messages are generated by the simulator's user interface. They are sent to the standard output.

"sim: interrupt occurred"

"sim: simulation done"

"Tracing is on."

"Tracing is off."

"Tracing was on."

"Tracing was off."

"Node <string> PID <string> completed, code <string> (no meaning)"

The following error messages are generated when errors are detected in the use of the NX commands.

iPSC/2 error messages are:

"csend: invalid type"

"csend: invalid length"

"csend: invalid node"

"csend: invalid pid"

"csend: no pid defined"

"csend: no buffer space available"

"isend: invalid type"

"isend: invalid length"

"isend: invalid node"

"isend: invalid pid"

"isend: no pid defined"

"isend: no more tids"

"isend: no buffer space available"

"crecv: invalid type"

"crecv: invalid length"

"crecv: no pid defined"

"hrecv: invalid type"

"hrecv: invalid length"

"irecv: invalid type"

"irecv: invalid length"

"irecv: no pid defined"

"irecv: no more tids"

"msgdone: invalid tid"

"msgdone: no pid defined"

"msgwait: invalid tid"

"msgwait: no pid defined"

"cprobe: invalid type"
"cprobe: no pid defined"

"iprobe: invalid type"
"iprobe: no pid defined"

"setpid: invalid pid"
"setpid: pid already in use"
"setpid: not a host process"
"mynode: no pid defined"

"mypid: no pid defined"

"mysrm: no pid defined"

"numnodes: no pid defined"

"infocount: no pid defined"
"infotype: no pid defined"
"infonode: no pid defined"
"infopid: no pid defined"

"load: invalid pid"
"load: invalid node"
"load: no pid defined"
"load: path name is too long"
"load: file name not found"
"load: process id is in use"

"killcube: invalid pid"

"killcube: invalid node"

"killcube: no pid defined"

"killproc: invalid pid"

"killproc: invalid node"

"killproc: no pid defined"

"waitone: selected process not found"

"waitone: invalid pid"

"waitone: invalid node"

"waitone: no pid defined"

"waitone: no buffer space available"

"waitall: invalid pid"

"waitall: invalid node"

"waitall: no pid defined"

"waitall: no buffer space available"

iPSC/1 error messages are:

"send: invalid channel"

"send: channel not open"

"send: not process's channel"

"send: invalid type"

"send: invalid length"

"send: invalid node"

"send: invalid pid"

"send: no buffer space available"

"send: invalid channel"

"sendw: channel not open"

"sendw: not process's channel"

"sendw: invalid type"

"sendw: invalid length"

"sendw: invalid node"

"sendw: invalid pid"

"sendw: no buffer space available"

"sendmsg: channel not open"

"sendmsg: not process's channel"

"sendmsg: invalid type"

"sendmsg: invalid length"

"sendmsg: invalid node"

"sendmsg: invalid pid"

"sendmsg: no buffer space available"

"recv: invalid channel"

"recv: channel not open"

"recv: not process's channel"

"recv: invalid type"

"recv: invalid length"

"recvw: invalid channel"

"recvw: channel not open"

"recvw: not process's channel"

"recvw: invalid type"

"recvw: invalid length"

"recvmsg: invalid channel"

"recvmsg: channel not open"

"recvmsg: not process's channel"

"recvmsg: invalid type"

"recvmsg: invalid length"

"status: invalid channel"

"status: channel not open"

"status: not process's channel"

"probe: invalid channel"

"probe: channel not open

"probe: not process's channel"

"probe: invalid type"

"probemsg: invalid pid"

"copen: invalid pid"

"copen: no channel available"

"cclose: invalid channel""cclose: channel not open"

"cclose: not process's channel"

"load: invalid pid"

"load: invalid node"

"load: path name is too long"

"load: file name not found"

"load: process id is in use"

"lkill: invalid pid"

"lkill: invalid node"

"lwait: selected process not found"

"lwait: invalid pid"

"lwait: invalid node"

"lwait: no buffer space available"

"lwaitall: invalid pid"

"lwaitall: invalid node"

"lwaitall: no buffer space available"