April 1991
Order Number: 312043-001

iPSC®/2 AND iPSC®/860
INTERACTIVE PARALLEL DEBUGGER
MANUAL

intel® Corporation

REV.	REVISION HISTORY	DATE
-001	Original Issue	04/91

RESTRICTED RIGHTS

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

PREFACE

This manual describes the Interactive Parallel Debugger (IPD), a symbolic source-level debugger for Fortran and C programs running on iPSC system nodes. It contains information describing how to use IPD, as well as detailed reference information on the IPD commands.

NOTE

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

This manual assumes that you are an application programmer proficient in the C and Fortran languages and the iPSC Supercomputer. The manual contains an overview of the Interactive Parallel Debugger (IPD), presents an example of using IPD with a simple Fortran program, and lists all the IPD commands in a reference format.

ORGANIZATION

Chapter 1	"Introduction," is overview of IPD features. It also presents some important information that you need to understand to use IPD effectively.
Chapter 2	"Using IPD Commands in a Sample Session," introduces most of the IPD commands in the form of a debug session that finds a bug in a parallel application.
Chapter 3	"IPD Commands," provides detailed information on all the IPD commands in a reference format.
Appendix A	"Source Code for the IPD Example," contains a complete Fortran source code listing of the example used in Chapter 2.

NOTATIONAL CONVENTIONS

This manual uses the following notational conventions:

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

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

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

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

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

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

[ ] (Brackets) Surround optional items.

. . . (Ellipsis dots) Indicate that the preceding item may be repeated.

| (Bar) Separates two or more items of which you may select only one.

{ } (Braces) Surround two or more items of which you must select one.

APPLICABLE DOCUMENTS

For more information, refer to the following manuals:

iPSC® System Manuals

iPSC®/2 and iPSC®/860 C Commands and Routines Quick Reference
Summarizes all C routines and commands for the iPSC system.

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

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

iPSC®/2 and iPSC®/860 Interactive Parallel Debugger Commands Quick Reference
Summarizes all iPSC system Interactive Parallel Debugger commands.

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

Other Manuals

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

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

UNIX System V Manual Set
Describes UNIX System V.

UNIX V - The Quick Reference Guide
Summarizes UNIX commands, buzzwords, C shell hints and standard directory layout.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

CHAPTER 2
USING IPD COMMANDS IN A SAMPLE SESSION

CHAPTER 3
IPD COMMANDS

APPENDIX A
SOURCE CODE FOR THE IPD EXAMPLE

LIST OF ILLUSTRATIONS

LIST OF TABLES

INTRODUCTION 1

INTRODUCTION

The iPSC Interactive Parallel Debugger (IPD) is a full-featured symbolic source-level debugger for parallel C, Fortran, and assembly language programs running on the iPSC system. IPD incorporates most of the features of standard debuggers designed for serial programs, and adds custom features that support the parallel programming model. IPD allows you to debug software designed to run on 386-based CX nodes, i860™-based RX nodes, or software running on both types of nodes.

IPD contains features specifically designed to aid debugging in a parallel environment. Among its parallel debugging features are the debug context, data reduction, an asynchronous user interface, and facilities to help debug message-passing. For example, you can set breakpoints for some processes and not others, monitor the message queues, and display a process status table. You can also display and change program variables using symbolic names.

As an aid to the programmer, IPD command and display syntax follows the language convention of the program being debugged. For Fortran programs, variable specifications follow the Fortran language convention, while for C programs, variable specifications follow the C language convention.

IPD FEATURES

Under IPD, you have control over the debug environment, you can load your program and control the program's execution, and there are several ways to examine and modify the program.

Your control over the debug environment includes the following:

- **Customizing the debug environment.** You can define aliases and debug variables. Aliases are shorthand versions of IPD commands. Debug variables are shorthand versions of strings used by IPD commands.

- **Logging a debug session.** You can set up a log file to record all the IPD commands and their responses.

- **Executing IPD command files.** You can execute files containing IPD commands from within IPD. If you create a file named *.ipdrc* in your home or current working directory it is executed automatically when you invoke IPD.

- **On-line help.** You can get help from IPD as you use it. For a brief description of all IPD commands, you can enter the **help** command (the abbreviation is **h**), or you can just enter a question mark (**?**). For more specific help on a given command, enter **help** or **?**, followed by the name of the command on which you wish help. This displays a brief description of the function, followed by the syntax.

IPD gives you precise control over program execution. This control includes the following:

- **Setting breakpoints.** You can set and remove execution (code) and data breakpoints. You can set execution breakpoints at procedure calls, source line numbers of executable statements, and instruction addresses. The program stops just before executing the specified code. You can set a data breakpoint on a variable name or a memory address, and specify whether the program is to break on an access or on a write. Access is defined as either a read or a write.

- **Starting execution.** You can start execution from the beginning of the program, continue after halting within the program, and single-step through the program.

- **Listing code.** You can list the source code, including line numbers, of the program that you're debugging. You can list source while single-stepping through the program. You can also display assembly code compiled for the i860 microprocessor-based RX nodes for debugging at a more detailed level.

- **Displaying the message and receive queues.** Many program errors have to do with message passing. With IPD, you can display the message and receive queues.

- **Displaying program variables.** This allows you to make sure that your program variables and memory addresses have the expected intermediate values.

IPD provides parallel debugging facilities for debugging programs that run on multiple nodes or, in 386 programs, that consist of multiple processes.

- **Setting the debug context.** IPD uses the *debug context* to keep track of the nodes and processes to which you are issuing IPD commands. For example, the context *(all:0)* is process 0 on all nodes. In programs compiled to run on multi-process CX nodes, you specify the process ID as part of the context. On RX nodes, the process IPD must always be 0.You can tell IPD which nodes and which processes IPD commands apply to. IPD displays the default context as part of the prompt, and you can either change the context or override it for a given command at any time in your IPD session.

- **Wait for completion or a breakpoint.** When you issue a **run** or **continue** command, IPD starts the specified processes executing and immediately returns control to the user (i.e., issues a prompt). The node processes run asynchronously unless you issue the **wait** command to wait for all nodes to encounter breakpoints or terminate. Then the IPD prompt will not return until all the processes, even those not in context, have stopped running, unless you issue a keyboard interrupt. It is important to be aware that executing processes are allowed to write to *stdout* and *stderr* in two situations only:

  - Before each IPD prompt.

  - During execution of a **wait** command.

  Processes may read from the keyboard only during execution of the command.

COMPILING FOR DEBUG

To debug programs under IPD, there are certain compile requirements:

- With the 386 compiler, use the **-B** switch to ensure that IPD has access to the appropriate debug information. By listing **-B** last in the sequence of compiler flags, it will have precedence over the previous flags, except for the **-O** and **-OLM** flags. You should remove these flags.

- For the i860 compiler, you need to use the **-g** switch for access to debug information. Programs not compiled with **-g** should be compiled with **-ga** to ensure that stack frames are generated for procedure calls.

- All programs must be compiled with no optimization.

You can debug programs not compiled for debug, but access to local symbols and line numbers will not be available.

IPD only runs on the System Resource Manager. From a remote workstation, you need to remotely log onto the SRM.

You must use the IPD **load** command to load your file into IPD for debugging.

Certain iPSC system calls interface with the normal operation of IPD. These system calls (**load()**, **getcube()**, **relcube()**, **killcube()**, and **killproc()**) should not be included in any version *of a program* you are debugging. If your program includes any of these iPSC routines, comment them out in your source, recompile, and relink.

INVOKING IPD

Because IPD resides on the iPSC SRM, you must invoke IPD under UNIX from the SRM. You may be logged in either directly or remotely. The IPD invocation syntax is

ipd

When you invoke the debugger, IPD automatically executes commands in its configuration file, if you create such a file. This file must called *.ipdrc* (note the period) and must reside in your home directory. It is an ASCII file containing IPD commands (see the **exec** command for more information).

You can load only node programs; IPD does not debug programs running on the SRM or a remote workstation.

You must use the iPSC **getcube** command to get the nodes on which you are going to load your program before invoking IPD. If you release the cube while IPD is executing, IPD will be killed.

IPD COMMANDS

The commands fall generally into three categories: execution control, program display, and debug environment. Tables 1-1 through 1-3 list the IPD commands by these functions. You can abbreviate any command, keyword, or switch to the minimum number of characters required to uniquely identify it. For example, for the **process** command, all of these abbreviations are valid: **proces, proce, proc, pro, pr** or **p**. If the command abbreviation is ambiguous, IPD will display an error message and ask you to retype the command.The tables also show the minimum abbreviation for each command.

Table 1-1. Execution Control Commands

Command	Abbrev	Description
break	**b**	Set and display breakpoints
continue	**conti**	Continue stopped or breakpointed processes
kill	**k**	Terminate processes
remove	**rem**	Remove breakpoints
run	**ru**	Start one or more processes
step	**ste**	Execute the next source statement
stop	**sto**	Stop execution of processes
wait	**w**	Wait until processes stop running

Table 1-2. Program Examination and Modification Commands

Command	Abbrev	Description
assign	**as**	Assign a new value to a program variable or memory location
disassemble	**disa**	Display assembler listing of i860 node program code
display	**disp**	Display the value of a program variable or memory location
frame	**fr**	Display the runtime activation stack
list	**li**	List source code of loaded program
msgqueue	**msgq**	Display messages sent but not yet received
recvqueue	**rec**	Display posted receives not yet satisfied
process	**p**	Display current state of processes
type	**t**	Display type of variable

Table 1-3. Debug Environment Commands

Command	Abbrev	Description
alias	**al**	Set or display command aliases
unalias	**una**	Delete command aliases
context	**conte**	Set the current node and process context
quit or exit	**q**	Exit IPD
exec	**exe**	Read in and execute a command file
source	**so**	Set or display the source directory search path list
help or ?	**h**	Display IPD commands and syntax
load	**loa**	Load node programs
log	**log**	Record the debug session
more	**mo**	Turn terminal scrolling on or off
set	**se**	Set or display command line variables
status	**sta**	Display current IPD status
unset	**uns**	Delete command line variables
system or !	**sy**	Execute a UNIX command

The only commands that you can issue prior to the load command are those in Table 1-3 (Debug Environment). With the exception of the **load** command, which sets the default context, and the context command, which allows you to change the default context, none of these commands use the context. All other IPD commands do require a default or specified context.

Sample Aliases

IPD searches the IPD alias list (see **alias** command) first before it matches a command with the IPD command table. If you wish to abbreviate a command further you may alias the command to one or more characters. Table 1-4 lists a suggested set of aliases for some of the commands that you are likely to use the most during a debug session. If you create a file containing these commands named *.ipdrc* in your home directory, these definitions are automatically included whenever you invoke IPD.

Table 1-4. Suggested IPD Aliases

Alias	Command	Description
a	**assign**	Assign a new value to a variable or address.
ct	**context**	Set the current node and process context
c	**continue**	Continue stopped or breakpointed processes
cw	**continue;wait**	Continue processes and then wait
d	**display**	Display the value of a variable or address
ls	**list**	List source program for loaded program
mq	**msgqueue**	Display messages sent but not yet received
ps	**process (all:all)**	Display current state of all processes
rm	**remove**	Remove breakpoints
r	**run**	Start up one or more processes
dir	**source**	Set or display the current source directory
s	**step**	Execute the next source statement
vi	**system vi**	Invoke the vi editor

Syntax of IPD Commands

Commands have the form:

command *arguments;* **command** *arguments # comments*
or
command -keywords *arguments # comments*
or
command (*context*) **-keywords** *arguments # comments*

where

arguments — Command arguments specific to each command. If the command accepts a number of arguments then the arguments must be separated by spaces. The order of command line arguments depends upon the command. For example, the order of the arguments for **assign** is significant, but not for **remove**. Refer to each command description to determine if the command line argument order is important.

keywords — Keywords appear in **boldface** in this document. A keyword without a following argument is a command line switch. Command line switches can appear anywhere on the command line after the command name. Keywords with a following argument are usually position dependent. You should refer to each command description to determine if the command line keyword and argument order is important.

(*context*) — The context argument is always defined within parentheses. The context argument defines the set of processes and nodes that are the target of the IPD command (see **context** command). The context argument must appear immediately after the command and before all other arguments.

; — The semicolon is a command separator. Multiple commands may appear on the same command line separated by a semicolon. The exceptions to this rule are the **alias, set** and **system** commands and comments.

 # comments — A comment consists of a pound sign (#) followed by a space or a pound sign (#) as the first character of a command line. All characters to the end of the line are considered comment characters and are not interpreted by IPD. This includes semicolons.

To specify an address or value in octal, it must have a leading *0o*. To specify an value in hexadecimal, it must have a leading *0x*. The leading zero is required.

For all IPD commands a *filename* argument refers to a UNIX pathname where the tilde (~) character denotes your home directory. IPD will only substitute the tilde with your environment variable *$HOME*. IPD will not expand *~user* names.

CONTEXT, EXECUTION POINT, AND SCOPE

To use IPD, you need to understand *debug context*, *execution point*, and *scope*. The context defines the nodes and processes under debug — those to which IPD commands refer. The execution point is the point in a process just before the next statement to be executed. Each process has its own execution point. The scope of a variable is within those parts of a program where it is recognized and accessible. The execution point determines what variables are in scope and what file a line number refers to.

The context determines the nodes and the processes on those nodes that an IPD command affects. When you enter IPD and execute the **load** command, you must specify the nodes onto which you are loading your program and the process IDs used to refer to them. This sets the initial default context. If you do not specify a context for the commands whose syntax allows you to specify a context, IPD uses the default context. You can change the default context with the **context** command. The default context is shown as the IPD prompt.

Some of the IPD commands require the context to include only processes running the same object module. A *load module* is an executable object module that you have loaded onto the system with the IPD load command. These commands are **assign**, **break**, **disassemble**, **display**, **list**, and **type**.

For example, you can set breakpoints, list, or disassemble in only one load module at a time. Restricting certain commands to a single load module makes sense, because you would not, in general, arbitrarily set a breakpoint on a single line number in completely different load modules. You may use these commands on multiple nodes as long as the same modules are loaded on these nodes. If your context for these commands is such that it specifies different load modules, IPD returns an error.

For the other IPD commands that use the context, the execution points for different nodes or processes can be in different load modules. These commands fall generally under the headings either of execution commands or information display commands.

- Execution commands that allow context to be in separate load modules: **continue**, **kill**, **run**, **step**, **stop**, **wait**.

- Information display commands that allow context to be in separate load modules: **frame**, **msgqueue**, **process**, **recvqueue**.

IPD gives you several ways to determine the current scope and context, such as the display of the current context as the prompt, and the **context**, **frame**, and **process** commands. While you have access to any point in the program(s) that you have loaded using IPD, if the current execution point is not within the routine or program to which you want access, you need to prefix the variable name with the routine name and/or the file name on the command line. Likewise, you need to set the context either with the **context** command or within a given command to make sure that the command applies to the nodes and/or processes that you want it to.

Consider the following example. The **frame** command displays processes that have been activated as you execute the program. For this program, the **frame** command tells you that nodes 0 through 2 are waiting in the **flick**() system call after executing the **gdhigh**() system call, node 3 is hung up in a different routine, the **msgwait**() system call in the *shadow* routine:

```
(all:0) > frame
  ***** (0..2:0) *****
    __flick()     [_flick.s{}0x00023fe8]
    _gdhigh()     [_gdhigh.c{}0x000240f8]
    gdhigh_()     [gdhigh_.c{}0x0001e9dc]
    gauss()     [gauss.f{}#72]
    main()     [pgfmain.c{}0x000001ac]
  ***** (3:0) *****
    __flick()     [_flick.s{}0x00023fe8]
    msgwait_()     [msgwait_.c{}0x0002011c]
    shadow()     [gauss.f{}#209]
    gauss()     [gauss.f{}#58]
    main()     [pgfmain.c{}0x000001ac]
```

If multiple processes in the current context would result in identical display of information, the information is displayed only once, preceded by a line displaying the context to which the information applies. In the previous example, nodes 0 through 2 were doing the same thing; node 3 has a separate display because the information is different.

The next asks for the display of the value of the variable *iam* in the *shadow* routine. You need to make sure the scope is correct:

```
(all:0) > disp iam
 *** ERROR: Not found: variable _flick.s{}__flick()iam
```

The error message indicates that the variable is not in the current scope; while the **frame** command showed that the nodes executing the program stopped in the **flick**() routine, the variable you are looking for is in the *shadow* routine. The following results if you qualify the variable name with the name of the routine:

```
(all:0) > disp shadow()iam
 ** gauss.f{}shadow()iam **
ERROR(S) DETECTED ON NODES

 *** NODE ERROR: Procedure is not active: node 0: reading data on
                 node
```

This failure is due to an incorrect context, so you need to override the default context; in this case, node 3 is the only one executing the shadow routine.

```
(all:0) > disp (3:0) shadow()iam
 ** gauss.f{}shadow()iam **
  ***** (1:0) *****
  iam = 1
```

EFFECTS ON IPD OF DIFFERENT NODE TYPES AND COMPILERS

Different node types and compilers have certain effects on the operation of IPD of which you should be aware.

Breakpoint Location Differences

Data breakpoints in IPD behave differently on programs running on i860-based RX nodes and 386-based CX nodes. On RX nodes, data breakpoint traps occur *before* the memory access occurs; on CX nodes, data breakpoint traps occur *after* the memory access occurs.

Compilers for the i860 microprocessor and the 386 microprocessor number lines differently in certain block structures:

- The i860 compiler does not provide line number information for the closing curly brace of a multiple-line C statement. As a result, you can't set a breakpoint on such a line. For example:

```
while (i<0) {
    ...
}                                   /* You cannot set the breakpoint here */
```

- Multi-line C functions such as the following are numbered differently:

```
printf( "%d %d %d %d\n",            /* set breakpoint here in 386 code */
                        i,
                        j,
                        k,
                        l );        /* set breakpoint here in i860 code */
```

On RX nodes, line number information is generated only for the LAST line of the function call (where *l* is); on CX nodes, the first line of the function call is assigned the line number.

Main Program Naming Differences

Fortran programs are not required to have a PROGRAM statement. There are various conventions for naming the main routine Fortran when a PROGRAM statement is not supplied by the programmer. For the RX Fortran compiler, if the PROGRAM statement is omitted, the main routine is given the name *_unnamed()* for the CX compiler, it is given the name *MAIN()*.

You need to be aware of this when you are qualifying breakpoints or variables in the main routine.

Lower Bounds of Fortran Arrays on CX Nodes

If you compile a Fortran program to run on CX nodes, and you define a lower bound for a Fortran array that is not equal to 1, you will have to calculate subscripts as though the lower bound were 1, as in the following declaration:

```
INTEGER IARY(0:4)
```

This array has five elements. To display *IARY*(0) on a CX node using IPD, you would use the following command:

```
(1:0) > display iary(1)
```

For code compiled to run on RX nodes, you use the correct subscripts (i.e., ***iary(0)***).

Display of Fortran Variable Types

Fortran character types result in the displays indicated in Tables 1-5 and 1-6. The displays that might not be what the user expected have "<---" after them. This results because the symbol table does not have sufficient information to distinguish the declared type from the printed type in these instances.

Table 1-5. Fortran Variable Type Display On CX Nodes

Declared type	Printed type
character var	CHARACTER*1 var
character*n var	CHARACTER*n var
character*n var(x,y)	CHARACTER*n var(x,y)
logical*1 var	LOGICAL*1 var
logical*1 var(x)	CHARACTER*x var <---
logical*1 var(x,y)	CHARACTER*x var(y) <---
logical*2 var	INTEGER*2 var <---
logical*4 var	INTEGER var <---
logical var	INTEGER var <---
integer*2 var	INTEGER*2 var
integer*4 var	INTEGER var
integer var	INTEGER var
real*4 var	REAL var
real var	REAL var
real*8 var	DOUBLE PRECISION var
double precision var	DOUBLE PRECISION var
complex var	COMPLEX var
complex*8 var	COMPLEX var
complex*16 var	DOUBLE COMPLEX var
structure var	illegal var type
union /var/	illegal var type

Table 1-6. Fortran Variable Type Display On RX Nodes

Declared type	Printed type	
character var	CHARACTER*1 var	
character*n var	CHARACTER*n var	
character*n var(x,y)	CHARACTER*n var(x,y)	
logical*1 var	LOGICAL*1 var	
logical*1 var(x)	LOGICAL*1 var(x)	
logical*1 var(x,y)	LOGICAL*1 var(x,y)	
logical*2 var	INTEGER*2 var	<---
logical*4 var	INTEGER var	<---
logical var	INTEGER var	<---
integer*2 var	INTEGER*2 var	
integer*4 var	INTEGER var	
integer var	INTEGER var	
real*4 var	REAL var	
real var	REAL var	
real*8 var	DOUBLE PRECISION var	
double precision var	DOUBLE PRECISION var	
complex var	COMPLEX var	
complex*8 var	COMPLEX var	
complex*16 var	DOUBLE COMPLEX var	
structure var	STRUCTURE var	
union /var/	UNION /var/	

ADDITIONAL INFORMATION

You should be aware of the following additional information when you are using IPD:

- You cannot set a data breakpoint on a local variable if your current execution point lies in a function not compiled for debug. IPD requires valid frame pointers to determine whether the data breakpoint is active to ensure correct operation of local data breakpoints, and would not be able to do so without debug information. IPD cannot ensure this and therefore does not allow it.

- The amount of data returned by any IPD request is currently limited to 1K (1024) bytes. This means that the following commands are limited as follows:

  - **assign** may only assign 1024 bytes (per node)

  - **display** may only display 1024 bytes (per node)

  - **msgq** may only display the first 42 messages in the queue (per node)

  - **recvq** may only display the first 51 receive requests in the queue (per node)

  - **frame** may only display the top 256 procedures of the call chain (per node)

- There are critical sections in the debugger where IPD does not allow the user to interrupt it from the keyboard. This is necessary because IPD on the SRM and NX on the nodes have data structures (for keeping track of processes, breakpoints, etc.) that must be synchronized at all times. Thus, the user is not allowed to interrupt during the modification of these data structures.

- If you are sure that the system has hung up and is not going to respond, using the control-backslash (**<Ctrl-\ >**) key sequence should kill the debugger. If the system is *still* hung up, subsequent interrupts (**<DEL> or <Ctrl-C>**) and the IPD **quit** commands should work.

The next chapter introduces you to the IPD commands, and the final chapter provides detailed reference information on each of the commands.

USING IPD COMMANDS IN A SAMPLE SESSION 2

INTRODUCTION

This chapter presents general information on how to use IPD in the context of a sample debug session. The program used here solves LaPlace's equation in two dimensions using the iterative Gauss-Seidel method, in which each iterative cycle requires that each array element is averaged with its nearest neighbors. This application decomposes the problem by columns; that is, blocks of columns are distributed to each processor to be used in the computation.

Because each node owns a block of columns, only the outer column of each of these blocks must be shared with another node. To reduce communication between the nodes, each processor maintains a "shadow" buffer that contains a copy of the neighboring column from the previous iteration, as illustrated in Figure 2-1.

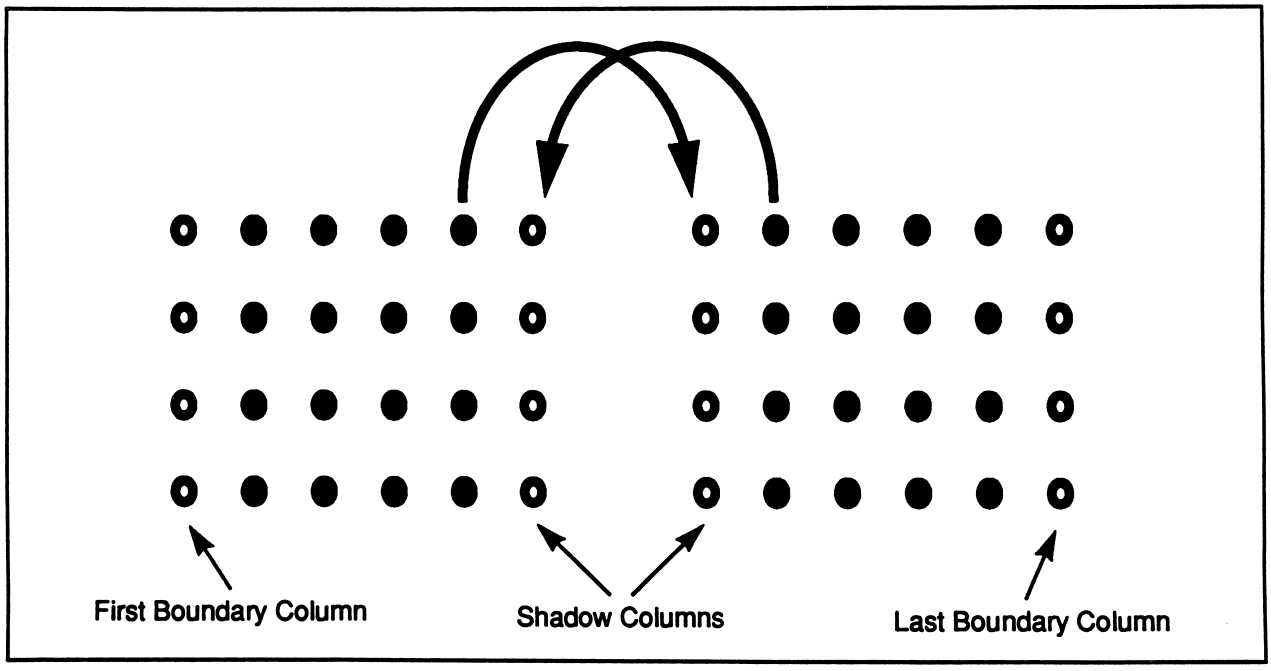

Figure 2-1. Shadow Columns in the Gauss-Seidel Example

PREPARING THE EXAMPLE

Copy the contents of */usr/ipsc/examples/flipd* to your own area and issue the **make** command. Because the example was run on RX nodes, it specifies ***rx*** on the make command line. If you are using CX nodes, enter ***cx*** instead; if SX, enter ***sx***.

```
% make rx
   make "F77=if77" "F77FLAGS=-g" gauss
   if77 -g -c gauss.f
   if77 -o gauss gauss.o  -node
```

The makefile itself is listed in Appendix A. Entering the **make** command creates the executable file *gauss*. As the makefile shows, to debug a program with IPD for RX nodes, compile it with the **-g** switch. Both CX and SX nodes require the **-B** switch.

INVOKING IPD

Next, get the nodes (with the iPSC **getcube** command) that you are going to use, and then invoke IPD. The example assumes that you get four RX nodes and invoke IPD from the same directory that contains the example's source and executable files. (For other node types, see the **getcube** command in the *iPSC®/2 and iPSC®/860 Programmer's Reference Manual*).

```
% getcube -t4rx
getcube successful: cube type 4m8rxn0 allocated
% ipd
 *** IPD Parallel Debugger, Release 3.3
 *** Copyright (c) 1990,1991 Intel Corporation
ipd >
```

Now you are ready to load your program.

LOADING THE PROGRAM AND SETTING THE DEBUG ENVIRONMENT

Now you are ready to load the program. Any time you load a program into IPD for debug, you must use the IPD **load** command; if the iPSC **load()** routine is part of your code, make sure you comment it out before you compile the program for debugging. This example, of course, contains no **load** commands.

```
ipd > load (all:0) gauss
 *** loading node program...
 *** nodes loaded, initializing...
 *** load complete
(all:0) >
```

After you have loaded the program, the prompt changes to the default context. The context defines the nodes and processes that will be affected by the command. In the **load** command above, you specified the context as ***(all:0)***; that is, all the nodes that you got with the **getcube** command, and process 0 on each of the nodes. This example assumes that you are using RX nodes, so only process 0 is valid. If you were using CX nodes, you might have more than one process per node.

You can also set the context with the **context** command. Specifying the context in a command overrides the default context.

It is usually useful to have a log file containing a record of your debugging session. Use the IPD **log** command to specify this. In the following command, you turn on logging and name the file *gauss.log*.

```
(all:0) > log -on gauss.log
```

Now list IPD's current status:

```
(all:0) > status

  Cube name: defaultname
  Cube type: 4m8rxn0
  More: on
  Log file: gauss.log
  Source search paths for gauss:
       .
```

The **status** command returns the name of your current group of nodes (here, *defaultname*), the types of nodes in the group (the *cubetype*, in this case, 4 RX nodes, starting at node 0, each with 8 Mbytes of memory), the status of the **more** command, the name (if any) of the current log file, and the source search paths. The IPD **more** facility controls screen scrolling; when you are listing something that fills up more than one screen, it pauses when the screen is full and waits for you to enter a keystroke. During interactive debugging, the IPD **more** is set to "on" by default.

Unless all of your source files are in your current directory, you need to add to the source search paths by using the **source** command so IPD can find your source files.

Another useful feature of the debug environment is the ability to define aliases. If you have created an *.ipdrc* file and included alias definitions to it, these definitions are included when you invoke IPD. Entering the **alias** command displays all previously defined aliases.

```
(all:0) > alias
Alias          Command String
=====          ==============
c              continue ; wait
s              step
ld             load (all:0)
```

Here, three aliases have been defined. For example, rather than entering ***load (all:0)*** in the load command earlier, you could have replaced that string with **ld**.

Another way to set up your debug environment so it is more comfortable for you to use is to use the **set** command to define a variable representing something that you would otherwise have to type out numerous times. For example, you could define the string ALL to represent the context string *(all:0)*, and another string S to represent the name of a procedure you will use a lot in this example.

```
(all:0) > set ALL (all:0)
(all:0) > set S shadow()
(all:0) > set
Variables      Variable String
=========      ===============
ALL            (all:0)
S              shadow()
```

Once you have defined these variables, you can use them on command lines in place of the original string; a dollar sign ($) is required in front of the variable name. For example, if the execution point were at the same place on all nodes, you could use the **display** command to display the variable *iam* in the routine *shadow()* on all nodes as follows:

```
disp $ALL iam
```

RUNNING THE PROGRAM

Now run the program. By entering the **run** command followed by the **wait** command, the program will run until all processes are complete, or some other event occurs (like a breakpoint).

```
(all:0) > run; wait
```

After some seconds, you see no sign that the program is going to return, so you correctly conclude that it is hung up somewhere. Press **<Del>** or **<Ctrl-C>**to cause control to return to the console, and IPD returns the following message:

```
*** interrupt...
```

This has not, however, halted execution, as you can see by entering the **process** command:

```
(all:0) > process
 Context        State        Reason    Src/Obj Name   Procedure        Location
 =======        =====        ======    ============   =========        ========
 *(all:0)       Executing               gauss
```

The asterisk in front of the context indicates that the status of the program has changed.

TRACKING THE BUG

To analyze the problem, you need to halt execution with the **stop** command.

```
(all:0) > stop
```

If you enter the **process** command now, more information is available. Besides aliasing, IPD also understands unique command abbreviations. Since no other commands begin with p, you can abbreviate the **process** command as follows:

```
(all:0) > p
 Context        State        Reason    Src/Obj Name   Procedure        Location
 =======        =====        ======    ============   =========        ========
*(all:0)        Stopped                 _flick.s       __flick        0x00023fe8
```

The asterisk in front of the context display indicates that the process status has changed again. The display also tells you that the program is stopped in the iPSC **flick()** routine. This is an indication that the program is blocked somehow. To find out where the problem is, you can use the **frame** command to trace the stack to provide a history of the routines called, starting with the most recent.

```
***** (0..2:0) *****
   __flick()     [_flick.s{}0x00023fe8]
   _gdhigh()     [_gdhigh.c{}0x000240f8]
   gdhigh_()     [gdhigh_.c{}0x0001e9dc]
   gauss()     [gauss.f{}#72]
   main()     [pgfmain.c{}0x000001ac]
 ***** (3:0) *****
   __flick()     [_flick.s{}0x00023fe8]
   msgwait_()     [msgwait_.c{}0x0002011c]
   shadow()     [gauss.f{}#209]
   gauss()     [gauss.f{}#58]
   main()     [pgfmain.c{}0x000001ac]
```

The names followed by the double parentheses () indicate the routines that the program has called. The names followed by the double braces {} indicate the object modules in which these routines reside. The routines that show line numbers were compiled for debug and are user routines. Those with memory addresses were not compiled for debug.

The stack trace tells you that nodes 0 through 2 (the 0..2 syntax indicates a range) went to **flick()** from **gdhigh()**, which is an iPSC message-passing routine, while node 3 is hung up waiting for a message (the iPSC **msgwait()** routine) in the subroutine *shadow*. In a flash of deductive reasoning, you decide that the problem is likely to be on node 3, in the *shadow* subroutine, near line number 174.

The **msgqueue** and **recvqueue** commands allow you to look at the message send and receive queues for misdirected messages.

```
(all:0) > msgq
 *** Unreceived messages in (all:0)
      Source              Destination          Type          Length
 ================     ================     ==========     ==========
 (0:0)                (2:0)                1000000002              8
 (2:0)                (3:0)                1000000001              8
 (1:0)                (3:0)                1000000002              8
(all:0) > recvq
 *** Unsatisfied receives posted in (all:0)
  Recv Posted By        Msg Type       Buffer Len        Handler
 ================     ==========     ==========     ==============
 (0:0)                1000000002              8
 (1:0)                1000000002              8
 (2:0)                1000000001              8
 (3:0)                       100            144
```

This tells you that, as you suspected, node 3 appears to be the one that is waiting for an unreceived message; there are two messages waiting to be received by node 3, while node 3 is looking to receive a message of a type that none of the other nodes have sent.

From the **frame** command earlier, you learned that node 3 is stuck in the iPSC **msgwait()** routine, which indicates that the unsatisfied receive is probably an **irecv()**. The **list** command allows you to list source code from the current execution point, a procedure, or a source line number. Here, list the whole *shadow* routine to find the unsatisfied **irecv()**:

```
(all:0) > list shadow()
  ***** (all:0) *****
  File: ./gauss.f
165:      subroutine shadow(ndim,n,totdim, iam,nbrnodes,range,a)
166:c
167:c  Shadow performs the exchange of neighbor columns into the shadow buffers
168:c
169:      integer          ndim,n,totdim
170:      integer          iam, nbrnodes, range
171:      double precision a(ndim,range+2)
172:
173:      integer          length,type,leftnode,rightnode,leftid,rightid
174:
175:      leftnode  = iam - 1
176:      rightnode = iam + 1
177:      length    = (totdim+2)*8
178:      type      = 100
179:
180:      if(iam.eq.0) then
181:c
182:c   If I am the leftmost node of the array (node 0) then only exchange
183:c   with the right (to the left is a boundary of the array)
184:c
185:         rightid = irecv(type, a(1,range+2), length)
186:         call csend(type, a(1,range+1), length, rightnode,0)
187:         call msgwait(rightid)
188:
189:      else if (iam .eq. nbrnodes) then
190:c
191:c   If I am the rightmost node of the array (highest numbered node) then
192:c   only exchange with the node to the left.
193:c
194:         leftid = irecv(type, a(1,1), length)
195:         call csend(type, a(1,2), length, leftnode,0)
196:         call msgwait(leftid)
197:
198:      else
199:c
200:c   Otherwise I am a node in the middle, so exchange with nodes to either side.
201:c
202:         leftid  = irecv(type, a(1,1), length)
```

```
203:          rightid = irecv(type,a(1,range+2), length)
204:
205:          call csend(type, a(1,2), length, leftnode,0)
206:          call csend(type, a(1,range+1), length, rightnode,0)
207:
208:          call msgwait(leftid)
209:          call msgwait(rightid)
210:
211:       endif
212:       end
```

This routine deals with the "shadow" columns that each node must share with the adjacent nodes, as described earlier. The variable *iam* is the current node number. If *iam* is node 0, it sends and receives only to the right (higher-numbered nodes) because there are no nodes to the left. If the node is the highest-numbered node (the right-most), it communicates only to the left. All other nodes receive from and send to nodes on either side, both left and right. Node 3 (context 3:0), because it is the node with the highest number, should be executing the middle block of the if statement, but the **frame** command indicated that node 3 was halted at line #174, in the last block.

You probably want to examine your variables to see what happened. With the **display** command, you can display the value of any variable or memory address. Look at the variables *iam* and *nbrnodes* to find out why the middle block is not being executed. Set the context with the **context** command to ensure that you are looking at node 3 only. Notice that in both these **display** commands (which use the **disp** abbreviation) the variable name is qualified with the name of the routine in which they reside. That is necessary here because node 3 (according to the previous **process** command) is inside the **flick**() routine, which is outside the scope of the variables you need to examine. It is important to be aware of the scope of your variables.

```
(all:0) > context (3:0)
(3:0) > disp shadow()iam

 ** gauss.f{}shadow()iam **
  ***** (3:0) *****
  iam = 3

(3:0) > disp (3:0) shadow()nbrnodes

 ** gauss.f{}shadow()nbrnodes **
  ***** (3:0) *****
  nbrnodes = 4
```

Sure enough, *iam* is not equal to *nbrnodes*. You begin to suspect that *nbrnodes* should, in fact, be *nbrnodes*-1.

IPD also gives you the **type** command, so you can quickly see the type of any variable.

```
(3:0) >  type shadow()nbrnodes
 ** gauss.f{}shadow()nbrnodes **
     INTEGER nbrnodes
```

To prove your suspicion about the cause of the bug in the program, you decide to use IPD's ability to set breakpoints and assign values to variables. The **break** command allows you to display current breakpoints or set breakpoints at procedures, source line numbers, instruction addresses, variables, and memory addresses. In this case, you reset the context to *all*, set a breakpoint at line 150, before the program enters the *if* statement, and then display the breakpoint (using **b**, the minimum abbreviation for **break**).

```
(3:0) > context (all:0)
(all:0) > break gauss.f{}#175
(all:0) > b
  (all:0)
Bp #  Type  File name   Procedure  Breakpoint Condition        Bp context
====  ====  =========   =========  ====================       ==========
   1  C Bp  gauss.f     shadow     Line 175                   (all:0)
```

Now you enter **run** followed by **wait**. The **run** command starts the program from the beginning; **wait** displays user process information when it reaches the breakpoint.

```
(all:0) > run ; wait
 Context        State        Reason    Src/Obj Name   Procedure        Location
 =======        =====        ======    ============   =========        ========
*(all:0)        Breakpoint   C Bp 1    gauss.f        shadow           Line 175
```

Now, once again, display the current value of *nbrnodes*, and then, on node 3, use the **assign** command to temporarily reassign the value of *nbrnodes* to 3, instead of 4.

```
(all:0) > disp nbrnodes

 ** gauss.f{}shadow()nbrnodes **
  ***** (all:0) *****
  nbrnodes = 4

(all:0) > context (3:0)
(3:0) > assign nbrnodes=3
(3:0) > disp nbrnodes

 ** gauss.f{}shadow()nbrnodes **
  ***** (3:0) *****
  nbrnodes = 3
```

Using the **step** command, you can single step through the next several steps (this uses the *s alias* shown earlier):

```
(3:0) > s
 Context        State        Reason    Src/Obj Name   Procedure        Location
 =======        =====        ======    ============   =========        ========
*(3:0)          Stepped                gauss.f        shadow           Line 176
(3:0) > s
 Context        State        Reason    Src/Obj Name   Procedure        Location
 =======        =====        ======    ============   =========        ========
*(3:0)          Stepped                gauss.f        shadow           Line 177
(3:0) > s
 Context        State        Reason    Src/Obj Name   Procedure        Location
 =======        =====        ======    ============   =========        ========
*(3:0)          Stepped                gauss.f        shadow           Line 178
```

Now, for the next two steps, if you add the **list** command with a count of 1, (*list,1*), you can display the lines you are stepping though.

```
(3:0) > s ; list,1
 Context        State        Reason    Src/Obj Name   Procedure        Location
 =======        =====        ======    ============   =========        ========
*(3:0)          Stepped                gauss.f        shadow           Line 180
  ***** (3:0) *****
  File: ./gauss.f
155:       if(iam.eq.0) then
(3:0) > s ; list
 Context        State        Reason    Src/Obj Name   Procedure        Location
 =======        =====        ======    ============   =========        ========
*(3:0)          Stepped                gauss.f        shadow           Line 194
  ***** (3:0) *****
  File: ./gauss.f
162:          leftid = irecv(type, a(1,1), length)
```

The second time you entered the list command, you did not need to specify a count of 1; IPD retains the count previously specified.

At last, your program is executing the correct block of the *if* statement. You can remove your breakpoint with the **remove** command (specifying the breakpoint number 1), and then continue executing the file with the **continue** command. If you use the **c** alias, defined as shown earlier, as **continue; wait**, it will execute until it finishes. The **continue** command, unlike the **run** command, continues from the current point, and retains the reassigned values of any variables.

```
(3:0) > context (all:0)
(all:0) > rem 1
(all:0) > c

C 0 200.0 200.0 200.0 200.0 200.0 200.0 200.0 200.0 200.0 200.0 200.0 200.0 200.0
200.0 200.0 200.0 200.0 200.0
C 1   0.0 100.0 139.4 157.8 167.7 173.5 177.1 179.1 180.1 180.1 179.2 177.1 173.6
167.8 157.9 139.4 100.0   0.0
C 2   0.0  60.4  99.9 124.2 139.6 149.4 155.7 159.4 161.2 161.2 159.4 155.7 149.4
139.6 124.3  99.9  60.5   0.0
C 3   0.0  41.9  75.4  99.7 116.9 128.8 136.8 141.7 144.0 144.0 141.7 136.8 128.8
116.9  99.8  75.4  42.0   0.0
                        .
                        .
                        .
```

The program cannot, however, complete execution because you have altered the value of *nbrnodes*. However, you have now identified your problem, and can fix it in the source code and recompile.

GETTING HELP

You can get help from IPD as you use it. For general help on IPD, you can enter the **help** command (the abbreviation is **h**), or you can just enter a question mark (**?**), as follows.

```
(all:0) > ?
COMMAND      ABBREV   DESCRIPTION
alias         al      Set or display command aliases
assign        as      Assign a new value to a program variable
break         b       Set or display breakpoints
context       conte   Set the default debug context
continue      conti   Continue stopped or breakpointed processes
disassemble   disa    Display an assembly listing of program code
display       disp    Display the value of a program variable
exec          exe     Read debugger commands from a file
exit          exi     Exit IPD - same as quit
frame         f       Display a stack traceback
help or ?     h       Display help information
kill          k       Terminate processes
list          li      List source code
load          loa     Load node programs
log           log     Record the debug session in a file
more          mo      Turn terminal scrolling on or off
msgqueue      ms      Display the queue of messages sent but not received
process       p       Display the current state of processes
quit          q       Exits IPD - same as exit
```

```
recvqueue     rec     Display the queue of receives posted but not satisfied
remove        rem     Remove breakpoints
run           ru      Restart processes without deleting breakpoints
set           se      Set or display debug variables
source        so      Set or display source directory search paths
status        sta     Display the current IPD status
step          ste     Execute the next source statement
stop          sto     Interrupt node processes
system or !   sy      Execute a UNIX shell command
type          t       Display the type of a variable
unalias       una     Delete aliases
unset         uns     Delete debug variables
wait          w       Wait for processes to stop
```

For more specific help on a given command, enter **help** or **?**, followed by the name of the command on which you wish help. This displays a brief description of the function, followed by the syntax. For example, you could request help on **break**.

```
(all:0) > ?break
 Display Breakpoint information:
     break [context]

 Set Code Breakpoint at procedure:
     break [context] [file{}]procedure() [-after count]

 Set Code Breakpoint at source line number:
     break [context] #line [-after count]
     break [context] file{}#line [-after count]
     break [context] [file{}]procedure()#line [-after count]

 Set Code Breakpoint at instruction address:
     break [context] address [-after count]

 Set Data Breakpoint on variable:
     break [context] [-access|-write] variable [-after count]
     break [context] [-access|-write] file{}variable [-after count]
     break [context] [-access|-write] [file{}]procedure()var [-after count]

 Set Data Breakpoint on memory address:
     break [context] [-access|-write] address [-after count]
```

EXITING IPD

You have found your problem and assured yourself that the program will run correctly after you fix the bug and recompile, so you can exit IPD with either the **quit** or the **exit** command.

```
(all:0) > quit
 *** IPD exiting...
%
```

For a complete description of each of the commands, refer to Chapter 3.

IPD COMMANDS 3

INTRODUCTION

This chapter provides detailed reference information on each of the IPD commands. The commands are listed in alphabetical order.

ALIAS ALIAS

Display or set aliases.

Syntax

alias [*alias_name* [*command_string*]]

Arguments

alias_name A string (the first character must be a letter) that you enter in place of a command.

command_string An IPD command string that is what you want the *alias_name* to represent. The command string is all of the text after the *alias_name* to the end of the **alias** command line, including any spaces, the pound sign (#), and semicolons.

Notes

Aliases are usually abbreviated names that save keystrokes. Input on a command line is matched first with the list of aliases before it is compared with the IPD command list. A recursive alias definition is flagged as an error when you use the alias.

The **alias** command with no arguments lists the current IPD aliases. When you issue the command with the *alias_name* argument alone, the command displays the definition of that *alias_name*. To define a new alias or redefine an existing alias, you must specify the *alias_name* followed by the *command_string* that defines it.

Use the **unalias** command to delete an alias. The **unalias** and **unset** commands are the only commands that cannot be aliased.

You may not use the **alias** command on the same IPD command line with another command.

Examples

1. Define an alias for the **step** command

```
(all:0) > alias s step
```

ALIAS *(cont.)* ALIAS *(cont.)*

2. Display the current aliases (this example assumes some aliases have been previously defined).

```
(all:0) > alias
  Alias      Command String
  ======     ==============
  x          exec -echo
  c          continue ; wait
  s          step
```

ASSIGN ASSIGN

Assign a value to a program variable or memory address.

Syntax

assign [*context*] [*file*{}][*procedure*()]*variable*[,*count*] = *value*
or
assign [*context*] [*-size_switch*] *address*[:*address*|,*count*] = *value*

Arguments

context Defines the nodes and processes that this command will affect. All nodes and processes executing this command must be running the same load module. If not, the command returns an error. If you do not specify a context, the default context (see the **context** command) applies to this command. Specify the context as follows:

({**all** | *nodelist*}:{**all** | *pidlist*})

variable The symbolic name of the variable to which you want to assign a value. If you specify an array name without a subscript, each element in the array is assigned the *value*. For Assembly language programs, you can use symbolic names if you have used the proper assembler directives to produce the symbolic debug information. For C or Assembly programs, IPD follows the C scoping rules. It first looks for the variable in the current code block, then in the current procedure. Next it looks for the variable in the static variables local to the current file and finally in the global program variables. To specify variables not in the current scope, prefix the variable name with the *procedure*() and/or *file*{} qualifiers.

Use language-specific syntax to specify a variable. For example, in Fortran you would specify an element of a two-dimensional array as ***a(1,1)***; in C, it would be ***a[1][1]***.

file{} The name of the source module in which the variable resides. To refer to a file other than the location of the current execution point, you must prefix the variable name with *file*{}. When you refer to a procedure, you can omit the *file*{} name unless there are duplicate procedure names, because IPD can find the source file from the symbol table information.

procedure() The name of the procedure in which the variable resides. You need to specify the procedure when the execution point is not in that procedure.

ASSIGN *(cont.)*

ASSIGN *(cont.)*

size_switch The *size_switch* indicate how many bytes (1, 2, or 4) should be assigned to the given address. You may only specify whole numbers when assigning to an address. Floats, complex, characters, and strings are not allowed. The *size_switch* can be one of the following:

byte
short
long

address A valid memory address to which you want to assign a value. You can specify a range of addresses either by following the initial address by a colon (:) and the final address, or by specifying a *count* of bytes; for example ***0x208:0x21b***.

count A positive integer used to denote a range of an array variable or address. First, you designate the beginning array element or address followed by a comma and the *count*; for example, ***x(10),10***, or ***0x208,8***. This allows you to assign the same value to multiple contiguous elements or addresses.

value The value that you want to assign. A value is converted, using C conversion rules, to the type of the variable being assigned. You must enclose a character value in single quotes ('*value*') and a string value in double quotes ("value")

Notes

The **assign** command changes the value of a variable for the current run. If you re-run the program with the **run** command, the values of all variables are reset to their original values

When specifying a variable, use the same language syntax convention as that of the source language. For example, to specify a Fortran element, you would use ***names(1)***; for a C element, ***names[1]***. For Assembler programs, you may use either C or Fortran syntax to assign a value to a memory address and it is assumed you know what you are doing.

To specify a range of addresses, you can use either the *address*:*address* form or the *address*,*count* form, not both. If you try to specify an array element outside the bounds of an array, you will get a warning message and IPD will go ahead and perform the requested assignment if the resulting address is valid.

A maximum of 1024 bytes may be assigned by one command.

You cannot assign to an entire structure; you must specify the member of a structure or union.

ASSIGN *(cont.)*

ASSIGN *(cont.)*

Examples

1. Assign a new value to the variable *nbrnodes* in the current scope, using a context different from the default.

```
(all:0) > assign (3:0) nbrnodes=3
(all:0) > disp nbrnodes

 ** gauss.f{}shadow()nbrnodes **
  ***** (3:0) *****
  nbrnodes = 3
```

2. Assign a new value to the variable *iam* in the procedure *shadow*(), using the current context.

```
(3:0) > assign shadow()iam = 2
(3:0) > display shadow()iam

 ** gauss.f{}shadow()iam **
  ***** (3:0) *****
  iam = 2
```

BREAK BREAK

Set a breakpoint or display current breakpoints.

Syntax

Display Breakpoint information:
break [*context*]

Set Code Breakpoint at procedure:
break [*context*] [*file*{}]*procedure*() [**-after** *count*]

Set Code Breakpoint at source line number:
break [*context*] [*file*{}][*procedure*()]**#***line* [**-after** *count*]

Set Code Breakpoint at instruction address:
break [*context*] *address* [**-after** *count*]

Set Data Breakpoint on variable:
break [*context*] {**-access** | **-write**} [*file*{}][*procedure*()]*variable* [**-after** *count*]

Set Data Breakpoint on a memory address:
break [*context*] {**-access** | **-write**} *address* [**-after** *count*]

Arguments

context Defines the nodes and processes that this command will affect. All nodes and processes executing this command must be running the same load module. If not, the command returns an error. If you do not specify a context, the default context (see the **context** command) applies to this command. Specify the context as follows:

({**all** | *nodelist*}:{**all** | *pidlist*})

file{} The name of the source module in which the procedure, line, or variable resides. To refer to a file other than the location of the current execution point, you must prefix the line number or variable name with *file*{}. When you refer to a procedure, you can omit the *file*{} name unless there are duplicate procedure names, because IPD can find the source file from the symbol table information.

BREAK *(cont.)*

BREAK *(cont.)*

procedure() The name of the procedure at which you wish to set the breakpoint, or the procedure in which the line or variable you are specifying resides. You only need to qualify a line number or variable name with the procedure name if the current execution point is outside that procedure. If you set a breakpoint at a procedure name, execution is halted just before the first executable line in the procedure, or at the entry point, if the first executable line does not have a number.

#line The source line number at which you want to set the breakpoint. The line number must be preceded with a pound sign (#). The statement must be executable. For example, you cannot set a breakpoint on a Fortran **FORMAT** statement, a C brace ({ or }), a comment, or an empty line. The process breaks just before executing the specified statement. To qualify the line number, use the *file*{} and/or *procedure*() arguments.

address The address can be either an instruction address or a memory address. When it is an instruction address, it must be a valid code address, and the process breaks just before executing the instruction at the *address*. When it is a memory address, it must be a data address, and you must specify either the **-access** or the **-write** switch, to define when the break occurs.

-access Specifies that a break will occur when the program accesses the specified variable or memory address (in i860 code, the break occurs before the access; in 386 code, the break occurs after the access). An access is either a read or a write. Use the **process** or **wait** command to determine where in your source code the access occurred which caused the break.

-write Specifies that a break will occur when the program writes the specified variable or memory address (in i860 code, the break occurs before the access; in 386 code, the break occurs after the access). Use the **process** or **wait** command to determine where in the source code the break occurred.

variable The *variable* is the symbolic name of the variable on which you want to set a data breakpoint. For Assembly language programs, you can use symbolic names if you have used the proper assembler directives to produce the symbolic debug information. For C or Assembly programs, IPD follows the C scoping rules. It first looks for the variable in the current code block, then in the current procedure. Next it looks for the variable in the static variables local to the current file and finally in the global program variables. To specify variables not in the current scope, prefix the variable name with the *procedure*() and/or *file*{} qualifiers.

BREAK *(cont.)* BREAK *(cont.)*

-after *count* In all forms of the **break** command, the *count* argument is a positive integer indicating the number of times this breakpoint is encountered before execution is halted. The default count is 1. For example, if you have a Fortran loop defined by the following

```
DO 10  I = 1,100
```

and you wish to break when the variable *I* equals 5, you would set the breakpoint inside the loop with an **-after** *count* of 5.

Display Format

Without any arguments the **break** command lists all breakpoints whose context has any node or process in the current default context or *context* argument. An example of the **break** command display is as follows:

```
  (all:0)
Bp #  Type  File name   Procedure  Breakpoint Condition        Bp context
====  ====  =========   =========  ====================        ==========
   1  C Bp  gauss.f     shadow     Line 150                    (all:0)
```

In the preceding display the first line shows the current context for the **break** command. The first column of the breakpoint display under the heading *Bp #* shows the breakpoint number. It is this number that is used as an argument to the **remove** command. The second column shows the type of breakpoint: *C Bp* for code breakpoint and *D Bp* for data breakpoint.

The column under the heading *File name* shows the name of the source file associated with the breakpoint. Under the heading *Procedure* is the name of the procedure where the code or variable is located. For global or static variables the *Procedure* field is set to <*global*> or <*static*>. To the right of the procedure name under the heading *Breakpoint Condition* is the condition under which the breakpoint will occur. The after clause will not be displayed unless the *count* is greater than 1.

The last column under the heading *Bp context* shows the breakpoint context. If the text overflows the *File name*, *Procedure* and *Breakpoint Condition* columns then the right most characters of the text will be truncated. However, if the context overflows the *Bp context* field, then the display for the breakpoint will be continued on the next line. This is denoted by blanks in all fields except the *Bp context* field which contains the continued breakpoint context.

BREAK *(cont.)*

BREAK *(cont.)*

Notes

In code running on i860-based RX nodes, data breakpoints based on a memory access occur before the memory access; on 386-based CX nodes, the breakpoints occur after the memory access.

If a single C statement consists of multiple source lines, set the breakpoint at the starting line number in 386 code, and at the ending line in i860 code.

When you set a breakpoint on a function such as the following:

```
break my_function()
```

the breakpoint is set on the first line of the function, if the function was compiled with symbols. If it was not compiled with symbols, or line number information has been stripped, the breakpoint is set on the function's entry point. As a result, if you set a breakpoint on a function, and then attempt to set a breakpoint on the first executable line of the same function, you will get a "breakpoint already exists" error.

Examples

1. Set a breakpoint at the procedure *shadow*() in the current source file for node 0, process 0 only.

```
(0:0) > b shadow()
```

2. Set a data breakpoint when the variable *iam* is accessed.

```
(3:0) > break -access iam
```

3. Set a breakpoint at line number 175 in the file *gauss.f*. Set the breakpoint so that the break occurs at the beginning of the tenth execution of the function for process 0 on nodes 1, 2, and 3.

```
(all:0) > break (1..3:0) gauss.f{}#175 -after 10
```

4. Set a breakpoint at line number 180 in the source file *gauss.f*.

```
(all:0) > break gauss.f{}#180
```

BREAK *(cont.)*

BREAK *(cont.)*

5. Display the current breakpoints. The **break** command displays those breakpoints that have a node or process in the current context. The display context is shown on the line before the table and the breakpoint context is shown in the right most column of the display.

```
(0:0) > break (all:0)

Bp #  Type  File name   Procedure  Breakpoint Condition      Bp context
====  ====  =========   =========  ====================      ==========
   1  C Bp  gauss.f     shadow     Call shadow               (0:0)
   2  D Bp  gauss.f     shadow     Access iam                (3:0)
   3  C Bp  gauss.f     shadow     Line 175 after 10         (1..3:0)
   4  C Bp  gauss.f     shadow     Line 180                  (all:0)
```

CONTEXT CONTEXT

Set the debug context. The debug context defines the set of processes that are the target of debug commands.

Syntax

context ({**all** | *nodelist*}:{**all** | *pidlist*})

Arguments

all	When used as the first argument (left side of the colon), **all** specifies all nodes where loaded processes reside. On the right side of the colon, **all** specifies all loaded processes under debug on the specified nodes.
nodelist	A single value indicates a single node. You can specify a range of nodes with the syntax *node1..node2*, where *node2* > *node1*. Specify a list of nodes by separating node numbers with commas, using the syntax *node, node, node...* The *nodelist* may include both a range of nodes and a list of nodes.
pidlist	A single value indicates a single NX/2 process ID (pid). For RX nodes, the only valid *pid* is 0. On CX nodes only, you can specify a range of pids with the syntax *pid1..pid2*, where *pid2* > *pid1*. The only valid pid for RX nodes is 0. Specify a list of CX processes by separating process numbers with commas, using the syntax *pid, pid, pid...pid*. The *pidlist* may include both a range of nodes and a list of nodes.

Notes

The initial default context is set with the first **load** command. You can change the default context with the **context** command. When you need to override the default context for a given command, you specify the context as part of the command syntax. This override is valid only for that command.

The **context** command can only refer to existing processes under debug. Only the **load** and **process** commands allow the specified context to refer to nodes and processes not under debug.

Examples

1. Set the context to process 0 on node 0.

```
(all:0) > context (0:0)
(0:0) >
```

CONTINUE CONTINUE

Continue execution of stopped or breakpointed processes in the default or specified context.

Syntax

continue [*context*]

Arguments

context The nodes and processes that you want this command to affect (see the **context** command). The default context is used if you do not specify one. Specify it as follows:

({**all** | *nodelist*}:{**all** | *pidlist*})

Notes

It is an error to use the **continue** command in a context containing running processes. You can use the **continue** command to start a process after it has been loaded rather than the **run** command (but you can redirect standard input only with the **run** command).

After the processes have been restarted, IPD will return control to you by issuing the next IPD prompt. If you wish to wait for a process to terminate or hit a breakpoint you must use the **wait** command.

Examples

1. Continue executing process 0 on node 1 when the current context is *(all:0)*.

```
(all:0) > continue (1:0)
(all:0) >
```

2. Continue all processes in the current context.

```
(all:0) > continue
(all:0) >
```

DISASSEMBLE DISASSEMBLE

Display machine code listing of process's instructions on i860 nodes.

Syntax

Disassemble from the current execution point:
disassemble [*context*] [,*count*]

Disassemble starting from an instruction address:
disassemble [*context*] *address*[:*address*|,*count*]

Disassemble starting from a procedure:
disassemble [*context*] [*file*{}]*procedure*()[,*count*]

Disassemble starting from a source line number:
disassemble [*context*] [*file*{}][*procedure*()]#*line*[: #*line* |,*count*]

Arguments

context Defines the nodes and processes that this command will affect. All nodes and processes executing this command must be running the same load module. If not, the command returns an error. If you do not specify a context, the default context (see the **context** command) applies to this command. Specify the context as follows:

({**all** | *nodelist*}:{**all** | *pidlist*})

If processes within the default or specified context are stopped at different locations in the load module, multiple disassembly lists will be displayed, one for each process with a unique execution point.

count An integer used to indicate the number of assembly instructions to disassemble. If *count* is positive, disassembly starts at the point specified and continues for *count* instructions. If negative, disassembly begins at *count*-1 instructions preceding the specified starting point and ends at this point. If you do not specify a *count*, the last *count* argument given to the **disassemble** command is used. Upon invoking IPD, the initial *count* is 50 instructions. One way to use the *count* argument is to specify a large count and use the IPD **more** function (see the **more** command) to browse through the instructions.

DISASSEMBLE (*cont.*)

DISASSEMBLE (*cont.*)

address The address at which to start the disassembly. You can specify a range of addresses by specifying *,count* following the *address*, or *address:address*.

file{} The name of the source module in which the procedure or line resides. To refer to a file other than the location of the current execution point, you must prefix the line number with *file{}*. When you refer to a procedure, you can omit the *file{}* name unless there are duplicate procedure names, because IPD can find the source file from the symbol table information.

procedure() The name of the procedure at which you wish to start disassembling, or the procedure in which the line you are specifying resides. You only need to qualify a line number with the procedure name if the current execution point is outside that procedure.

#line Specifies a source line number at which to start disassembly. The line number must be prefixed by a pound sign (#) and must exist in the symbol table debug information. You can specify a range of lines with the syntax ***#line:#line***. You must specify the range in ascending order.

Notes

The disassemble command allows you to display assembly language code for modules compiled to run on RX nodes. The target processes must be stopped to perform the disassembly. If they are not stopped, an error message is displayed.

If the specified procedure or address matches a source line number, that line number is displayed before the instructions. If there is no matching line number, the procedure name + address offset is shown, as in the following:

```
procedure() + 0x25.
```

DISASSEMBLE (*cont.*)

DISASSEMBLE (*cont.*)

Examples

1. Assume that the current context is *(all:0)* in a Fortran program. Disassemble 30 instructions, starting at the procedure *shadow*().

```
(all:0) > disa shadow(),30
  ***** (all:0) *****
  gauss.f{}shadow() + 0x0
00000b18: ec1f1001  orh         0x1001, r0, r31
00000b1c: e7ff1c00  or          0x1c00, r31, r31
00000b20: 1fe01801  st.l        fp, 0(r31)
00000b24: a3e30000  mov         r31, fp
00000b28: 1fe00805  st.l        r1, 4(r31)
00000b2c: 1c7f87fd  st.l        r16, -4(fp)
00000b30: 1c7f8ff9  st.l        r17, -8(fp)
00000b34: 1c7f97f5  st.l        r18, -12(fp)
00000b38: 1c7f9ff1  st.l        r19, -16(fp)
00000b3c: 1c7fa7ed  st.l        r20, -20(fp)
00000b40: 1c7fafe9  st.l        r21, -24(fp)
00000b44: 1c7fb7e5  st.l        r22, -28(fp)
  gauss.f{}shadow()#165
00000b48: 147cffe9  ld.l        -24(fp), r28
00000b4c: 1470fffd  ld.l        -4(fp), r16
00000b50: 139d0001  ld.l        r0(r28), r29
00000b54: 12110001  ld.l        r0(r16), r17
00000b58: 97be0002  adds        2, r29, r30
00000b5c: 0810f000  ixfr        r30, f16
00000b60: 08128800  ixfr        r17, f18
00000b64: 1c7ff7d9  st.l        r30, -40(fp)
00000b68: 96320001  adds        1, r17, r18
00000b6c: 4a1491a1  fmlow.dd    f18, f16, f20
00000b70: 1c7f97d1  st.l        r18, -48(fp)
00000b74: 1c7f8fe1  st.l        r17, -32(fp)
00000b78: 1c7f8fdd  st.l        r17, -36(fp)
00000b7c: 2c74ffd6  fst.l       f20, -44(fp)
  gauss.f{}shadow()#175
00000b80: 147cfff1  ld.l        -16(fp), r28
00000b84: 139d0001  ld.l        r0(r28), r29
00000b88: 97beffff  adds        -1, r29, r30
00000b8c: 1c7ff7cd  st.l        r30, -52(fp)
(all:0) >
```

DISPLAY DISPLAY

Display the value of the specified variable, memory address or processor registers.

Syntax

Display the value of variable in current scope of context:
display [*context*] [*-format_switch*] *variable*[,*count*] [*variable*[,*count*] ...]

Display the value of a global or static C variable:
display [*context*] [*-format_switch*] *file*{}*variable*[,*count*] [*file*{}*variable*[,*count*] ..]

Display the value of a local procedure variable:
display [*context*] [*-format_switch*] [*file*{}]*procedure*()*variable*[,*count*]
[[*file*{}]*procedure*()*variable*[,*count*] ...]

Display the value of a memory address:
display [*context*] *address*[:*address*|,*count*] ...

Display the contents of the processor registers:
display [*context*] **-register**

Arguments

context Defines the nodes and processes that this command will affect. All nodes and processes executing this command must be running the same load module. If not, the command returns an error. If you do not specify a context, the default context (see the **context** command) applies to this command. Specify the context as follows:

({**all** | *nodelist*}:{**all** | *pidlist*})

format_switch The *format_switch* overrides the symbol table information that would normally determine how a symbol's value would be printed. The *format_switch* can be one of the following:

alphanumeric	**double**
complex	**float**
dcomplex	**hexadecimal**
decimal	**octal**

real (the equivalent of the C float type)
string (see information in Notes)

DISPLAY *(cont.)*

DISPLAY *(cont.)*

variable — The symbolic name of the variable that you wish to display. If you specify an array name without a subscript, each element in the array is assigned the *value*. For Assembly language programs, you can use symbolic names if you have used the proper assembler directives to produce the symbolic debug information. For C or Assembly programs, IPD follows the C scoping rules. It first looks for the variable in the current code block, then in the current procedure. Next it looks for the variable in the static variables local to the current file and finally in the global program variables. To specify variables not in the current scope, prefix the variable name with the *procedure*() and/or *file*{} qualifiers.

count — A positive integer used to denote a range of an array variable or address. First, you designate the beginning array element or address followed by a comma and the *count*; for example, ***x(10),10***, or ***0x208,8***. This allows you to assign the same value to multiple contiguous elements or addresses.

file{} — The name of the source module in which the variable resides. To refer to a file other than the location of the current execution point, you must prefix the variable name with *file*{}. When you refer to a procedure, you can omit the *file*{} name unless there are duplicate procedure names, because IPD can find the source file from the symbol table information.

procedure() — The name of the procedure in which the variable resides. You need to specify the procedure when the execution point is not in that procedure.

address — A valid memory address the contents of which you want to display. You can specify a range of addresses either by following the initial address by a colon (:) and the final address, or by specifying a *count* of bytes; for example ***0x208:0x21b***. You may not mix the display of memory locations and variables. If the first display item is a memory address, the remaining display items must also be memory addresses.

-register — Display all of the processor registers.

DISPLAY *(cont.)*

DISPLAY *(cont.)*

Notes

When specifying a variable, use the same language syntax convention as that of the source language. For example, to specify a Fortran element, you would use ***names(1)***; for a C element, ***names[1]***. For Assembler programs, you may use either C or Fortran syntax to assign a value to a memory address and it is assumed you know what you are doing.

To specify a range of addresses, you can use either the *address*:*address* form or the *address*,*count* form, not both. If you try to specify an array element outside the bounds of an array, you will get a warning message and IPD will go ahead and perform the requested display if the resulting address is valid.

The display command can display only 1K bytes per node; if asked for more, it will display only the first 1K bytes.

You cannot display the elements of a structure by specifying the name of the structure, as you can with an array.

Use the **-string** switch when you want a C character array to be displayed as a null-terminated string. Otherwise, it is displayed as individual characters. For example, in a C program with a variable declared to be ***char name[5]***:

```
(1:0) > display name
name[0] = J
name[1] = o
name[2] = e
name[3] = y
name[4] =
(1:0) > display -string name
name = Joey
```

Examples

1. Display the variable named *iam* in process 0 on node 0.

```
(0:0) > display iam
  (0:0) size = 4
```

DISPLAY *(cont.)* DISPLAY *(cont.)*

2. Display 20 elements of the array *a*, starting at *a(1,4)*. To display the entire array, you would simply specify the array name.

```
(all:0) > disp (0:0) gauss()a(1,4),20

 ** gauss.f{}gauss()a(1,4) **
  ***** (0:0) *****
  a(1,4) = 0.0000000000000
  a(2,4) = 3.1250000000000
  a(3,4) = 5.4687500000000
  a(4,4) = 6.6406250000000
  a(5,4) = 7.1289062500000
  a(6,4) = 7.3120117187500
  a(7,4) = 7.3760986328125
  a(8,4) = 7.3974609375000
  a(9,4) = 7.4043273925781
  a(10,4) = 7.4064731597900
  a(11,4) = 7.4071288108826
  a(12,4) = 7.4073255062103
  a(13,4) = 7.4073836207390
  a(14,4) = 7.4074005708098
  a(15,4) = 7.4074054602534
  a(16,4) = 7.4074068572372
  a(17,4) = 7.4074072530493
  a(18,4) = 0.0000000000000
  a(19,4) = 0.0000000000000
  a(20,4) = 0.0000000000000
```

EXEC EXEC

Read and execute commands from the specified file.

Syntax

exec [-echo | -step] *filename*

Arguments

-echo	Causes the IPD commands in the specified file to be echoed to the terminal before they are executed. Along with the command, the current prompt will be echoed to show the default context. By default IPD does not echo commands.
-step	Causes the IPD command file to be executed line by line. The screen displays each IPD command before executing it (comment lines and blank lines are skipped). You can chose to execute the line by pressing the **<Return>** key. Then, the next command appears on the screen. If you no longer wish to step through the command file, use the keyboard interrupt to terminate the **exec** command.
filename	The name of the IPD command file.

Notes

When you specify **-echo**, a "++" is prefixed to each command line as it is displayed to denote that it is being read from a command file.

You may use the **exec** command inside the command file. Up to eight levels of **exec** nesting is supported. For every level of nested **exec** two "++" characters will be prefixed to the displayed command line if it is being echoed.

You may insert comments in command files by using a "#" followed by a blank and then the comment. All remaining characters to the end of the line are considered to be a comment including any semicolons. A "#" in the first character of a line denotes that the entire line is a comment.

When you invoke IPD, it executes the default command file called *.ipdrc* if you have created one in your home directory. The *.ipdrc* file is often used to define configuration information, such as a list of convenient aliases and command line variables. The commands in *.ipdrc* are not echoed.

EXEC *(cont.)*

EXEC *(cont.)*

Examples

1. Execute the command file *picf*, which consists of the following lines:

```
load (0:0) main
load (1..3:0) node
context (1..3:0)
break #84
break #90
```

When you execute this file, you get the following results:

```
ipd> exec -echo picf
ipd> ++ load (0:0) main
       *** loading node program...
       *** nodes loaded, initializing...
       *** load complete
(0:0)> ++ load (1..3:0) node
       *** loading node program...
       *** nodes loaded, initializing...
       *** load complete
(0:0)> ++ context (1..3:0)
(1..3:0) > ++ break #84
(1..3:0) > ++ break #90
(1..3:0) >
```

EXIT EXIT

Terminate a debug session and exit IPD.

Syntax

exit

Arguments

None

Notes

The **exit** command terminates an IPD session. It is equivalent to the **quit** command. Either command will terminate only those processes loaded by the debugger.

Examples

1. Exit IPD.

```
(all:0) > exit
  *** IPD exiting...
%
```

FRAME FRAME

Display the stack traceback(s) of the current or specified context.

Syntax

frame [*context*]

Arguments

context The nodes and processes that you want this command to affect (see the **context** command). The default context is used if you do not specify one. Specify it as follows:

({**all** | *nodelist*}:{**all** | *pidlist*})

Notes

The **frame** command displays a stack traceback, which lists the routines accessed, and the files in which those routines are located. If the routine was compiled to produce debug information, line numbers are displayed. If not, memory addresses are displayed.

If routines in the program were compiled without either the **-g** or **-ga** switch (for i860 code), these routines may not generate stack frames, and may cause missing routines in the stack traceback.

Parentheses (())following a name indicate a routine. Braces ({}) indicate a file.

The display is limited to a 256-deep traceback. If the number of procedures exceeds 256, only the top 256 are displayed.

FRAME *(cont.)*

FRAME *(cont.)*

Examples

1. In the following example, after a program was executed, it hung up, so execution was stopped. The **frame** command traces the stack to provide a history of the routines called, starting from the most recent. In this example, node 3 is found to have a different history than nodes 0, 1, and 2.

```
(all:0) > frame
     ***** (0..2:0) *****
       __flick()     [_flick.s{}0x00023fe8]
       _gdhigh()     [_gdhigh.c{}0x000240f8]
       gdhigh_()     [gdhigh_.c{}0x0001e9dc]
       gauss()     [gauss.f{}#72]
       main()     [pgfmain.c{}0x000001ac]
     ***** (3:0) *****
       __flick()     [_flick.s{}0x00023fe8]
       msgwait_()     [msgwait_.c{}0x0002011c]
       shadow()     [gauss.f{}#209]
       gauss()     [gauss.f{}#58]
       main()     [pgfmain.c{}0x000001ac]
```

HELP HELP

Display IPD commands and syntax.

Syntax

List all commands:
{ help | ? }

Obtain syntax help:
{ help | ? } *command*

Arguments

command	The *command* argument is any IPD command. The command line syntax will be displayed for this command.

Examples

1. Display the syntax for the **break** command. Entering ***help break*** would produce the same result.

```
(all:0) > ?break
 Display Breakpoint information:
     break [context]

 Set Code Breakpoint at procedure:
     break [context] [file{}]procedure() [-after count]

 Set Code Breakpoint at source line number:
     break [context] #line [-after count]
     break [context] file{}#line [-after count]
     break [context] [file{}]procedure()#line [-after count]

 Set Code Breakpoint at instruction address:
     break [context] address [-after count]
```

HELP *(cont.)* HELP *(cont.)*

```
Set Data Breakpoint on variable:
    break [context] [-access|-write] variable [-after count]
    break [context] [-access|-write] file{}variable [-after count]
    break [context] [-access|-write] [file{}]procedure()var [-after count]

Set Data Breakpoint on memory address:
    break [context] [-access|-write] address [-after count]
```

2. Display the IPD command summary list. Entering ? would produce the same result.

```
(all:0) > help
COMMAND      ABBREV   DESCRIPTION
alias         al      Set or display command aliases
assign        as      Assign a new value to a program variable
break         b       Set or display breakpoints
context       conte   Set the default debug context
continue      conti   Continue stopped or breakpointed processes
disassemble   disa    Display an assembly listing of program code
display       disp    Display the value of a program variable
exec          exe     Read debugger commands from a file
exit          exi     Exit IPD - same as quit
frame         f       Display a stack traceback
help or ?     h       Display help information
kill          k       Terminate processes
list          li      List source code
load          loa     Load node programs
log           log     Record the debug session in a file
more          mo      Turn terminal scrolling on or off
msgqueue      ms      Display the queue of messages sent but not received
process       p       Display the current state of processes
quit          q       Exits IPD - same as exit
recvqueue     rec     Display the queue of receives posted but not satisfied
remove        rem     Remove breakpoints
run           ru      Restart processes without deleting breakpoints
set           se      Set or display debug variables
source        so      Set or display source directory search paths
status        sta     Display the current IPD status
step          ste     Execute the next source statement
stop          sto     Interrupt node processes
system or !   sy      Execute a UNIX shell command
type          t       Display the type of a variable
unalias       una     Delete aliases
unset         uns     Delete debug variables
wait          w       Wait for processes to stop
```

KILL KILL

Terminate and remove processes in the current or specified context.

Syntax

kill [*context*] [**-force**]

Arguments

context The nodes and processes that you want this command to affect (see the **context** command). The default context is used if you do not specify one. Specify it as follows:

({**all** | *nodelist*}:{**all** | *pidlist*})

-force Kill process(es) without asking for verification.

Notes

Since **kill** is a potentially destructive command, the **kill** command will ask you if you are sure you want to kill the processes. You must enter a *y* to kill the process. Any other character(s) will be taken as a "no". Use the **-force** switch to force the kill without a user prompt.

A **kill** will terminate a process and release all IPD data structures associated with the process.

If you do not specify the **-force** switch on a **kill** command in a command file, it will attempt to read the verification response from the terminal.

Examples

1. Kill process 0 on node 0 when the current context is *(1..3:0)*.

```
(1..3:0) > kill (0:0)
     ***This command will delete all processes in (0:0).
     Are you sure you want to do this(y/n)? y
(1..3:0) >
```

2. Kill all processes in the current context without a question. In this case, there are no processes outside the current context, so when you do this there is no default context, as indicated by the "ipd" prompt.

```
(all:0) > kill -f
ipd >
```

LIST LIST

Display source code lines.

Syntax

List from current execution point:
list [*context*] [,*count*]

List starting from procedure:
list [*context*] [*file*{}]*procedure*()[,*count*]

List starting from a source line number.
list [*context*] [*file*{}][*procedure*()]**#***line*[**:** **#***line* | ,*count*]

Arguments

context Defines the nodes and processes that this command will affect. All nodes and processes executing this command must be running the same load module. If not, the command returns an error. If you do not specify a context, the default context (see the **context** command) applies to this command. Specify the context as follows:

({**all** | *nodelist*}:{**all** | *pidlist*})

The **list** command lists the source lines at the current execution point. If the default or specified context has processes stopped at different locations, multiple listings will be displayed, one for each process with a unique execution point.

count An integer used to indicate the number of lines of source code to list. If *count* is positive, listing starts at the point specified and continues for *count* instructions. If negative, listing begins at *count*-1 instructions preceding the specified starting point and ends at this point. If you do not specify a *count*, the last *count* argument given to the **list** command is used. Upon invoking IPD, the initial *count* is 50 lines. One method of using the *count* argument is to specify a large count and use the IPD **more** function (see the **more** command) to browse through the instructions.

LIST (*cont.*)

LIST (*cont.*)

file{} The name of the source module in which the procedure or line resides. To refer to a file other than the location of the current execution point, you must prefix the line number with *file{}*. When you refer to a procedure, you can omit the *file{}* name unless there are duplicate procedure names, because IPD can find the source file from the symbol table information.

procedure() The name of the procedure at which you wish to start listing, or the procedure in which the line you are specifying resides. You only need to qualify a line number with the procedure name if the current execution point is outside that procedure.

#line Specifies a source line number at which to start listing. The line number must be prefixed by a pound sign (#) and can be any source line in a source file. You can specify a range of lines with the syntax *#line:#line*. You must specify the range in ascending order.

Notes

If you wish to list from the current execution point, all processes in the context must be stopped to define this point. However, if you specify the starting point by specifying a line number or procedure, the state of the processes does not matter.

IPD finds the source files by searching the source directory search path defined by the **source** command. You cannot specify a path as part of the *file{}* qualifier.

If you specify a *file{}*, it must have been used in the compilation of a loaded module. Source files unrelated to any loaded module cannot be listed with the **list** command. Use the **system** command to access the UNIX **cat** or **vi** commands if you want to look at files of this kind. If you specify a source file that has the same name as a file used in compiling a program under debug, but is not the actual file used, this will not generate an error or warning, but will give faulty information. There is no way for the debugger to detect this situation.

Before each listing the **list** command will display a line showing the current context and the name of the source file that is being listed. If the source lines being listed are from a file that does not contain the current execution point, the context information is omitted, and only the file name is displayed prior to the listing.

Examples

1. Assume that the current context is *(1:0)*. Issue the **list** command after the main program encounters a code breakpoint to display each source line you are stepping through.

LIST (*cont.*)

LIST (*cont.*)

```
(1:0) > run ; wait
 Context        State        Reason    Src/Obj Name   Procedure        Location
 =======        =====        ======    ============   =========        ========
*(1:0)        Breakpoint   C Bp 1    gauss.f        shadow           Line 180

(1:0) > step ; list,1
 Context        State        Reason    Src/Obj Name   Procedure        Location
 =======        =====        ======    ============   =========        ========
*(1:0)         Stepped                 gauss.f        shadow           Line 180
  ***** (1:0) *****
  File: ./gauss.f
155:      if(iam.eq.0) then
(1:0) > step ; list
 Context        State        Reason    Src/Obj Name   Procedure        Location
 =======        =====        ======    ============   =========        ========
*(1:0)         Stepped                 gauss.f        shadow           Line 194
  ***** (1:0) *****
  File: ./gauss.f
162:         leftid = irecv(type, a(1,1), length)
```

2. List ten source lines starting at line number 180 (entering *list #180,17* would produce the same result).

```
(1:0) > list #180:#196
180:      if(iam.eq.0) then
181:c
182:c   If I am the leftmost node of the array (node 0) then only exchange
183:c   with the right (to the left is a boundary of the array)
184:c
185:         rightid = irecv(type, a(1,range+2), length)
186:         call csend(type, a(1,range+1), length, rightnode,0)
187:         call msgwait(rightid)
188:
189:      else if (iam .eq. nbrnodes) then
190:c
191:c   If I am the rightmost node of the array (highest numbered node) then
192:c   only exchange with the node to the left.
193:c
194:         leftid = irecv(type, a(1,1), length)
195:         call csend(type, a(1,2), length, leftnode,0)
196:         call msgwait(leftid)
(1:0) >
```

LOAD LOAD

Load an application onto the nodes and into the debugger.

Syntax

load *load_context filename*

Arguments

load_context The context for the **load** command. It is specified as follows:

({**all** | *nodelist*}: *pidlist*)

The **all** argument on the left- hand side of the context specifies all nodes. You may not specify all pids in the *load_context*. Refer to the description of the **context** command for more information on the *nodelist* and *pidlist* arguments. This argument is required, and sets the initial default context.

filename The load module file name of the program that you want to load. Specify the path name if the file is not in the current directory.

Notes

The **load** command loads an application onto the nodes for debugging under IPD. It also sets the default context. You can execute commands that require a context only after you execute the **load** command.

Use the **run** command to redirect standard input.

Examples

1. Load the file *gauss* on all nodes.

```
ipd > load (all:0) gauss
 *** loading node program...
 *** nodes loaded, initializing...
 *** load complete
(all:0) >
```

LOG

LOG

Turn debug session logging on or off, or display the name of the current log file.

Syntax

log [[-on] *filename* **| -off]**

Arguments

[-on] *filename*	Specifies the name of the file that will contain the debug log. The *filename* argument may be a complete or relative pathname. The **-on** is assumed by default if you specify a file name.
-off	Turns off logging to the current log file.

Notes

The **log** command with no arguments displays the name of the current log file. The arguments allow you to specify a log file name and turn on logging, or turn it off. Only one log file can be active at a time. If IPD is currently logging input and output and you use the **log** command to specify another log file, then the current log file will be closed and the new log file will be opened.

If you specify a log file that already exists then the file will be overwritten with the new log information.

Examples

1. Turn on logging to file *log007*.

```
(all:0) > log -on gauss.log
```

2. Display the name of the current log file.

```
(0:0) > log
        Log file: gauss.log
```

MORE MORE

Control scrolling of information on the IPD display.

Syntax

more [-on | -off]

Arguments

-on	Turn on the **more** functionality for terminal output. Whenever IPD output would scroll off the screen the display is stopped, a **more** prompt is issued on the last line of the display and IPD waits for user input before continuing with its output.
-off	Turn off the **more** functionality for terminal output. If IPD output is greater than the length of the terminal screen, then the output will scroll off the screen.

Notes

The **more** command allows you to control information scrolling on the display returned by IPD commands. The default **more** state depends upon IPD's standard input and standard output. If the standard input and standard output is a terminal then the default is "**more -on**". However, if IPD's standard input or standard output is a file then the default is "**more -off**".

To determine the current IPD **more** state, invoke the **more** command without arguments.

Examples

1. Turn on IPD's **more** functionality.

```
(all:0) > more -on
```

2. Display the current more state.

```
(all:0) > more
 More: on
```

MSGQUEUE MSGQUEUE

Display messages sent but not yet received.

Syntax

msgqueue [*context*] [*type* ...]

Arguments

context The nodes and processes that you want this command to affect (see the **context** command). The default context is used if you do not specify one. Specify it as follows:

({**all** | *nodelist*}:{**all** | *pidlist*})

type Only messages of the type specified by the *type* argument(s) will be displayed.

Notes

Note that **msgqueue** displays only the messages in the current or specified context.

The **msgqueue** command displays the messages that have been sent and have arrived on the specified node but have not yet been received by a process on the node. Use the **recvqueue** command to display which processes have posted receives that have not been satisfied.

Examples

1. Display all messages sent to process 0 that have not been received.

```
(all:0) > msgq
 *** Unreceived messages in (all:0)
      Source              Destination          Type          Length
 ================     ================     ==========     ==========
 (0:0)                (2:0)                1000000002              8
 (2:0)                (3:0)                1000000001              8
 (1:0)                (3:0)                1000000002              8
```

PROCESS PROCESS

Display information about user processes controlled by IPD.

Syntax

process [*context*] [**-change**] [**-loadfile**]

Arguments

context The nodes and processes that you want this command to affect (see the **context** command). The default context is used if you do not specify one. Specify it as follows:

({**all** | *nodelist*}:{**all** | *pidlist*})

-change Display only those processes that have changed state since the last process display.

-loadfile Display the load module name instead of the source file name. By default, the current source file and procedure are displayed for stopped processes, and the load module name is displayed for running processes.

Notes

The process command provides information about the processes running under IPD. The following is an example of the **process** display:

```
   Context     State          Reason       Src/Obj name   Procedure  Location
 ==========  ======         ======       ==========     ==========  ========
   (1,2:0)     Breakpoint   C Bp 1       node.f         scan()      line #53
 * (3..5:0)   Executing                  node
   (11:0)      Breakpoint   C Bp 2       assem.s        test()      0x000456
```

If an asterisk (*) appears in the first column of the process display, then the processes on that line of the display have changed state since the last process display. The first column of information under the heading *Context* shows the nodes and processes in context format (see the **context** command for more information). If the *Context* field overflows, the process command splits the information into multiple lines. The second column under the heading *State* shows the current state of the processes. A process can be in one of seven states: *Initial*, *Executing*, *Breakpoint*, *Stepping*, *Stepped*, *Stopped* or *Terminated*. For processes in the Breakpoint and Terminated state, the next column under the heading *Reason* gives further information on the process's state. For processes at a breakpoint, the *Reason* column shows the breakpoint type and breakpoint number (see the **break** command for more information on breakpoint type).

PROCESS *(cont.)* PROCESS *(cont.)*

For Terminated processes the *Reason* column gives a short description on why the process has terminated. For all process states except Executing and Stepping the next three columns show the source file name, procedure name and location of the process. You may use the **-loadfile** switch to specify that the load module should be displayed instead of the file name and procedure. For processes in the Executing and Stepping states, the column under the heading *Src/Obj name* shows the name of the object file that was loaded.

Notes

The **wait** and **step** commands perform an implicit **process** command upon returning control to the user.

Examples

1. Display process information. Two source modules are loaded: *node* and *main*. Notice that the *node* program is loaded but has not yet executed on nodes 1, 2 and 3.

```
(all:all) > process

   Context     State          Reason       Src/Obj name   Procedure  Location
  ==========  ======         ======       ==========     ==========  ========
  (0:0)       Breakpoint     C Bp 1       main.f         MAIN        line #84
  (1..3:0)    Initial                     node.f         MAIN        line #86
```

2. Continue the execution of the *node* program, hitting another breakpoint. The **wait** command performs an implicit **process** command to display the process information. Notice that the *node* program has executed and is now stopped at a breakpoint. The leading asterisk (*) indicates that the state has changed since the last time process was displayed.

```
(all:all) > context (1..3:0)
(1..3:0) > continue
(1..3:0) > wait
   Context     State          Reason       Src/Obj name   Procedure  Location
  ==========  ======         ======       ==========     ==========  ========
* (1..3:0)    Breakpoint     C Bp 3       node.f         MAIN        line #93
```

QUIT QUIT

Terminate a debug session and exit IPD.

Syntax

quit

Arguments

None

Notes

The **quit** command terminates an IPD session. It is equivalent to the **exit** command. Either command will terminate only those processes loaded by the debugger.

Examples

1. Exit IPD.

```
(all:0) > quit
  *** IPD exiting...
%
```

RECVQUEUE RECVQUEUE

Display pending receives.

Syntax

recvqueue [*context*] [*type* ...]

Arguments

context The nodes and processes that you want this command to affect (see the **context** command). The default context is used if you do not specify one. Specify it as follows:

({**all** | *nodelist*}:{**all** | *pidlist*})

type Only messages of the type specified by the *type* argument(s) will be displayed.

Notes

The **recvqueue** command displays receive requests posted by processes in the default or specified context.

The **recvqueue** command displays receives that have been posted but not satisfied. Use the **msgqueue** command to display messages that have been sent but not received.

Processes that have receives posted are not necessarily blocked. The process may have posted one or more asynchronous receives (i.e., **irecv()** or **hrecv()**) and continued executing. If you have posted an iPSC **hrecv()** call, which requires a handler, the name of the handler is listed under the final column.

Examples

1. Display all receives that have not been satisfied by an incoming message.

```
(all:0) > recvq
 *** Unsatisfied receives posted in (all:0)
  Recv Posted By        Msg Type      Buffer Len        Handler
 ================     ==========     ==========     ==============
 (0:0)                1000000002              8
 (1:0)                1000000002              8
 (2:0)                1000000001              8
 (3:0)                       100            144
```

REMOVE REMOVE

Remove breakpoints.

Syntax

remove [*context*] {*breakpoint_number* [*breakpoint_number* ...] | **-all**}

Arguments

context The nodes and processes that you want this command to affect (see the **context** command). The default context is used if you do not specify one. Specify it as follows:

({**all** | *nodelist*}:{**all** | *pidlist*})

breakpoint_number
The number of the breakpoint to be removed. To determine the breakpoint number, use the **break** command without any arguments.

-all Remove all breakpoints in the default or specified context.

Notes

The **remove** command removes the specified breakpoints or all breakpoints in the default or specified context.

You may remove nodes from a breakpoint context by specifying the **remove** command with the desired nodes in the context argument. IPD will not remove the breakpoint but will only remove the nodes from the breakpoint context. Only when all the nodes have been removed from the breakpoint context will the breakpoint be removed.

When you remove a breakpoint, its breakpoint number is no longer valid; but the number is not used again in the same debug session.

REMOVE *(cont.)* REMOVE *(cont.)*

Example

1. Display all current breakpoints, remove breakpoints 1 and 2 on *(0:0)*, then redisplay the breakpoints.

```
(0:0) > break (all:0)

Bp #  Type  File name   Procedure  Breakpoint Condition        Bp context
====  ====  =========   =========  ====================       ==========
   1  C Bp  gauss.f     shadow     Call shadow                (0:0)
   2  D Bp  gauss.f     shadow     Access iam                 (0:0)
   3  C Bp  gauss.f     shadow     Line 175 after 10          (1..3:0)
   4  C Bp  gauss.f     shadow     Line 180                   (all:0)
(0:0) > remove (0:0) 1 2
(0:0) > b (all:0)

Bp #  Type  File name   Procedure  Breakpoint Condition        Bp context
====  ====  =========   =========  ====================       ==========
   3  C Bp  gauss.f     shadow     Line 175 after 10          (1..3:0)
   4  C Bp  gauss.f     shadow     Line 180                   (all:0)
```

RUN RUN

Start execution of processes in the current or specified context.

Syntax

run [*context*] [*<infile*]

Arguments

context The nodes and processes that you want this command to affect (see the **context** command). The default context is used if you do not specify one. Specify it as follows:

({**all** | *nodelist*}:{**all** | *pidlist*})

<infile The name of a file containing input to the program you are running under IPD. By specifying an input file, the program looks to the file, rather than standard input, when it needs input as it is executing.

Notes

The **run** command will always execute the program from its beginning. Use the **continue** command to continue execution of a stopped or breakpointed process.

If you assign a value to a variable, the **run** command resets it to the initial value. You must use either the **continue** or the **step** command to retain the assigned value of a variable.

It is an error to use the **run** command in a context containing running processes.

The operation of the **run** command depends on whether or not the program is in the initial state. A program that is loaded but has never been executed is in the initial state, as the **process** command will indicate. In this case, the **run** command simply executes the program as does the **continue** command (except that **run** allows you to redirect the standard input.)

RUN *(cont.)* RUN *(cont.)*

If you have already executed the program, the **run** command does the following:

1. Kills the current program, which flushes (deletes) all messages currently in the cube
2. Reloads the program
3. Resets all of the user breakpoints
4. Starts executing the program.

IPD executes node processes asynchronously. Node program output to the terminal is printed either before IPD issues a command prompt or during the execution of the **wait** command. Node program keyboard input is processed only during execution of the **wait** command.

Use the **wait** command to wait for all processes in a context to stop.

Examples

1. Run a program from its beginning. Issue a **wait** command so the prompt will not return until it reaches a breakpoint, finishes running, or you issue a keyboard interrupt.

```
(all:0) > run; wait
```

SET SET

Set or display command line variables.

Syntax

List all set variables:
set

List variable definition:
set *variable_name*

Define new or redefine old variable:
set *variable_name string*

Arguments

variable_name The symbolic name of the command line variable you are defining.

string The *string* argument is any and all text after the *variable_name* to the end of the set command line. This includes the pound sign (#), spaces, and semicolons. You may build a command line variable from other command line variables by specifying a previously defined *variable_name* prefixed with a dollar sign ($) in the *string*.

Notes

The set command allows you to set or display command line variables. Command line variables are expanded when they are used. A recursive variable definition generates an error when it is used.

To use a command line variable in a command, precede the *variable_name* with a dollar sign ($). The *variable_name* must be followed by a space to separate it from the next argument on the command line. If you do not wish a space after the *variable_name*, enclose it in braces:

${*variable_name*}

Use the **unset** command to delete command line variables. The **unalias** and **unset** commands are the only commands that can not be aliased.

You may not use the **set** command on the same command line with another command.

SET *(cont.)*

SET *(cont.)*

Examples

1. Define the command line variable *myproc* as *(1..3:0)*. Then, use this command line variable in the **context** command.

```
(0:0) > set myproc (1..3:0)
(0:0) > context $myproc
(1..3:0) >
```

2. Display the current command line variables.

```
(1..3:0) > set
  Variables    Variable String
  =========    ===============
  myproc         (1..3:0)
```

3. Set *x* to the command line variable *long_name[104]*. Alias **an** to assign in the *$myproc* context. Use **an** to assign the variable *x*.

```
(1..3:0) > set x long_name[104]
(1..3:0) > alias an assign $myproc
(1..3:0) > an $x = 100
```

SOURCE SOURCE

Set or display the current source directory search path.

Syntax

Display source directory search path:
source [*filename*]

Set new source directory search path:
source *filename directory* [*directory*] ...

Add directories to source directory search path:
source *filename* **-add** *directory* [*directory*] ...

Remove directories from source directory search path:
source *filename* **-remove** *directory* [*directory*] ...

Arguments

filename	The name of a previously loaded executable file.
directory	Path name for the directory that contains the application source files.
-add	Add the specified directories to the source directory search path. The directories will be appended to the end of the search path list.
-remove	Remove the specified directories from the source directory search path list.

Notes

The **source** command with no arguments displays the search paths for all loaded modules. If you specify a filename, the search path for that file is displayed. When replacing, adding, or deleting directories from the search paths, you must specify a load module filename so IPD can associate the search path list with the correct executable file.

The directories are listed in the order that IPD uses to search for a source file for the **list** command. The default directory search path assigned at load time is the current directory (.). A directory must exist and be readable to be added to the search list. If a non-existent directory is specified in a list of directories to be added, an error message is displayed, and only the directories that precede the non-existent directory in the list will be added.

SOURCE *(cont.)* SOURCE *(cont.)*

Examples

1. Display the current source directory search path for the previously loaded program *gauss*. Try to list the source file, but an error is returned when it cannot be found in the current directory. Add */usr/you/Fpi/* to the source directory search path and list the node program.

```
(all:0) > source node
  Source search paths for gauss:
        .
(all:0) > source node -add /usr/you/Fpi
(all:0) > source node
   Source search paths for gauss:
        .
   Source search paths for node:
       /usr/you/Fpi
(all:0) > list,10
***** (all:0) ***** node.f
57       program node
58
59       include 'fcube.h'
60
61      integer SIZETYPE, INITTYPE, PARTTYPE, MSGSIZE, CUBESIZE,
62     >                  HOST, HOSTPID, APPLPID, DOUBLESIZE
63
64       integer*4 worknodes, mynode, pid, size
65       integer*4 basicpoints, extrapoints, mypoints, i, j
66       integer*4 starttime, points
(all:0) >
```

STATUS STATUS

Display the debugger status.

Syntax

status

Arguments

None

Notes

The **status** command displays the name of the cube in which IPD is running, the types of nodes in the cube, the state of the IPD **more** command (it is on by default), the name of the log file (if any) to which the output from the debug session is being written, and the list of source search paths for each load module under debug.

Examples

1. Get current IPD status.

```
(all:0) > status

  Cube name: defaultname
  Cube type: 4m8rxn0
  More: on
  Log file: gauss.log
  Source search paths for gauss:
       .
```

STEP STEP

Single-step through the processes in the current or specified debug context.

Syntax

Step through source line(s):
step [*context*] [**-call**] [,*count*]

Arguments

context The nodes and processes that you want this command to affect (see the **context** command). The default context is used if you do not specify one. Specify it as follows:

({**all** | *nodelist*}:{**all** | *pidlist*})

-call Treat user-defined subroutine and function calls as single statements. If **-call** is not specified, the routine is entered and its statements stepped through.

,*count* The number of source lines or instructions to step through before returning control to the user. The default *count* is one source line or machine instruction.

Notes

The step command allows you to execute your program one source line at a time. Any procedures compiled without line number information are treated as if the command were **step -call**, even if you did not specify **-call**.

Single-stepping is synchronous; the **step** command will not return until all processes in its context have stepped. If your program hangs during the **step** command, then use the interrupt signal to regain the IPD prompt. At this point the current state of the process is running. Use the **stop** command to stop the process.

Upon returning control to the user, IPD will invoke the **process** command to display process information.

STEP (*cont.*) STEP (*cont.*)

Examples

1. In the program *gauss*, stopped at line number 178 on node 1, step to the next line, and list it.

```
(1:0) > s
 Context        State        Reason    Src/Obj Name   Procedure        Location
 =======        =====        ======    ============   =========        ========
*(1:0)          Stepped                gauss.f        shadow           Line 178
(1:0) > s ; list,1
 Context        State        Reason    Src/Obj Name   Procedure        Location
 =======        =====        ======    ============   =========        ========
*(1:0)          Stepped                gauss.f        shadow           Line 180
  ***** (1:0) *****
  File: ./gauss.f
180:      if(iam.eq.0) then
```

STOP STOP

Stop program execution in the current or specified context.

Syntax

stop [*context*]

Arguments

context The nodes and processes that you want this command to affect (see the **context** command). The default context is used if you do not specify one. Specify it as follows:

({**all** | *nodelist*}:{**all** | *pidlist*})

Notes

The **stop** command stops program execution. Stopping program execution when you do not have an IPD prompt requires sending an interrupt signal (i.e., **<Del>** or **<Ctrl-C>**) to allow IPD to regain control before issuing the **stop** command.

Examples

1. When you run the Fortran *gauss* example, it blocks at its first receive. Send an interrupt signal, then issue the **process** command. This indicates the program is still executing. Issue the **stop** command, and then the **process** command again.

```
(all:0) > run; wait
 *** interrupt...
(all:0) > process
 Context        State        Reason    Src/Obj Name   Procedure        Location
 =======        =====        ======    ============   =========        ========
*(all:0)        Executing               gauss
(all:0) > stop
(all:0) > p
 Context        State        Reason    Src/Obj Name   Procedure        Location
 =======        =====        ======    ============   =========        ========
*(all:0)        Stopped                 _flick.s       __flick
```

SYSTEM SYSTEM

Execute a shell command.

Syntax

system *shell_command*
or
! *shell_command*

Arguments

shell_command A string consisting of UNIX shell commands (not an IPD command) to be executed. All text following the **!** or **system** to the end of the line, including any semicolons, the pound sign (#), and spaces, is part of the *shell_command.*

Notes

Use either the **system** or **!** command to execute a UNIX shell command from within IPD. The *command* is **NOT** interpreted by IPD. All *command* text to the end of the **system** command line is passed directly to the **sh** (i.e., Bourne shell). You may not use the **system** command on the same IPD command line as any other command.

Do not use commands that might affect the cube under debug; (e.g., ***system waitcube*** and ***system relcube -a***).

If a log file is active, output from this command is written in the log file.

Examples

1. Issue the shell command **ls -l** from within IPD.

```
(all:0) > system ls -l /usr/ipsc/examples/fortran/gauss
total 23
-r--r--r--   1 root     other       1413 Mar 30 21:03 README
-r--r--r--   1 root     other        187 Mar 30 21:03 gauss.f
-r--r--r--   1 root     other        475 Mar 30 21:03 makefile
(all:0)>
```

TYPE TYPE

Display the type of variables in the current or specified context.

Syntax

Display type of variable in current scope of context:
type [*context*] [**-struct**] *variable* [*variable*] ...

Display type of global or static C variable:
type [*context*] [**-struct**] *file*{}*variable* [*variable*] ...

Display type of local procedure variable:
type [*context*] [**-struct**] [*file*{}]*procedure*()*variable* [*variable*] ...

Arguments

context — Defines the nodes and processes that this command will affect. All nodes and processes executing this command must be running the same load module. If not, the command returns an error. If you do not specify a context, the default context (see the **context** command) applies to this command. Specify the context as follows:

({**all** | *nodelist*}:{**all** | *pidlist*})

-struct — Displays type information for the members of a structure or union variable.

variable — The symbolic name of the variable for which type information is to be displayed. For Assembler programs, variable names can be used if the proper assembler directives have been used to produce the symbolic debug information. For C or Assembler programs, IPD follows the C scoping rules. It looks first for the variable in the current code block, then the current procedure, then in the static variables local to the current file, and finally, in the global program variables.

file{} — The name of the source module in which the variable resides. To refer to a file other than the location of the current execution point, you must prefix the variable name with *file*{}. When you refer to a procedure, you can omit the *file*{} name unless there are duplicate procedure names, because IPD can find the source file from the symbol table information.

TYPE *(cont.)*

TYPE *(cont.)*

procedure() The name of the procedure in which the variable resides. You need to specify the procedure when the execution point is not in that procedure.

Examples

1. Determine the type of the Fortran variable *tms* in process 0 of node 0.

```
(all:0) > type (0:0) tms
integer*4 tms
(all:0) >
```

2. Determine the type of the C structure variable *msg*.

```
(0:0) > type msg
struct msg_type msg;
(0:0) > type -struct msg
struct msg_type  {
        double a;
        double b;
        int points;
} msg;
(0:0) >
```

UNALIAS UNALIAS

Delete previously defined aliases.

Syntax

unalias {*alias_name* [*alias_name* ...] | **-all** }

Arguments

alias_name	A string that was chosen as an alias for an IPD command using the **alias** command.
-all	Remove all currently defined aliases.

Notes

The **unalias** command removes a previously defined alias. Use the **alias** command without arguments to display the current list of alias names. The **unalias** command cannot be aliased.

Examples

1. Remove the alias **ct**.

```
(all:0) > alias
   Alias      Command String
    ======     ==============
    ct         context
(all:0) > unalias ct
(all:0) > alias
    Alias      Command String
     ======     ==============
(all:0) >
```

UNSET UNSET

Delete previously defined command line variables.

Syntax

unset {*variable_name* [*variable_name* ...] | **-all** }

Arguments

variable_name	The symbolic name of the command line variable you are deleting. Do not precede the *variable_name* with a $.
-all	Remove all currently defined command line variables.

Notes

The **unset** command removes the definitions of command line variables previously defined with the **set** command. Use the **set** command with no arguments to display a list of the current command line variable names. The **unset** command cannot be aliased.

Examples

1. Delete the command line variable *myproc*.

```
(0:0) > set
       Variables   Variable String
       =========   ===============
       myproc        (1..3:0)
(0:0) > unset myproc
(0:0) > set
        Variables   Variable String
        =========   ===============
(0:0) >
```

WAIT WAIT

Wait until all processes within the context have stopped running.

Syntax

wait [*context*]

Arguments

context The nodes and processes that you want this command to affect (see the **context** command). The default context is used if you do not specify one. Specify it as follows:

({**all** | *nodelist*}:{**all** | *pidlist*})

Notes

The **wait** command causes IPD to return the prompt only when all processes within the context are not in a Running, Executing, or Stepping state (see the **process** command for process state information).

A program's output written to *stdout* appears between IPD commands and is not intermixed with IPD output. If the program needs to read from the terminal, you must use the **wait** command to process the read requests. This is analogous to the **waitcube** functionality (see the *iPSC®/2 and iPSC®/860 Programmer's Reference Manual* for more information on iPSC commands). If you wish to redirect the program's standard input, use the **run** command.

After a **run** or **continue** command, IPD will immediately issue a prompt. If you wish to wait for a process to hit a breakpoint or terminate, use the **wait** command.

If you no longer wish to wait for all the processes to stop running, use the interrupt signal to regain the IPD prompt.

Upon returning control to the user, IPD will invoke the **process** command to display the process information.

WAIT *(cont.)*

WAIT *(cont.)*

Examples

1. Issue a **run** command followed by a **wait**. When all the processes have stopped running, the **wait** command will perform a **process** command and return a prompt.

```
(all:0) > run ; wait
 Context        State        Reason    Src/Obj Name   Procedure        Location
 =======        =====        ======    ============   =========        ========
*(all:0)        Breakpoint   C Bp 1    gauss.f        shadow           Line 150
(all:0) >
```

SOURCE CODE FOR THE IPD EXAMPLE A

MAKEFILE

The following is the makefile used in Chapter 2. It included in the directory */usr/ipsc/examples/flipd*, along with the program example, which is in the same directory.

```
#
# makefile : 8.3 91/03/12 11:00:19
#
# This file is used to compile and link the gauss.f for the ipd example program.
#
.f.o:
   $(F77) $(F77FLAGS) -c $<

help:
   @echo
   @echo "You must specify the type of node you wish to build a node"
   @echo "executable for, choose one of the following:"
   @echo
   @echo "     make cx    (for 386 nodes with 387 coprocessors)"
   @echo "     make sx    (for 386 nodes with SX coprocessors)"
   @echo "     make rx    (for i860 nodes)"
   @echo

cx:
   make "F77FLAGS=-B" gauss

sx:
   make "F77FLAGS=-sx -B" "FLDFLAGS=-sx" gauss

rx:
   make "F77=if77" "F77FLAGS=-g" gauss

gauss:gauss.o
   $(F77) -o gauss gauss.o $(FLDFLAGS) -node

clean:
   rm gauss gauss.o
```

README

There is also, in the same directory, a README file that describes how to use the example.

```
README README : 1.1 91/03/12 10:18:06

************** /usr/ipsc/examples/f/ipd/README ******************


This directory contains the Fortran source and make files for the ipd
example program.

This example program solves a LaPlace Equation in two dimensions using
an iterative Gauss-Seidel method. It uses column-wise data decomposition
of a two dimensional array, and on each cycle of the iteration it
averages each array element with its 4 nearest neighbors. To obtain the
nearest neighbors from columns that reside on different nodes, the
"boundary columns" are exchanged on each cycle, using "shadow buffers"
to receive copies of the neighbor columns from the previous cycle.

To run this example you need to
   (1) make executables from source,
   (2) get the cube,
   (3) invoke ipd, and
   (4) follow the instructions in the IPD manual for the tutorial.

For example:

   1) Create the node executable (gauss) from the source file
      gauss.f to run on RX nodes:

   make rx

   2) Get a 2 dimensional cube (4 nodes) named "sandy":

   getcube -c sandy -td2

   3) Invoke IPD:

   ipd

   4) Execute the tutorial commands.


You should copy this example directory to your own user directory before
modifying the source or make files.
```

GAUSS.F

The source code for the example program follows.

```
      program gauss
c
c   gauss.f : 8.1 91/02/28 10:04:06
c
c   Example program that solves LaPlace Equation in two dimensions using
c   an iterative Gauss-Seidel method. It uses column-wise data decomposition
c   of a two dimensional array, and on each cycle of the iteration it
c   averages each array element with its 4 nearest neighbors. To obtain the
c   nearest neighbors from columns that reside on different nodes, the
c   "boundary columns" are exchanged on each cycle, using "shadow buffers"
c   to receive copies of the neighbor columns from the previous cycle.
c
c
      double precision a(34,34), resid, residtemp, residprev, aold,eps
      double precision residtest, amax
c
c   totdim is the dimension of the square global grid.  It must be
c   a multiple of the number of nodes (a power of 2)
c
c   ndim is the dimension of a.  It must be at least totdim+2
c   but may be more
c
c   range is the dimension of the data owned by a node
c
c   n is the dimension of the node's grid, which includes shadow
c   buffers
c
      integer          totdim, ndim, n, iam, nbrnodes, range
      integer          xmin, xmax, ymin, ymax, maxiter

c
c  -- Initialize --
c
      ndim     = 34
      totdim   = 16
      iam      = mynode()
      nbrnodes = numnodes()
      range    = totdim/nbrnodes
      n        = range + 2
```

```
      xmin      = 2
      xmax      = range + 1
      ymin      = 2
      ymax      = totdim + 1

      maxiter = 200
      eps    = 1.0D-05
      resid = 0.0D0
      residtest = 0.0D0
      amax      = 0.0D0
      residprev = 0.0D0

      call inita(iam,nbrnodes,totdim,ndim,n,a)
c
c  --  Perform the Calculation --
c
      do 101 k = 1,maxiter
      call shadow(ndim,n,totdim, iam,nbrnodes,range,a)

      do 100 i = ymin, ymax
       do 100 j = xmin, xmax
        aold      = a(i,j)
        a(i,j)    = ( a(i+1,j) + a(i-1,j) + a(i,j+1) + a(i,j-1) )/4.0
        residtemp  = dabs(a(i,j) - aold)
        if(residtemp .gt. resid) then
           resid = residtemp
           amax  = aold
        endif
100   continue

      residtest = dabs(resid - residprev)/amax
      call gdhigh(residtest,1,residtemp)
      if( residtest .lt. eps) then
*         write(*,1003)iam,k
*1003     format(I3,3x, I3,3x, 'breaking out of loop')
         goto 102
      else
         residprev = resid
         resid = 0.0D0
      endif
101   continue
102   continue
```

```
c
c   -- Print out the solution --
c
c   First write out the left boundary column and the data columns owned by
c   node 0.  The data columns include the top and bottom boundaries
c
      if(iam .eq. 0) then
         write(*,1001)
         do 300 j = 1,n-1
            write(*,1000) j-1, (a(i,j), i=1,totdim+2)
300      continue
1000     format('C',I2, 20(1x,F5.1))
1001     format(' ')

c
c   Then node 0 receives the rest of the columns up to totdim
c   Node 0 puts received data into its first data column, which
c   is already printed
c

         do 400 j = range+1, totdim+1
            call crecv(j,a(1,2),8*(totdim+2))
            write(*,1000) j, (a(i,2), i=1,totdim+2)
400      continue

c   Other nodes send local columns 2 through range+1.
c   The message type is the global column number

      else
         do 500 k = 1,range
            call csend(iam*range + k, a(1,k+1),8*(totdim+2),0,0)
500      continue

c   If I am the rightmost node, then also send the right boundary

         if(iam .eq. nbrnodes-1) then
           call csend(iam*range +range+1,a(1,range+2),8*(totdim+2),0,0)
         endif
      endif
      end

C

      subroutine inita(iam,nbrnodes,totdim,ndim,n,a)
      integer          iam,nbrnodes,totdim,ndim,n
      double precision a(ndim,n)
      double precision topbound, botbound, leftbound, rightbound
```

```
      topbound    = 0.0
      botbound    = 0.0
      leftbound   = 200.0
      rightbound  = 200.0
c
c  -- Zero out a --
c
      do 100 i=1,totdim+2
         do 100 j=1,n
            a(i,j) = 0.0
100   continue
c
c  -- Put in boundaries --
c
      do 200 j=1,n
         a(1,j)           = topbound
         a(totdim+2,j)    = botbound
200   continue

      if(iam .eq. 0) then
         do 300 i=1,ndim
            a(i,1)    = leftbound
300      continue
      endif

      if(iam .eq. nbrnodes-1) then
         do 400 i=1,ndim
            a(i,n)    = rightbound
400      continue
      endif
      end

C


      subroutine shadow(ndim,n,totdim, iam,nbrnodes,range,a)
c
c  Shadow performs the exchange of neighbor columns into the shadow buffers
c
      integer          ndim,n,totdim
      integer          iam, nbrnodes, range
      double precision a(ndim,range+2)

      integer          length,type,leftnode,rightnode,leftid,rightid

      leftnode  = iam - 1
      rightnode = iam + 1
      length    = (totdim+2)*8
      type      = 100
```

```
      if(iam.eq.0) then
c
c   If I am the leftmost node of the array (node 0) then only exchange
c   with the right (to the left is a boundary of the array)
c
          rightid = irecv(type, a(1,range+2), length)
          call csend(type, a(1,range+1), length, rightnode,0)
          call msgwait(rightid)

      else if (iam .eq. nbrnodes) then
c
c   If I am the rightmost node of the array (highest numbered node) then
c   only exchange with the node to the left.
c
          leftid = irecv(type, a(1,1), length)
          call csend(type, a(1,2), length, leftnode,0)
          call msgwait(leftid)

      else
c
c   Otherwise I am a node in the middle, so exchange with nodes to either side.
c
          leftid  = irecv(type, a(1,1), length)
          rightid = irecv(type,a(1,range+2), length)

          call csend(type, a(1,2), length, leftnode,0)
          call csend(type, a(1,range+1), length, rightnode,0)

          call msgwait(leftid)
          call msgwait(rightid)

      endif
      end
```