April 1993
Order Number: 312547-001

PARAGON™ OSF/1
INTERACTIVE PARALLEL DEBUGGER
MANUAL

Intel® Corporation

REV.	REVISION HISTORY	DATE
-001	Original Issue	4/93

LIMITED RIGHTS

Preface

This manual describes the Interactive Parallel Debugger (IPD), a symbolic source-level debugger for Fortran, C, and assembly language programs running under the Paragon™ OSF/1 operating system. It contains information describing how to use IPD, as well as detailed reference information on IPD commands.

This manual assumes you are an application programmer proficient in the use of C, Fortran, or assembly language and the Paragon OSF/1 operating system. The manual contains an overview of IPD, and describes all of the IPD commands in a reference format.

Organization

Chapter 1	"Introduction," is an overview of IPD features. It also presents some important information you need to use IPD effectively.
Chapter 2	"IPD Commands," provides detailed information on all the IPD commands in a reference format.

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) 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:

Paragon™ OSF/1 Commands Reference Manual
312486
Provides detailed information about the commands for the Paragon OSF/1 operating system.

Paragon™ OSF/1 C System Calls Reference Manual
312487
Provides detailed information on the C calls for the Paragon OSF/1 operating system.

Paragon™ OSF/1 C Compiler User's Guide
312490
Describes the C compiler for the Paragon OSF/1 operating system.

Paragon™ OSF/1 Fortran Compiler User's Guide
312491
Describes the Fortran compiler for the Paragon OSF/1 operating system.

Paragon™ OSF/1 Fortran System Calls Reference Manual
312488
Provides detailed information on the Fortran calls for the Paragon OSF/1 operating system.

Paragon™ OSF/1 User's Guide
312489
Gives an overview of the Paragon OSF/1 operating system. Tells how to develop and run programs.

Comments and Assistance

Intel Supercomputer Systems Division is eager to hear of your experiences with our new software product. Please call us if you need assistance, have questions, or otherwise want to comment on your Paragon system.

U.S.A./Canada Intel Corporation
phone: 800-421-2823
email: support@ssd.intel.com

Intel Corporation Italia s.p.a.
Milanofiori Palazzo
20090 Assago
Milano
Italy
1678 77203 (toll free)

France Intel Corporation
1 Rue Edison-BP303
78054 St. Quentin-en-Yvelines Cedex
France
0590 8602 (toll free)

Japan Intel Corporation K.K.
Supercomputer Systems Division
5-6 Tokodai, Tsukuba City
Ibaraki-Ken 300-26
Japan
0298-47-8904

United Kingdom Intel Corporation (UK) Ltd.
Supercomputer System Division
Pipers Way
Swindon SN3 IRJ
England
0800 212665 (toll free)
(44) 793 491056 (*answered in French*)
(44) 793 431062 (*answered in Italian*)
(44) 793 480874 (*answered in German*)
(44) 793 495108 (*answered in English*)

Germany Intel Semiconductor GmbH
Dornacher Strasse 1
8016 Feldkirchen bel Muenchen
Germany
0130 813741 (toll free)

World Headquarters
Intel Corporation
Supercomputer Systems Division
15201 N.W. Greenbrier Parkway
Beaverton, Oregon 97006
U.S.A.
(503) 629-7600

Table of Contents

Chapter 1
Using the Interactive Parallel Debugger

Chapter 2
IPD Commands

List of Tables

Using the Interactive Parallel Debugger 1

The Interactive Parallel Debugger (IPD) is a complete symbolic, source-level debugger for parallel programs that run under the Paragon™ OSF/1 operating system. Beyond the standard operations that facilitate the debugging of serial programs, IPD offers custom features that ease the task of debugging parallel programs.

Through a command-line interface, which includes on-line help, you can examine and modify running processes. Among the features specifically designed to aid debugging in a parallel environment are facilities to help debug message-passing, and the ability to set a command context to apply commands to multiple processes running on multiple nodes. With these facilities, you can set breakpoints in selected processes, monitor the queues of messages passing among processors, and display stack tracebacks and the values of registers or variables.

IPD lets you debug parallel programs written in the following programming languages:

- C
- Fortran
- i860™ assembly language

The IPD command and display syntax for variables follows the language convention of the program being debugged.

IPD Features

IPD gives you control over the debug environment and program execution, and provides several ways to examine and modify the program.

Debug Environment Control

Control over the debug environment allows you to customize aspects of your debugging session to save time. This includes the following:

- Defining command aliases and setting debug variables.
- Setting the debug context.
- Debug session recording.
- IPD command file creation and execution.
- Access to on-line help.

You can customize the debug environment by defining aliases and debug variables. Aliases are your versions of the IPD commands, and debug variables are your versions of strings used in IPD commands. This allows you to create convenient shortcuts to commands you use most commonly.

You have control over the *debug context*, which determines the processes that are the targets of IPD commands. IPD sets the initial default context at load time. You can then change the default context, or specify a context for a single command at any time during the session. For example, the context *(all:0)* is the process with process type 0 on all nodes. The default context is displayed as part of the prompt.

You can record all or part of your debug session. Executing the IPD **log** command records all subsequently entered IPD commands and their responses in a log file.

You can create a file consisting of a set of IPD commands that you intend to execute more than once, and execute this file from within the debugger. In addition, you can create a special file containing commands that are to be executed whenever you invoke IPD. This file must be named *.ipdrc* and must reside in your home directory. This file can be used, for example, to define your standard alias and debug variable definitions.

On-line help is also available as you use IPD. By entering the **help** or **?** commands, a brief description of all IPD commands is displayed. By adding the name of a command to the **help** command line, detailed help on that command is provided.

Program Execution Control

IPD gives you control over the execution of your program by providing the following:

- Program loading.
- Running, halting, and single-stepping through program execution.
- Code breakpoints.

Load control allows you to specify the partition into which you are loading your program, and, if desired, the nodes within that partition. You can load multiple files on different sets of nodes within a partition, and you can specify the process type of all processes. In addition, you can pass arguments to your program on loading, and redirect standard input.

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

When you issue a **run**, **rerun**, or **continue** command, execution of the specified processes is started, and then a prompt is displayed, allowing you control over command entry while the program is running. If you issue the **wait** command, the prompt is not returned until all processes within the context stop, unless you issue a keyboard interrupt. It is important to be aware that executing processes are allowed to write to *stdout* and *stderr* in only two situations:

- Before each IPD prompt.
- During execution of a **wait** command.

While another command is executing, processes can only be read from the keyboard only if you issue a **wait** command.

IPD facilitates setting and removing execution (code) breakpoints. You can set execution breakpoints at procedure calls, source line numbers of executable statements, and instruction addresses.

Program Examination and Modification

IPD provides numerous ways to examine and modify your program to aid in debugging, including the following:

- Source code listing.
- Message queue display.
- Program variable, memory address, register, and stack traceback display.
- Assignment of new values to program variable and memory addresses.

With the **list** command, you can list source code from the current execution point, from a specified procedure, or from a source line number, specifying the number of lines to be listed. Line numbers are displayed in the listing. For debugging on a more detailed level, the **disassemble** command allows you to display assembly code.

In parallel programs running on multiple nodes, many program errors are connected with messages passed among processes. IPD commands allow you to display queues of messages sent but not yet received, and receives that have been posted but not yet filled.

The **display** command allows you to ensure that your program variables and memory addresses have the expected intermediate values. In addition, you can use the **frame** command to display a stack traceback, listing the routines accessed, and the files in which those routines are located. If a routine is compiled to produce debug information, line numbers are displayed; if not, memory addresses are displayed.

Another important feature is the ability to assign a new value to a program variable or memory location for the current run. This gives you the opportunity to see the result of such a change without having to edit and recompile your program before you know what the change will accomplish.

Compiling for Debugging

To compile for debugging, you should use the following Paragon OSF/1 compiler switches:

-Mdebug Generate symbol and line number information. (Default **-Mnodebug**.)

-Mframe Generate stack frames on function calls. (Default **-Mnoframe**.) Debugging code compiled with **-Mnoframe** will result in stack tracebacks that have missing calls when you use the **frame** command.

-O0 Optimization off. If you do not specify **-O0** (the default is **-O1**), access to individual source lines will be decreased, and display or modification of variables and registers will probably have unpredictable results.

You can debug programs not compiled for debugging, but your ability to debug will be very limited.

When you are debugging code compiled with the **-nx** or **-lnx** switches, you can debug the code running on the nodes, or by setting the context to **(host,host)**, you can debug the controlling process in the service partition. If you compile without either of these switches, you are running in the service partition, and the debug context is automatically set correctly.

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

Invoking IPD

IPD resides on the Intel supercomputer system, and runs under the Paragon OSF/1. You can be logged in either directly or remotely. You invoke IPD from the Paragon OSF/1 prompt with the following syntax:

ipd

When you invoke the debugger, IPD automatically looks for a configuration file named *.ipdrc* in your home directory. If you have created this file, it executes the IPD commands contained in this file. For more information on the *.ipdrc* file, see the description of the **exec** command in Chapter 2.

IPD Commands

The IPD commands fall generally into three categories: execution control, program display, and debug environment. Table 1-1, Table 1-2, and Table 1-3 list the IPD commands associated with 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 displays 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 Command

Command	Minimum Abbreviation	Description
break	**b**	Set and display breakpoints
continue	**conti**	Continue processes stopped by command or by a breakpoint
instrument	**i**	Add, remove, or display program instrumentation for performance data collection
kill	**k**	Terminate processes
remove	**rem**	Remove breakpoints
rerun	**rer**	Restart the application without reusing command line arguments
run	**ru**	Restart the application, reusing any previous command line arguments
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	Abbreviation	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	**f**	Display the runtime activation stack
list	**li**	List source code of loaded program
msgqueue	**ms**	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	Abbreviation	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 Paragon OSF/1 command

The only commands you can issue prior to the **load** command are those in Table 1-3. 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 require either a default or specified context.

Syntax of IPD Commands

IPD command lines have the following general form (where *full_command* denotes an IPD command and all appropriate arguments):

full_command[; *full_command*;] ... [#*comment*]

full_command The form of a *full_command* can be one of the following:

command arguments

command -*switch arguments*

command (*context*) -*switch arguments*

command One of the IPD commands

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.

-*switch* A command option shown in boldface and preceded by a dash is a command line switch. Whether a switch has a following argument depends upon the command. Command line switches with no following argument can appear anywhere on the command line after the command name. Switches 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 **the 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 can be entered either at the end of a command line, starting with a pound sign (#) followed by a space, or on a line by itself, indicated by a pound sign (#) as the first character of a command line. All following 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 a number base other than decimal, it must have a leading zero, followed by the first letter of the base. In octal, it must have a leading *0o*. A hexadecimalvalue must have a leading *0x*. The leading zero is required.

For all IPD commands, a *filename* argument refers to a Paragon OSF/1 pathname where the tilde (~) character denotes your home directory. IPD only substitutes your environment variable *$HOME* for the tilde; IPD does not expand *~user* names.

Using Aliases

When you issue an IPD command, IPD first searches the IPD alias list before it matches a command to the IPD command table. You can alias any command to one or more characters for your convenience. If you create a file named *.ipdrc* in your home directory containing a set of **alias** commands that define convenient aliases for those commands that you use most during a debug session, these definitions are automatically included whenever you invoke IPD. See the **alias** command for more information.

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 the 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 use the same syntax for loading your program that you use from the shell. IPD loads the program and sets the initial default context. If the program specifies partitions and nodes internally, it works as if it were loaded from the shell. If you do not specify a context for the commands whose syntax allows you to specify a context, IPD uses the default context; the context used by the comamnd (either default or specified) is referred to as the *current context*. You can change the default context with the **context** command. The default context is shown as the IPD prompt.

The Paragon OSF/1 operating system allows you to change your program's process type (ptype) with a call to **setptype()**. After a call to **setptype()**, the process has a new process type, but still owns the old process type for the duration of the application's execution (even if the process is gone, the process type is not reusable). In this case, IPD interprets the old and new process types as alternate names for the same process, so the call to **setptype()** does not invalidate the default context.

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, instrument, list,** and **type.**

For example, you can set breakpoints in, list, or disassemble 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,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 procedures that have been activated as you execute the program. For this program, the **frame** command tells you that nodes 0 through 2 are blocked in the **flick**() system call called from the **gdhigh**() system call; node 3 is blocked in a different routine, the **csend**() system call in the *shadow* routine:

```
(all:0) > frame
***** (0..2:0) *****
   __flick()     [_flick.s{}0x00018dc8]
   _gdhigh()     [_gdhigh.c{}0x00018ed8]
   gdhigh_()     [gdhigh_.c{}0x0001493c]
   gauss()     [gauss.f{}#86]
   main()     [pgfmain.c{}0x000001a8]
***** (3:0) *****
   __flick()     [_flick.s{}0x00018dc8]
   csend_()     [csend_.c{}0x0000fe0c]
   shadow()     [gauss.f{}#219]
   gauss()     [gauss.f{}#66]
   main()     [pgfmain.c{}0x000001a8]
```

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 following command line asks for the display of the value of the variable *iam*, which is 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 (routine names must be followed by parentheses):

```
(all:0) > disp shadow()iam
*** ERROR: search failed
***      Not found: iam
```

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 **
  ***** (3:0) *****
  iam = 1
```

Additional Information

You should be aware of the following additional information when using IPD.

Using Breakpoints

Breakpoints may be set on only the last line of a multi-line C function call, because line number information is generated only for the last line of the call. In the following example, the breakpoint must be set on the line where the *l* is:

```
printf( "%d %d %d %d\n",
                        i,
                        j,
                        k,
                        l );
```

For multi-line Fortran statements, breakpoints can be set only on the first line of the statement. In the following example, the breakpoint must be set on the "print *" line:

```
      print *,
     &        'is ',
     &        'a ',
     &        'multi-line statement.'
```

Referencing Unnamed Fortran Main Programs

Fortran programs are not required to have a PROGRAM statement. If the PROGRAM statement is omitted, the main routine is given the name *_unnamed()*. You need to be aware of this when you are qualifying breakpoints or variables in the main routine.

Displaying Fortran Variable Types

Fortran data types are represented as shown in Table 1-4. The display of some of the variable types (those shown with "<---" after them) may be unexpected. This is because the debug information generated by the compiler is not sufficient to distinguish the declared type from the type displayed by IPD in these instances.

Table 1-4. Fortran Variable Type Display

Declared type	Represented as
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

Using Keyboard Interrupts

The following information is for using keyboard interrupts during program execution:

- There are critical sections in the debugger where IPD does not allow the user to interrupt it from the keyboard. This is necessary because there are 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 IPD has hung up and is not going to respond, using the control-backslash (`<Ctrl-\ >`) key sequence should kill the debugger.

The next chapter provides detailed reference information on each of the IPD commands.

IPD Commands 2

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 choose to represent a command.

command_string The IPD command string that the *alias_name* represents. All of the text following the *alias_name* to the end of the **alias** command line, including spaces, the pound sign (#), and semicolons, are part of the *command_string*.

Description

An alias is a character string of your choice that you define to use in place of an IPD command string. Usually, aliases are abbreviations, chosen to save keystrokes. Input on a command line is matched with the list of aliases before it is compared with the IPD command list. A recursive alias definition (an alias that uses the same alias in its definition) is flagged as an error when you use the alias.

Entering 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. You can define an alias for **unalias**, but you cannot define an alias named "unalias".

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 that 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 address.

Syntax

Assign a value to a program variable:
assign [*context*] [*file*{}][*procedure*()]*variable*[,*count*] = *value*

Assign a value to a program address:
assign [*context*] [-*size_switch*] *address*[:*address*|,*count*] = *value*

Arguments

context Defines the nodes and process types 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 applies to this command. Specify the context as follows:

(*nodelist*:*ptypelist*)

The *nodelist* is the list of nodes, and the *ptypelist* is the list of processes on those nodes to which the command will apply. These can be specified as a single value, a comma-separated list, a range, a combination, or the special value all, indicating all nodes and/or process types. For more information, see the **context** command.

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 looks for the variable in the following four places, in order: in the current code block, in the current procedure, 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]**.

ASSIGN *(cont.)*

ASSIGN *(cont.)*

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.

size_switch The *size_switch* is an option you can use when you assign a value to an address. It specifies how many bytes (1, 2, or 4) are to be assigned to the given address. You may only assign whole numbers an address; these may be hexadecimal, octal, or decimal. 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 as *start_address*:*end_address* (for example **0x208:0x21b**) or as *address*,*count*, where *count* is the desired number of bytes in the address range (for example **0x208,20**).

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*").

Description

The **assign** command changes the value of a variable for the current run. If you re-run the program with the **run** or **rerun** commands, 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]**.

ASSIGN *(cont.)* ASSIGN *(cont.)*

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 but, if the address is writable, IPD will perform the requested assignment.

You cannot assign values to a structure or union as a whole; you must specify the individual members of a structure or union one at a time.

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*]

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 applies to this command. Specify the context as follows:

(*nodelist*:*ptypelist*)

The *nodelist* is the list of nodes, and the *ptypelist* is the list of processes on those nodes to which the command will apply. These can be specified as a single value, a comma-separated list, a range, a combination, or the special value all, indicating all nodes and/or process types. For more information, see the **context** command.

file{} The name of the source module in which the procedure or line resides. To refer to a file that is not where the current execution point is located, 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.

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 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. 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 line information is not available for that procedure because that procedure was not compiled for debug.

#line The source line number at which you want to set the breakpoint. The line number must be preceded with a pound sign (#). In general, the statement must be executable. For example, you cannot set a breakpoint on a Fortran **FORMAT** statement, 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*() qualifiers.

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*.

-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 on a line in the body of the loop with **-after** 5.

Description

When you define a breakpoint, it takes on either the context that you assign it, or the default context. A breakpoint's context denotes the nodes and processes to which it applies. When you display breakpoints, only those breakpoints in the current context are listed.

Entering the **break** command with no arguments displays all breakpoints in the current context. You can also use the **break** command with the *context* argument to display all breakpoints in the specified context. Following is an example of the **break** command display:

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

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

In the preceding display, the first line shows the current context for the **break** command. The labeled columns denote the following:

Bp # The number of each breakpoint. The breakpoint number is used as an argument to the **remove** command.

Type The type of the breakpoint: *C Bp* denotes code breakpoint.

File name The name of the source file associated with the breakpoint.

Procedure 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*>.

Breakpoint Condition
The condition under which the breakpoint will occur. The **after** clause is not displayed unless the *count* is greater than 1.

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

If a single C statement consists of multiple source lines, set the breakpoint at the ending line; for a multiple line Fortran statement, set the breakpoint on the first line.

When you set a breakpoint on a function, as in this example:

```
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.

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

Examples

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

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

2. 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 line 175 for process type 0 on nodes 1, 2, and 3.

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

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

```
(all:0) > break gauss.f[]#180
```

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

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

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

CONTEXT CONTEXT

Set the debug context, defining the default set of processes and nodes to which debug commands apply.

Syntax

context [(*nodelist*:*ptypelist*)]

Arguments

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. Rather than a list of nodes, you can use the special value **all** , which specifies all nodes where loaded processes reside. To debug the controlling process in the service partition, you can use the special value **host**.

ptypelist A single value indicates a single Paragon OSF/1 process type. You can specify a range of process types with the syntax *ptype1..ptype2*, where *ptype2* > *ptype1*. Specify a list of process types by separating process type numbers with commas, using the syntax *ptype, ptype,...ptype*. The *ptypelist* may include both ranges and lists of process types. Rather than a list of process types, you can use the special value **all**, which specifies all loaded processes under debug on the specified nodes. To debug the controlling process in the service partition, uing the special value **host** specifies the proper pid in the service partition.

To debug the controlling process in the service partition, uing the special value **host** specifies the proper pid in the service partition. Processes tthat do not have a process type are designated with a value that is the Paragon OSF/1 pid of the process subtraced from zero. Single Paragon OSF/1 processes, processes created by fork, and controlling processes that have not called setptype() are treated this way.

Description

The default context is first set with the **load** command and is displayed as part of the IPD prompt. You can change the default context with the **context** command. When you need to override the default context for a given command, 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.

CONTEXT *(cont.)* CONTEXT *(cont.)*

Without arguments, the **context** command displays the nodes and process types in the default context. Processes may change their process type by calling *setptype()*. However, after calling *setptype()*, the process still "owns" the old process type as well as the new one. In this situation, IPD considers the old and new process types as alternate names for the same process and the **context** command displays both.

Examples

1. Set the context to process type 0 on all nodes.

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

2. Display the default context.

```
(all:0)> context
                Current     Previous
Processors      Ptype       Ptypes      Program
==========      =======     ========    =======
(all)                 0                 gauss
```

3. Process 0 on node 1 calls **setptype(5)**. Redisplay the process types.

```
(all:0) > context
                Current     Previous
Processors      Ptype       Ptypes      Program
==========      =======     ========    =======
(0,2..3)              0                 gauss
(1)                   5     (0)         gauss
```

CONTINUE CONTINUE

Continue execution of processes stopped by command or breakpoint in the current context.

Syntax

continue [*context*]

Arguments

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

(*nodelist*:*ptypelist*)

The *nodelist* is the list of nodes, and the *ptypelist* is the list of processes on those nodes to which the command will apply. These can be specified as a single value, a comma-separated list, a range, a combination, or the special value all, indicating all nodes and/or process types. For more information, see the **context** command.

Description

It is an error to use the **continue** command in a context containing running processes. However, you can use the continue command to start a process after it has been loaded.

After the processes have been continued, IPD returns control to you by issuing the next IPD prompt. To cause IPD to wait to return control to you until a process terminates or a breakpoint is hit, use the **wait** command.

Examples

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

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

CONTINUE *(cont.)*

CONTINUE *(cont.)*

2. Continue all processes in the default context.

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

3. Continue all processes and wait for them to stop.

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

DISASSEMBLE DISASSEMBLE

Display machine code listing of process's instructions.

Syntax

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

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

Disassemble starting from 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:

(*nodelist*:*ptypelist*)

The *nodelist* is the list of nodes, and the *ptypelist* is the list of processes on those nodes to which the command will apply. These can be specified as a single value, a comma-separated list, a range, a combination, or the special value all, indicating all nodes and/or process types. For more information, see the **context** command.

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** facility (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 in which the current execution point is not located, you must specify *file{}* as a prefix to the line number. When you refer to a procedure, you can omit the *file{}* name unless there are duplicate procedure names in different files.

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 The source line number at which to start disassembly. The line number must be prefixed by a number 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) or *#line,count*.

Description

The disassemble command allows you to display assembly language code. The contents of the program's address space in memory are disassembled, rather than the contents of the executable file. The target processes must be stopped to perform the disassembly. If they are not stopped, an error message is displayed.

If you enter the command without specifying a starting point (using the current execution point), and the processes within the current context are stopped at different locations in the load module, multiple disassembly lists are displayed, one for each process with a unique execution point.

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 example:

```
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 applies to this command. Specify the context as follows:

(*nodelist*:*ptypelist*)

The *nodelist* is the list of nodes, and the *ptypelist* is the list of processes on those nodes to which the command will apply. These can be specified as a single value, a comma-separated list, a range, a combination, or the special value all, indicating all nodes and/or process types. For more information, see the **context** command.

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**	**real** (equivalent to the C float type)
complex	**float**	**string** (see Description)
dcomplex	**hexadecimal**	
decimal	**octal**	

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

variable The symbolic name of the variable that you wish to display. For assembly 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, looking for the variable in the following four places, in order: in the current code block, in the current procedure, in the static variables local to the current file, and finally in the global program variables.

count One of the ways to specify a range is to specify the beginning array element followed by a comma and a *count* (for example, **x(10),10**). You must specify the range of an array in ascending order. If you only use the array name without a subscript, all elements in the array will be displayed.

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.

procedure() To display a local variable in a procedure other than the procedure of the current context's execution point, qualify the variable with the [*file*{}]*procedure*() prefix. Because IPD can find the source file from the symbol table information, you can omit the *file*{} prefix unless the specified procedure name is duplicated in another source module.

address Display the contents of the memory location specified by the *address* argument. You can display either memory locations or variables, not both. If the first display item is a memory address, the remaining display items must also be memory addresses.

There are two ways to denote a range of memory locations. You can either specify the beginning address and the ending address, separated by a colon (for example, **0x208:0x21b**), or you can specify the beginning address followed by a comma and a count (for example, **0x208,10**).

-register Display all of the processor registers.

Description

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 assembly programs, you may use either C or Fortran syntax to display a memory address

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

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

You can display the elements of an array by specifying its name You cannot display the elements of a structure by specifying its name; individual elements must be specified.

Use the **-string** switch to display a C character array 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) iam = 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. This listing uses column-major indexing because this is a Fortran program example.

```
(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 IPD 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 execute the displayed command by pressing <Return>; the next command then appears on the screen. If you want to stop stepping through the command file, use the keyboard interrupt (entering `<Del>` or `<Ctrl-C>`) to terminate the **exec** command.
filename	The name of the IPD command file.

Description

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 are supported. For every level of nested **exec** two additional "++" characters will be prefixed to the displayed command line if it is being echoed.

You may insert comments in command files by typing # followed by a space and the comment. All characters, including semicolons, remaining in the line are considered part of the comment. When # is the first character of a line, the entire line is a comment.

You can cause IPD to execute a set of commands automatically upon IPD invocation if you put the desired commands in a file named *.ipdrc* 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 main
context (1..3:0)
break #84
break #90
```

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

```
ipd> exec -echo picf
ipd> ++ load main
       *** load symbol table for main... 100%
       *** loading program...
       *** initializing IPD for parallel appplication...
       *** 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

Description

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

Examples

1. Exit IPD.

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

FRAME FRAME

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

Syntax

frame [*context*]

Arguments

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

(*nodelist*:*ptypelist*)

The *nodelist* is the list of nodes, and the *ptypelist* is the list of processes on those nodes to which the command will apply. These can be specified as a single value, a comma-separated list, a range, a combination, or the special value all, indicating all nodes and/or process types. For more information, see the **context** command.

Description

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 the **-Mframe** switch, these routines may not generate stack frames, and may result in missing routines in the stack traceback. The **-Mframe** switch is on by default; it may slow execution of function calls.

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

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 help for the **context** command. Entering **help context** would produce the same result.

```
(all:0) > ?context
Set or display the default debug context:

context [(nodelist:ptypelist)]

The context command defines the default set of processes and nodes to which
many debug commands apply. The initial default context is set by IPD when
you load an application (see the load command).
The nodelist lists the nodes, and the ptypelist lists the processes on
those nodes, to which the commands are to apply. Either can be a single
value, a comma-separated list, or a range. The special value all means
all of the loaded nodes or processes. Another special value is host; to
debug the controlling process in the service partition, you can specify
the context (host:host), which means the node(s) in the service partition
running the controlling process, and the Paragon OSF/1 pid number of thex
the controlling process.
```

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

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
rerun         rer     Same as run except do not reuse previous argument list
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
```

INSTRUMENT INSTRUMENT

Add, remove, or display program instrumentation for performance data collection.

Syntax

Instrument program for the **prof** utility:
instrument [*context*] [[**-on**] **-prof** [*start_location* [*stop_location*]] [*directory_name*]

Remove performance instrumentation:
instrument [*context*] **-off**

List performance instrumentation information:
instrument [*context*]

Arguments

context Defines the nodes and processes that this command will affect. If the *start_location* or *stop_location* parameters are used, then 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** | *ptypelist*})

-on Turn profile instrumentation on. This is the default action when the **-prof** switch is given. The **-on** switch is provided for command symmetry with the **-off** switch.

-off Stop collecting data, write existing data to directory previously specified, and remove all profile instrumentation.

-prof Instrument the program for profiling information.

start_location The *start_location* is the point in the code at which profiling begins. This can be an entry or exit point to a procedure, a line number, or an address. The syntax for the *start_location* specification is one of the following:

[**-entry**|**-exit**] [*file*{}]*procedure*()

[*file*{}][*procedure*()]#*line*

address

INSTRUMENT *(cont.)*

INSTRUMENT *(cont.)*

stop_location The location at which performance data collection ends. The *stop_location* can be an entry or exit point of a procedure, a line number, or an address. The syntax for the *stop_location* specification can be one of the following:

[-entry|-exit] [*file{}*]*procedure*()

[*file{}*][*procedure*()]#*line*

address

Syntax elements for *start_location* and *stop_location* are defined as follows:

file{} The name of the source module in which the procedure or line resides. To refer to a line in a file other than the file containing 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 that require qualification.

procedure() The name of the procedure at which you wish to set the start or stop location, 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 the file containing that procedure.

#*line* The source line number at which you want to set the start or stop location. The line number must be preceded with a pound sign (#). The statement must be executable. For example, you cannot set a start or stop location on a Fortran **FORMAT** statement, a comment, or an empty line.

address The address at which you want to set a start or stop location. The address must be an instruction address.

-entry Place a start or stop location at the entry of the procedure specified by *procedure*(). This is the default action when only a *procedure*() name is given.

-exit Place a start or stop location at the exit of the procedure specified by *procedure*().

INSTRUMENT *(cont.)* INSTRUMENT *(cont.)*

directory_name *directory_name* is used as the name of the directory where the performance data files are written. The individual data files for each process are written to a file named *executable_name.pid.node.ptype* where *pid* is the process id, *node* is the node number, *ptype* is the current process type at the time the data is written to the file. Both node and ptype are set to -1 if they are undefined. The default for *directory_name* is mon.out. If a directory with *directory_name* exists the user is queried before its removal. An auxiliary file will be placed in the directory named **INFO** that contains information on each of the data files. The **INFO** file has the following format:

```
Controlling Process: executable_name pid_value
pid         node   ptype     Executable
xxxxxxxxxx  xxxx   xxxx   full_path_of_executable
....
....
```

The first line has three fields: the title, the name of the controlling process's executable, and its process id. The second line contains column titles for the lines to follow. Each of the rest of the lines, (line 3 through the last line in the file) contain a 10 character process id in the first field, a four character node number in the second field, a four character process type number in the third field and the full path name of the executable file for the process in the last field.

Description

In the context of the instrument command, instrumenting the code means to apply performance monitoring hooks to the code under debug to permit collection of performance data. The instrumentation of the code occurs when the **instrument** command is given. The actual collection of the data occurs when the program is executed. Data collection starts at the *start_location* and ends at the *stop_location*. The profiling data is written at the *stop_location*.

Typically, you would specify entry and exit points to procedures, or line numbers, as the *start_location* and *stop_location*, but the syntax permits flexible specification.

If the *stop_location* is placed within a loop only the first iteration of the loop is captured. If only the *start_location* is specified, performance monitoring starts at that location and continues until the end of the program. If neither start nor stop are specified, performance monitoring begins at the current execution point and continues until the end of the program.

To instrument different contexts where differing start or stop locations are desired, you must enter multiple **instrument** commands. You can instrument one context with a *start_location* and *stop_location* pair, and instrument a second context with a different *start_location* and *stop_location* pair.

INSTRUMENT *(cont.)* INSTRUMENT *(cont.)*

This command is equivalent to using the **-p** compiler switch on most Unix systems. For more information on the data collected, see the online manual pages **prof**() and **profil**().

To analyze the data generated by the prof instrumentation, use the **prof** utility. By default, the **prof** utility uses the data in the **INFO** file of the *mon.out* directory to choose the lowest *node:ptype* pair data file for the specified load file. To view **prof** output on other *node:ptype* pairs, you must specify the *executable_name.pid.node.ptype* data file through the **prof** utility **-m** switch.

Examples

1. Starting from the Unix shell, profile an application, *my_app*, for its entire run. Profiling data is placed in the *mon.out* directory.

```
ipd
ipd > load my_app
(all:0) > instrument -prof
(all:0) > continue
```

2. After starting IPD and loading a program, start collecting profiling data at the next call of the function **my_func**(). Also, in order not to overwrite the current data in the **mon.out** directory, define the output directory to be **prof_data**.

```
(all:0) > instrument -prof my_func() prof_data
(all:0) > conti
```

3. Given the following code, generate profiling data for the do loop:

```
005          program main
006
007          call init
008          do 10 i=1,n
009            ...
010 10       continue
011            ...
012            end
```

After starting IPD and loading the program, start data collection with the **instrument** command shown. This collects data on the program while it is in the loop and writes the data to the default *mon.out* directory. Note that the stop location is placed outside the do loop.

```
(all:0) > instrument -prof #8 #11
(all:0) > conti
```

KILL KILL

Terminate and remove processes in the current context.

Syntax

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

Arguments

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

(*nodelist*:*ptypelist*)

The *nodelist* is the list of nodes, and the *ptypelist* is the list of processes on those nodes to which the command will apply. These can be specified as a single value, a comma-separated list, a range, a combination, or the special value all, indicating all nodes and/or process types. For more information, see the **context** command.

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

Description

The kill command terminates and removes processes. Because **kill** is a potentially destructive command, the **kill** commands 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.

The **-force** switch is not necessary when executing a command file. IPD automatically suppresses the confirmation message when reading commands from a file.

A **kill** terminates a process and destroys all information IPD has about the process, including breakpoints, variable types, etc. When all processes in the default context have been killed, the prompt reverts to "ipd >".

KILL *(cont.)*

KILL *(cont.)*

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:

(*nodelist*:*ptypelist*)

The *nodelist* is the list of nodes, and the *ptypelist* is the list of processes on those nodes to which the command will apply. These can be specified as a single value, a comma-separated list, a range, a combination, or the special value all, indicating all nodes and/or process types. For more information, see the **context** command.

count An integer used to indicate the number of lines of source code to list. If *count* is positive, listing starts at the specified location and continues for *count* instructions. If negative, listing begins at *count*-1 instructions preceding the specified location and ends at that location.

If you do not specify a *count*, IPD uses the last *count* argument supplied to a **list** command in the current session, except when listing an entire procedure. Upon invoking IPD, the initial *count* is 50 lines. One way to use the *count* argument is to specify a large count and then use the IPD **more** facility (see the **more** command) to browse through the instructions.

LIST (*cont.*)

LIST (*cont.*)

file{} — The name of the source file 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 number 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), or *#line,count*.

Description

The **list** command displays source code lines. To list from the current execution point, all processes in the context must be stopped. This is because the current execution point can be defined only when all processes in the context are stopped. However, if you specify a line number or procedure as the starting point, 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. Refer to the **source** command for more information.

If you enter the command without specifying a starting point (using the current execution point), 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 are displayed, one for each process with a unique execution point.

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 Paragon OSF/1 commands such as **cat** or an editor to look at files of this kind. Specifying 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 does not generate an error or warning, but will provide faulty information. There is no way for the debugger to detect this situation.

If you specify a *procedure()* argument without *count* or *#line* arguments, then the entire procedure is listed, regardless of the last value of *count* specified.

LIST (*cont.*) LIST (*cont.*)

Before each listing, the **list** command displays 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 a 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.

```
(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
180:      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
194:         leftid = irecv(type, a(1,1), length)
```

LIST (*cont.*) LIST (*cont.*)

2. List 17 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 under debugger control.

Syntax

load *filename* [<*infile*] [*arguments*]

Arguments

filename	The file name of the program that you want to load. Specify the path name if the file is not in the current directory.
infile	The program's input file argument. All of the program's standard input (stdin) will be read from *infile*. The *infile* is not read until a **wait** command is issued.
arguments	Arguments to be passed to the program. Anything following *infile* is assumed to be an argument. This includes any semicolons. If the program was compiled with the **-nx** option, *arguments* should include any Paragon OSF/1 command line arguments necessary for loading the application (such as **-pn** *partition*, **-sz** *num_nodes*, **-pt** *process_type*, **-nd** *node_list*, etc.). For a complete description of these arguments, see the *Paragon™ OSF/1 User's Guide.*

Description

The **load** command loads an application under the debugger's control and sets the default debug context.

For parallel applications that use the special **-nx** runtime start-up routine, you can supply *arguments* on the **load** command line, just as if you were executing the application at the shell prompt. For non-parallel applications and for applications that call **nx_init()** and **nx_nfork()** directly, *arguments* may also be specified and will be passed to the application.

The **load** command may be used to load different programs on different nodes. To do so, compile the application with the **-nx** switch and specify the additional programs in the argument list as described in the *Paragon™ OSF/1 User's Guide.*

LOAD *(cont.)*

LOAD *(cont.)*

The default context is set when **nx_nfork()** or **nx_nforkve()** is called. For parallel applications that use the special **-nx** runtime start-up routine, the **nx_nfork()** call is executed automatically during the **load** command. In that case, the default context is set to include all compute partition processes that have the same ptype as the first program specified on the command line. For all other applications, the **load** command temporarily sets the default context to (*host:-host_pid*), where *-host_pid* is the negative of the Paragon OSF/1 pid of the controlling process; if the program later calls **nx_nfork()** directly, the default context is automatically reset to include all compute partition processes that have the same ptype as the first parallel process loaded.

The **run** and **rerun** commands may be also used to specify command line arguments or to redirect standard input. Those commands cause the application to be reloaded if the program has run, or the arguments or infile change.

Examples

1. Load the file *gauss* (compiled with the **-nx** option) on all nodes in the partition named *foo*; set the process type to 99.

```
ipd > load gauss -pn foo -pt 99
   *** load symbol table for gauss... 100%
   *** loading program...
   *** initializing IPD for parallel appplication...
   *** load complete
(all:99) >
```

2. Load the file *gauss* on 3 nodes in the *.compute* partition; set the process type to 99; redirect input to come from the file *gauss.dat* and pass the program the additional argument ***100***.

```
ipd > load gauss < gauss.dat -sz 3 -pt 99 100
   *** load symbol table for gauss... 100%
   *** loading program...
   *** initializing IPD for parallel appplication...
   *** load complete
(all:99) >
```

3. Load the file *gauss1* on node 0 in the *.compute* partition and set the process type to 1; load the file *gauss2* on nodes 1..3 in the *.compute* partition and set the process type to 2.

```
ipd > load gauss1 -on 0 -pt 1 \; gauss2 -on 1..3 -pt 2
   *** load symbol table for gauss1... 100%
   *** loading program...
   *** initializing IPD for parallel appplication...
   *** load complete
(0:1) >
```

LOAD *(cont.)*

LOAD *(cont.)*

4. Load the file *gauss* compiled without the **-nx** option.

```
ipd > load gauss
   *** load symbol table for gauss... 100%
   *** loading program...
   *** load complete
(host:-143) >
```

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 optional if you specify a file name.
-off	Turns off logging to the current log file.

Description

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 to 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, the current log file closed and the new log file is 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 *gauss.log*.

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

2. Display the name of the current log file.

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

MORE MORE

Control scrolling of IPD information on the display.

Syntax

more [-on | -off]

Arguments

-on	Turn on the **more** function to control scrolling of the display. Whenever IPD output from a given command would scroll off the screen, the display is halted. A **more** prompt is issued below the last displayed line of output, at the bottom of the screen, and IPD waits for user input (pressing any key on the keyboard) before continuing with its output.
-off	Turn off the **more** functionality for terminal output. Allows command output to scroll freely, even when it is greater than one screen in length.

Description

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 are 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. The default context is used if you do not specify one. Specify it as follows:

(*nodelist*:*ptypelist*)

The *nodelist* is the list of nodes, and the *ptypelist* is the list of processes on those nodes to which the command will apply. These can be specified as a single value, a comma-separated list, a range, a combination, or the special value all, indicating all nodes and/or process types. For more information, see the **context** command.

type Only messages of the type(s) specified by the *type* arguments will be displayed; otherwise, all types in the context are listed.

Description

The **msgqueue** command displays the messages that have been sent and have arrived on the node(s) in the current context but have not yet been received by a process on the node. If you do not specify a type, all message types are included, including those sent by library calls. Use the **recvqueue** command to display the processes that have posted receives that have not been satisfied.

Examples

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

```
(all:0) > msgq
 *** Unreceived messages in (all:0)
      Source          Destination      Msg Type   Msg Length   Buf Addr
 ================  ================  ==========  ==========  ==========

(all:0)
```

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. The default context is used if you do not specify one. Specify it as follows:

(*nodelist*:*ptypelist*)

The *nodelist* is the list of nodes, and the *ptypelist* is the list of processes on those nodes to which the command will apply. These can be specified as a single value, a comma-separated list, a range, a combination, or the special value all, indicating all nodes and/or process types. For more information, see the **context** command.

-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.

Description

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
```

PROCESS *(cont.)*

PROCESS *(cont.)*

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 column headings denote the following:

Context The nodes and process types in context format (see the **context** command for more information). If the *Context* field overflows, the **process** command splits the information into multiple lines.

State The current state of the processes. A process can be in one of ten states: *Initial*, *Executing*, *Breakpoint*, *Watchpoint*, *Stepping*, *Stepped*, *Signaled*, *Interrupted*, *Exited*, or *Exiting*. For processes in the Breakpoint and Exited 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).

Reason For terminated processes, describes why the process has exited.

Src/Obj name For all process states except Executing and Stepping, shows the source file name. For processes in the Executing and Stepping states, shows the name of the loaded object file.

Procedure name Shows the name of the procedure for all states except Executing and Stepping.

Location Shows the location of the process for all states except Executing and Stepping.

You may use the **-loadfile** switch to specify that the load module should be displayed instead of the file name and procedure.

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

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

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

Description

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

Examples

1. Exit IPD.

```
(all:all) > 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. The default context is used if you do not specify one. Specify it as follows:

(*nodelist*:*ptypelist*)

The *nodelist* is the list of nodes, and the *ptypelist* is the list of processes on those nodes to which the command will apply. These can be specified as a single value, a comma-separated list, a range, a combination, or the special value all, indicating all nodes and/or process types. For more information, see the **context** command.

type Only messages of the type specified by the *type* argument will be displayed; otherwise, all message types in the context are displayed.

Description

The **recvqueue** command displays receives that have been posted processes in the current contextbut 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 (using, for example, **irecv**() or **hrecv**()) and continued executing. If the process has posted an **hrecv**() call, which requires a handler, the name of the handler is listed under the final column.

RECVQUEUE *(cont.)* RECVQUEUE *(cont.)*

Examples

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

```
(all:0) > recvq
 *** Unsatisfied receives posted in (all:0)
Call Type  Recv Posted By     For Msg From   Msg Type  Msg Len    Handler
========= ================== ============== ========== ==========  ===========

(all:0) >
```

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. The default context is used if you do not specify one. Specify it as follows:

(*nodelist*:*ptypelist*)

The *nodelist* is the list of nodes, and the *ptypelist* is the list of processes on those nodes to which the command will apply. These can be specified as a single value, a comma-separated list, a range, a combination, or the special value all, indicating all nodes and/or process types. For more information, see the **context** command.

breakpoint_number
The number of the breakpoint to be removed. To determine the breakpoint number, use the **break** command.

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

Description

The **remove** command removes the specified breakpoints or all breakpoints from the nodes in the current context. Use the **break** command without arguments to get a listing of breakpoint numbers.

You may remove nodes from a breakpoint context by specifying the desired nodes in the context argument of the **remove** command.

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.

```
(all:0) > break

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

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

2. Remove breakpoint 3 for (0:0), then redisplay the breakpoints.

```
(all:0) > remove (0:0)

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

RERUN RERUN

Reload and restart the execution of the program, clearing previous command line arguments.

Syntax

rerun [*<infile*] [*program_args*]

Arguments

infile	The program's input file argument. All of the program's input will be redirected from *infile.*
program_args	Arguments to be passed to the program. All arguments after the *infile* argument (if it exists) are passed to the program. This includes any semicolons. See the **load** command for more information on *program_arguments.*

Description

The **rerun** command invokes the **kill**, **load** and **continue** commands to reload and execute a program from its beginning. Use the **continue** command to continue execution of a stopped or breakpointed process. All data in the program is re-initialized.

The **rerun** command does the following:

1. Kills the current program, which flushes (deletes) all outstanding messages for the application.
2. Reloads the program.
3. Resets all of the user breakpoints and instrumentation if the program arguments have not changed from the last **load**, **run**, or **rerun** command.
4. Resets the argument list.
5. Starts executing the program.

IPD executes application processes asynchronously. Application program output to the terminal is printed either before IPD issues a command prompt or during the execution of the **wait** command. Application program keyboard input is processed and redirected files are read only during execution of the **wait** command. Use the **wait** command to wait for all processes in a context to stop.

To restart the application without retyping the previous command line arguments, use the **run** command. The input redirection is not saved between **run** commands, so you need to respecify it if you issue another **run** command.

RERUN *(cont.)* RERUN *(cont.)*

Examples

1. Load the program, *gauss*, on nodes 0..3 with program arguments ***-d -f file_name***.

```
ipd > load gauss -on 0..3 -d -f file_name
```

Start the program and wait for it to complete.

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

After that completes, restart the program, this time without program arguments.

```
(all:0) > rerun -on 0..3
```

RUN RUN

Reloads and restarts the execution of a program, reusing previous command line arguments.

Syntax

run [*<infile*] [*program_args*]

Arguments

infile	The program's input file argument. If specified, all of the program's input is redirected from *infile*.
program_args	Arguments to be passed to the program. All arguments after the *infile* argument (if it exists) are passed to the program. This includes any semicolons. Refer to the **load** command for more information about program arguments.

Description

The **run** command invokes the **kill**, **load** and **continue** commands to reload and execute a program from its beginning. Use the **continue** command to continue execution of a stopped or breakpointed process, or to run in a specified context.

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.

The **run** command does the following:

1. Kills the current program, which flushes (deletes) all outstanding messages.
2. Reloads the program.
3. Resets all of the user breakpoints and instrumentation if the run command is invoked without arguments or if the arguments have not changed from the last **load**, **run**, or **rerun** command.
4. Resets the argument list if program arguments are specified.
5. Starts executing the program.

IPD executes Application processes asynchronously. Application program output to the terminal is printed either before IPD issues a command prompt or during the execution of the **wait** command. Program keyboard input is processed and redirected files are read only during execution of the **wait** command. The input redirection is not saved between **run** commands.

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

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

A **run** command that does not specify any application command line arguments reuses the argument list from the last **run** or **rerun** command command. To restart the application without using the previous command line arguments, use the **rerun** command.

See the description of the **load** command for more information on application command line arguments.

Examples

1. Load the program *gauss* with program arguments ***-d -f file_name***.

```
ipd > load gauss -d -f file_name
```

Start the program and wait for it to complete.

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

After that completes, restart the program using the same arguments.

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

SET SET

Set or display IPD 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 includes 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*.

Description

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 you use it.

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. While you can create an alias for the **unset** command, you cannot use "unset" as an alias.

You may not use the **set** command on the same command line as 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* (that is, an becomes an alias for the command string **assign (1..3:0) long_name[104]** in this example).

```
(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 paths.

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, used to specify which program's search path to access.
directory	A list of path names for the directories that contain the application source files.
-add	Add the specified directories to the source directory search path. The directories specified are appended to the end of the search path.
-remove	Remove the specified directories from the source directory search path.

Description

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 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 are added.

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

Examples

1. Display the current source directory search path for the previously loaded program *gauss*. Add */usr/you/Fpi/* to the source directory search path and list the node program.

```
(all:0) > source gauss
  Source search paths for gauss:
        .
(all:0) > source gauss -add /usr/you/Fpi
(all:0) > source gauss
   Source search paths for gauss:
        .
   Source search paths for gauss:
       /usr/you/Fpi
(all:0) > list,10
***** (all:0) ***** gauss.f
57       program gauss
58
59       include 'nx.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

Description

The **status** command displays the IPD version number, the name of the partition in which IPD is running, the number of nodes being used, the state of the IPD **more** facility, 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
IPD version number: Release 1.0
Partition name: .compute
Number of processors: 2
More: on
Log file:/home/me/test/IPD.log
Source search paths for gauss:
.
src
```

STEP STEP

Single step through the processes in the current debug context.

Syntax

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

Step one machine instruction:
step [*context*] **-instruction** [**-call**] [,*count*]

Arguments

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

(*nodelist*:*ptypelist*)

The *nodelist* is the list of nodes, and the *ptypelist* is the list of processes on those nodes to which the command will apply. These can be specified as a single value, a comma-separated list, a range, a combination, or the special value all, indicating all nodes and/or process types. For more information, see the **context** command.

-call Treat all subroutine and function calls as single statements. If **-call** is not specified, routines are entered and their statements stepped through.

-instruction Step one instruction instead of stepping one source line.

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.

Description

The **step** command allows you to execute a program one source line or machine instruction at a time. Upon returning control to the user from a **step** command, IPD invokes the **process** command to display process information.

When stepping through source line numbers, any procedures compiled without line number information are treated as if the command were **step -call**, even if you did not specify **-call**.

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

When stepping through machine instructions, you cannot step through a system call trap instruction to the operating system. The trap is treated as if the command were **step -instruction -call**.

Single-stepping is synchronous; the **step** command does not return until all processes in its context have stepped. If your program blocks during the **step** command, use the interrupt signal (pressing **<Del>** or **<Ctrl-C>**) to regain the IPD prompt. At this point the current state of the process is *Running*. Use the **stop** command to stop the process.

Examples

1. Assume that the program is stopped at line 93 on all nodes, just before a **crecv**(). Step to line 94.

```
(all:0) > process
   Context     State          Reason       Src/Obj name   Procedure  Location
  =========   ======         ======       =========      =========  ========
  (all:0)     breakpointed   C Bp 3       node.f         MAIN()     line #93
(all:0) > list 10
***** (all:0) ***** node.f
93      call crecv(SIZETYPE, size, PARTSIZE)
94      worknodes = size
95
96c
97c     receive integration parameters
98c
99      call crecv(INITTYPE, msg, MSGSIZE)
100
101c
102c     if this node is not among the worker nodes it returns to the
(all:0) > step
   Context     State          Reason       Src/Obj name   Procedure  Location
  =========   ======         ======       =========      =========  ========
* (all:0)     stepped                     node.f         MAIN()     line #94
(all:0) >
```

STOP STOP

Stop program execution in the current context.

Syntax

stop [*context*]

Arguments

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

(*nodelist*:*ptypelist*)

The *nodelist* is the list of nodes, and the *ptypelist* is the list of processes on those nodes to which the command will apply. For more information, see the **context** command.

Description

The **stop** command stops program execution. Processes that are blocked waiting for something, such as a **crecv**(), are not considered to be in a state, but are still executing. Stopping program execution when you do not have an IPD prompt requires that you send an interrupt signal (entering `<Del>` or `<Ctrl-C>`) so you can get a prompt at which you can enter a **stop** command to stop application processes. Many IPD commands require processes to be stopped so that valid information can be obtained from the operating system.

STOP (*cont.*) STOP (*cont.*)

Examples

1. A program named *gauss* 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 Paragon OSF/1 shell commands (not an IPD command) to be executed. All text following the **!** or **system** to the end of the line, including semicolons, the number sign (#), and spaces, is part of the *shell_command*.

Description

Use either the **system** or **!** command to execute a Paragon OSF/1 shell command from within IPD. The *shell_command* is not interpreted by IPD. All *shell_command* text to the end of the **system** command line is passed directly to **sh** (the Bourne shell). You may not enter any other IPD commands on the same line as the **system** command.

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/paragon/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 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:

(*nodelist*:*ptypelist*)

The *nodelist* is the list of nodes, and the *ptypelist* is the list of processes on those nodes to which the command will apply. These can be specified as a single value, a comma-separated list, a range, a combination, or the special value all, indicating all nodes and/or process types. For more information, see the **context** command.

-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.

TYPE *(cont.)* TYPE *(cont.)*

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.

Examples

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

Description

The **unalias** command removes a previously defined alias. Use the **alias** command without arguments to display the current list of alias names. You can create an alias for the **unalias** command, but you cannot use the name of the command as an alias.

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* to be unset with a **$**.
-all	Remove all currently defined command line variables.

Description

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. You can create an alias for the **unset** command , but you cannot use "unset" as an alias.

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. The default context is used if you do not specify one. Specify it as follows:

(*nodelist*:*ptypelist*)

The *nodelist* is the list of nodes, and the *ptypelist* is the list of processes on those nodes to which the command will apply. These can be specified as a single value, a comma-separated list, a range, a combination, or the special value all, indicating all nodes and/or process types. For more information, see the **context** command.

Description

The **wait** command causes IPD to return the prompt only when *all* processes within the context are not in an 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. To redirect the program's standard input, use the redirect argument in the **load**, **run**, or **rerun** command.

After a **run** or **continue** command, IPD immediately issues a prompt. To cause IPD to withold the prompt until a process hits a breakpoint or terminates, use the **wait** command.

Upon returning control to the user from **wait**, IPD invokes the **process** command to display the process information.

After you have issued a **wait**, if you decide to wait no longer for all the processes to stop running, use the interrupt signal (pressing `<Del>` or `<Ctrl-C>`) to regain the IPD prompt.

WAIT *(cont.)* WAIT *(cont.)*

Examples

1. Issue a **run** command followed by a **wait**. When all the processes have stopped running, the **wait** command issues a **process** command, and then returns 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) >
```

Index