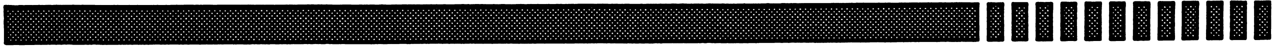
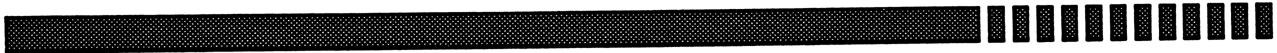
October 1989
Order Number: 311569-002



iPSC®/2 DECON
USER'S GUIDE



intel® Corporation

Copyright ©1989 by Intel Scientific Computers, Beaverton, Oregon. All rights reserved. No part of this work may be reproduced or copied in any form or by any means...graphic, electronic, or mechanical including photocopying, taping, or information storage and retrieval systems...without the express written consent of Intel Corporation. The information in this document is subject to change without notice.

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

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

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

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

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

UNIX is a trademark of AT&T
Excelan is a trademark of Excelan, Inc.
EXOS is a trademark or equipment designator of Excelan, Inc.
Green Hills Software, C-386, and FORTRAN-386 are trademarks of Green Hills Software, Inc.
iPSC is a registered trademark of Intel Corporation
Lucid and Lucid Common Lisp are trademarks of Lucid, Inc.
XENIX is a trademark of Microsoft Corporation
VAST2 is a registered trademark of Pacific-Sierra Research Corporation
VP/ix is a trademark of INTERACTIVE Systems Corp. and Phoenix Technologies, Ltd.
NFS is a trademark of Sun Microsystems
Sun Microsystems and the combination of Sun and a numeric suffix are trademarks of Sun Microsystems
VADS and Verdix are registered trademarks of Verdix Corporation
APSO is a service mark of Verdix Corporation
GVAS is a trademark of Verdix Corporation
Ethernet is a registered trademark of XEROX Corporation
VMS is a trademark of Digital Equipment Corporation

REV.	REVISION HISTORY	DATE
-001	Original Issue	09/89
-002	Revision	10/89

RESTRICTED RIGHTS

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

PREFACE

PURPOSE/SCOPE

This manual describes DECON, the iPSC®/2 Concurrent Debugger. DECON is a symbolic source-level debugger for Fortran and C programs running on the SRM and the nodes.

The manual assumes that you are a programmer familiar with the Fortran and C programming languages and the iPSC/2 Parallel Supercomputer. The manual contains an overview of DECON,then presents an example of using DECON with a simple Fortran program and lists all the DECON commands in a reference format.

ORGANIZATION

Chapter 1, "Introduction," is an overview of DECON. This chapter also lists any requirements that your system and program must satisfy..

Chapter 2, "DECON Commands," lists all the DECON commands in a reference format.

Chapter 3, "DECON Example," provides an example of how DECON can find what has turned out to be a frequent bug in concurrent applications.

Appendix A, "Source Code for the DECON Example," contains a complete Fortran source code listing of the example used in Chapter 3.

APPLICABLE DOCUMENTS

For more information, refer to the other members of the iPSC/2 manual set.

iPSC®/2 User's Guide
: This manual is intended to provide you with enough detail to begin using the iPSC/2 system.

iPSC®/2 System Administrator's Guide
This manual provides a detailed description of the system administration tasks related to operating and maintaining an iPSC/2 system.

iPSC®/2 C Language Reference Manual
This manual describes the C compiler for the iPSC/2 system.

iPSC®/2 Fortran Language Reference Manual
This manual describes the Fortran compiler for the iPSC/2 system.

iPSC®/2 Lisp Programmer's Reference Manual
This manual describes the iPSC/2 Lisp user interface and the iPSC/2 Lisp concurrent constructs.

iPSC®/2 Lisp Language Reference Manual
This manual describes the Lisp implementation that runs on the iPSC/2 nodes and its extensions.

iPSC®/2 Programmer's Reference
This manual describes iPSC/2 commands and iPSC/2 C and Fortran system calls.

iPSC®/2 VAST2 User's Guide
This manual describes how to use the iPSC/2 VX version of VAST2 software.

iPSC®/2 VX User's Guide
This manual describes how to develop programs for the iPSC/2 VX vector processing system.

iPSC®/2 VMS Interface Reference Manual
This manual describes how to install and use the VMS Bus Interface Adapter.

UNIX System V Manual Set
These manuals provide a complete description of the UNIX System V operating system.

NOTATIONAL CONVENTIONS AND SYNTAX

The commands, system calls, and routines presented in this manual have been written to follow certain grammatical rules and patterns; that is, they use a standard syntax. This syntax is described in the following sections.

Commands

You enter iPSC/2 commands at the workstation. All commands have the form:

command *arguments*

where the command is a keyword, and the arguments consist of zero or more optional or required terms. One kind of argument is called a switch, which consists of a dash followed by one or more letters, and sometimes a name or value must follow the switch. Command syntax is represented in this manual as follows:

keywords — Keywords appear in **boldface**. Keywords are those that you must use exactly as written to execute the command properly. Both command names and switches are in boldface type.

variables — Variable names are descriptive of an argument that you must supply, and appear in *italics*. They indicate that you must enter a name or value in place of the italicized word. You do not enter the name of the italicized word itself.

[ ] — Command line arguments that are optional appear in square brackets. If the optional argument is a keyword or a switch, it is in boldface type. If it represents a name or value that you must supply, it is italicized.

... — Ellipses (three periods in a row) indicate that you may repeat the argument.

| — A vertical bar indicates an exclusive or. For example, the following indicates that either the first or second option may be use, not both.

[**-c** *cubename* | **-a**].

For example, the following command syntax representation for the NX/2 **load** command shows some of the syntax elements:

load [**-c** *cubename*] [**-p** *pid*] [**-H**] [*node...*] *filename* [*arguments...*]

According to the syntax, only the word **load** and a file name are required to execute the command. However, five options are available, three of them requiring a switch. For two of them (*node* and *arguments*), you may enter more than one argument of this kind. The following example of this command uses the *cubename* option, two node values, an input file name *input_file*, and an argument value of 1000 (this example does not use the **-p** and **-H** options).

load -c alpha 3 4 input_file 1000

System Calls

The iPSC/2 system calls are used in software programs in the same way that standard system calls are used. The calls are described as follows:

Synopsis

return_type **name** *(parameters)*

Parameter Declarations

type *parameter*

The return type is shown in regular type. The name of the call is shown in **boldface.** The parameters are shown in *italics*. For example, the following are the synopsis and parameter declarations for **getcube()** for C programs:

getcube (*cubename, cubetype, srmname, keep*)

char **cubename*;
char **cubetype*;
char **srmname*;
long *keep*;

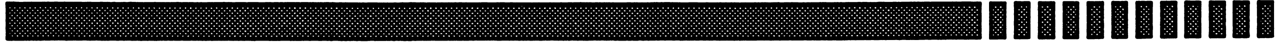
The Fortran syntax for the same routine is as follows:

SUBROUTINE **GETCUBE** (*cubename, cubetype, srmname, keep*)
CHARACTER *cubename**(*)
CHARACTER *cubetype**(*)
CHARACTER *srmname**(*)
INTEGER *keep*

C system calls that have no return type have the word *void* preceding the name of the call. Fortran routines that have no return value have the word SUBROUTINE in place of the return type. Fortran routines that return a value have the word FUNCTION following the return type. For example, **iprobe()**, which returns an integer, is shown as follows for the two languages:

Fortran:	INTEGER FUNCTION **IPROBE**(*id*)
C:	long **iprobe**(*id*)

TABLE OF CONTENTS



CHAPTER 1 INTRODUCTION

INTRODUCTION 1-1

DECON REQUIREMENTS 1-1

DECON FEATURES 1-2

THE DEBUG CONTEXT 1-3

THE EXECUTION POINT 1-4

CHAPTER 2 DECON COMMANDS

INTRODUCTION 2-1

ALIAS 2-3

ASSIGN 2-4

BREAK 2-6

CONTEXT 2-9

DECON 2-10

DIMENSION 2-12

DISPLAY 2-13

EXEC 2-16

FILE 2-18

FRAME 2-20

HELP 2-22

HLOAD 2-23

LIST 2-24

LOAD 2-28

LOG 2-29

MSGQ 2-30

PROCESS 2-32

QUIT 2-33

RECVQ 2-34

REMOVE 2-36

RUN 2-37

SET 2-38

SOURCE 2-39

STATUS 2-41

STEP 2-42

STOP 2-44

SYSTEM 2-46

TYPE 2-47

UNALIAS 2-49

UNSET 2-50

WAIT 2-51

CHAPTER 3 DECON EXAMPLE

INTRODUCTION 3-1

PREPARING THE EXAMPLE 3-3

INVOKING DECON 3-3

LOADING FILES AND SETTING BREAKPOINTS 3-3

RUNNING THE HOST PROGRAM 3-4

RUNNING THE NODE PROGRAM 3-5

BACK TO THE HOST PROGRAM 3-8

THE BUG 3-10

APPENDIX A SOURCE CODE FOR THE DECON EXAMPLE

MAKEFILE A-1

DHOST.F A-2

DPROMPT.F A-5

DFX.F A-7

DNODE.F A-8

LIST OF ILLUSTRATIONS

Figure 3-1. Control Flow for the DECON Demonstration Program 3-2

LIST OF TABLES

Table 2-1. DECON Commands 2-1

INTRODUCTION 1

INTRODUCTION

The iPSC/2 system provides a concurrent debugger called DECON. With DECON, you can debug programs consisting of several processes. DECON is a symbolic source-level debugger for Fortran and C programs running on the SRM and the nodes.

DECON contains features specifically designed to aid debugging in a concurrent environment. For example, you can set breakpoints for some processes and not others, monitor the message queues, and display a process status table. You can also display and change program variables, using their symbolic names.

DECON REQUIREMENTS

Programs that you intend to debug with DECON must be compiled with the **-B** switch. This switch ensures that DECON has access to the appropriate debug information. If your program consists of a number of files, the minimum requirement is that the file containing the main program and the files containing the routines that you want to debug be compiled with the **-B** switch. You can ensure that all Fortran files are compiled with **-B**, by setting **make**'s predefined macro F77FLAGS to **-B**. For C files, set CFLAGS to **-B**.

By listing **-B** last in the sequence of compiler flags, it will have precedence over the previous flags, except for the **-O** and **-OLM** flags. These flags should be removed.

DECON only runs on the System Resource Manager. You cannot run DECON on a remote workstation unless you remotely log onto the SRM.

DECON must load the file that it debugs. Hence, your program must not execute the **load()** system call. You must use either the DECON **hload** or **load** commands. If your program uses **load()**, comment it out in your source, recompile, and relink.

In addition to **load()**, programs that you intend to debug with DECON cannot include the following iPSC/2 system calls: **getcube()**, **relcube()**, **killcube()**, and **killproc()**. Comment them out in your source code.

DECON only debugs Fortran and C programs. iPSC/2 Lisp has its own integrated debugger.

DECON FEATURES

With DECON, you can load your program and stop it in the midst of execution at a pre-defined breakpoint. You can then investigate the program's intermediate state and see if it is what you expect. You can also single step your program and watch what happens as it executes.

- Setting breakpoints. You can set execution breakpoints and data breakpoints with the **break** command and remove them with the **remove** command.

  An execution breakpoint is set at either a source line number or a routine name. The source line number must be that of an executable statement. The program then stops just before executing that statement. If you specify a routine name, the program stops just before executing the first statement of that routine.

  A data breakpoint is set on a variable name. You can choose to have the program stop when it writes that variable or accesses it. Accessing is defined as either a read or a write.

- Starting execution. You can start execution with the **run** command. The program runs until it completes or encounters a breakpoint. The **step** command executes the next statement.
- Listing source code. You can list the source code of the program that you're debugging with the **list** command. This is useful when determining the line numbers of potential breakpoints. In addition, **list** without arguments starts at the next source code statement. This is a good way of discovering where your program has stopped.
- Displaying the message and receive queues. Many program errors have to do with message passing. With DECON, you can display the message and receive queues and watch them change as your program executes.
- Displaying program variables. You can check that your program variables have the expected intermediate values.

DECON also lets you manage the debug environment. You can define shorthand names for DECON commands, log your debug session, and execute files containing DECON commands.

- Customizing the debug environment. You can define aliases and debug variables. Aliases are shorthand versions of DECON commands. Debug variables are shorthand versions of strings used by DECON commands.
- Logging a debug session. You can set up a log file to record all the DECON commands and their responses.
- Executing DECON command files. You can execute files containing DECON commands from within DECON. If you create a file named *.deconcf* in your home or current working directory it is executed automatically when you invoke DECON.

DECON provides facilities for debugging programs that consist of multiple processes. DECON uses the *debug context* to keep track of what processes you're issuing DECON commands to. You can choose a particular process or a group of processes. For example, the context (host:100) is process 100 on the host. The context (nodes:0) is process 0 on all nodes.

- Setting the debug context. DECON has the concept of a *current context*, set with the **context** command. This is a default context assumed by DECON commands. It is displayed as part of the DECON prompt. You can temporarily override the current context by specifying a context as part of a DECON command. This is called the *specified context*.
- Starting and stopping individual processes. The **run** and **stop** commands operate in the current or specified context. For example, the command **run(1:0)** starts up process 0 on node 1.
- Wait for completion or a breakpoint. By default, control returns to DECON after the first process stops, even though other processes may still be running or waiting for messages. If the nodes are running asynchronously, you can use the **wait** command to wait for all nodes to encounter breakpoints or terminate. That is, after you issue the **wait** command, the DECON prompt does not return until all the processes, even those not in context, have stopped running.

THE DEBUG CONTEXT

Manipulating the debug context is essential to using DECON. Consider the following example. The file *host.f* contains the main host source. The file *prompt.f* contains the routine *userinpt()*. Assume that you want to set an execution breakpoint at the beginning of *userinpt()*. Because *prompt.f* is linked with the host program, DECON does not recognize *userinpt()* when the context is (nodes:0). To set an execution breakpoint at *userinpt()*, you must change the context to (host:100).

```
(nodes:0) > break userinpt()
        **** error : userinpt not a subroutine name
(nodes:0) > context(host:100)
(host:100) > break userinpt()
```

That last example changed the current context. To set the breakpoint without changing the current context, you would issue the following command:

```
(nodes:100) > break(host:100) userinpt()
```

To check that the breakpoint is set, use the **break** command without arguments. It lists the current breakpoints.

```
(host:100) > break
  (host:100)
       bp no   active  action  object                         on      off
       =====   ======  ======  ======                         ==      ===
         1       1      break   userinpt()
(host:100) >
```

THE EXECUTION POINT

DECON also uses the concept of an *execution point*. The execution point is the point in a process just before the next statement to be executed. The execution point determines what variables are in scope and what file a line number refers to. Each process has its own execution point The scope of a variable is those parts of a program where it is recognized and accessible.

For example, consider the same example as in the previous section. Assume that the variable *size1* is local to *userinpt()*. When the execution point is in the main program, DECON does not recognize *size1*.

```
(host:100) > display size1
        **** error : symbol size1 not found
```

Also assume that you still have the execution breakpoint set at the routine *userinpt()*. If you then run the main program and break when *userinpt()* is called, *size1* is in the current scope and DECON recognizes it. Also, the list command now lists lines from the file *prompt.f*, not *host.f*.

```
(host:100) > run
        (host:100) stopped at userinpt()   (bp# 1)
(host:100) > list
  (host:100)
17      print *
18      print *, '******************** INTEGRATION EXAMPLE **************
19     >*********'
20      print *
21      print *, 'This Example uses the iPSC to calculate pi by integrati
22     >ng    '
23      print *, 'the function:'
24      print *, '                    f(x) = 4 / (1 + x**2)'
25      print *, 'between x=0 and x=1.'
26      print *, 'The method used is the n-point rectangle quadrature rul
27     >e.'
(host:100) > display size1
        (host:100) size1 = 0
(host:100) >
```

DECON COMMANDS 2

INTRODUCTION

This chapter contains a complete list of DECON commands in a reference manual format. Table 2-1 lists the DECON commands by function.

Table 2-1. DECON Commands *(Sheet 1 of 2)*

Execution Control		
	break	Sets and displays breakpoints
	remove	Removes breakpoints
	run	Starts up one or more processes
	step	Executes the next statement
	stop	Stops execution
	wait	Waits until all processes stop running
Debug Environment		
	alias	Sets or displays aliases for DECON commands
	unalias	Deletes aliases
	context	Sets the debug context
	decon	Invokes DECON
	quit	Exits DECON
	exec	Executes a DECON command file
	file	Sets the current source file

Table 2-1. DECON Commands *(Sheet 2 of 2)*

Debug Environment *(continued)*		
	source	Sets or displays the current source directory
	help	Displays help information
	hload	Loads a host program
	list	Lists source code
	load	Loads a node program
	log	Records debug session
	set	Sets or displays debug variables
	unset	Deletes debug variables
	system	Executes a UNIX command
Program Display		
	zssign	Assigns a new value to the specified program variable.
	dimension	Displays the number of nodes and the dimension of the allocated cube
	display	Displays the value of a program variable
	frame	Displays the runtime activation stack
	msgq	Displays message queue
	recvq	Displays receive queue
	process	Displays current state of user processes
	status	Displays and empties DECON message buffer
	type	Displays a program variable's type

ALIAS ALIAS

Displays or sets aliases.

Syntax

alias [*alias_name command_name*]

Arguments

alias_name A string that you choose as another name for a DECON command.

command_name A DECON command.

Notes

Without arguments, **alias** displays the current aliases.

Aliases are usually abbreviated names that save keystrokes.

Use the **unalias** command to delete an alias.

Examples

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

```
(nodes:0) > alias c context
```

2. Display the current aliases.

```
(nodes:0) > alias
     Alias     Contents
      =====     ========
      c         context
```

ASSIGN

Assigns a value to a program variable.

Syntax

assign [*context*] *variable* [*value*]

Arguments

context — This is the context as defined in the **context** command. It is specified as follows:

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

Refer to the description of the **context** command for more information.

variable — The symbolic name of the variable that you want to display. Note that if the variable name in your program has a dollar sign ($), you must replace the $ with an underscore (_) when referring to the variable from within DECON. This is because the C and Fortran compilers convert $ to _.

value — The value that you want to assign to *variable*. If *variable* is a real number, you must include the decimal point. If you leave out this parameter, DECON prompts you for the new value.

Notes

If you leave out *value*, DECON prompts you for the new value. If you decide you don't to change variable, enter a **<Return>**.

When accessing Fortran variables, enter them in lowercase, even if they are uppercase in your source code. By default, the F77 Fortran compiler converts all symbol names to lowercase. Note, however, that if you compile your Fortran source code with the **-U** switch, the code is compiled as case-sensitive, and DECON treats it as such.

Examples

1. Assign a new value to the variable *integral.*

```
(host:100) > display integral
        (host:100) integral = 3.14160099e+00
(host:100) > assign integral
        (host:100) integral = 3.14160099e+00    <- 1.23
```

ASSIGN *(cont.)*

ASSIGN *(cont.)*

```
(host:100) > display integral
        (host:100) integral = 1.23000000e+00
(host:100) >
```

2. Display the current aliases.

```
(nodes:0) > alias
     Alias   Contents
     =====   ========
     c       context
```

BREAK

Sets a breakpoint or displays current breakpoints. Breakpoints can be execution breakpoints or data breakpoints.

Syntax

Execution Break

break [*context*] [#*stmt_no* | *routine_name*]

Data Break

break [[*context*] [**access** | **write**] *variable*]

Arguments

break [*context*] [#*stmt_no* | *routine_name*]

context — This is the context as defined in the **context** command. It is specified as follows:

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

Refer to the description of the **context** command for more information.

stmt_no — This is a source code line number. This number must be preceded with a #. If the source file is not current, you must precede the # with the source file name (minus the source extension) followed by {}.

Note that the statement must be executable. For example, you cannot set a breakpoint at the label of a Fortran **FORMAT** statement, a comment, or an empty line.

The process stops just before executing the specified statement.

routine_name — A routine name. The program stops when this routine is called. The execution point is within the routine and just before its first executable statement.

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

break [[*context*] [**access** | **write**] *variable*]

context — This is the context as defined in the **context** command. It is specified as follows:

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

Refer to the description of the **context** command for more information.

access — Specifies that a break will occur when the program accesses the specified variable. An access is either a read or a write. If neither **access** nor **write** is specified, the break occurs when the variable is accessed. The process stops at the line after the line containing the access.

write — Specifies that a break will occur when the program writes the specified variable. If neither **access** nor **write** is specified, the break occurs when the variable is accessed. The process stops at the line after the line containing the access.

variable — Symbolic name for a program variable. Note that if the variable name in your program has a dollar sign ($), you must replace the $ with an underscore (_) when referring to the variable from within DECON. This is because C and Fortran compilers convert $ to _.

Notes

When process excution stops due to a breakpoint, the status message displays the number of the next line to be executed.

break without any arguments lists the currently active breakpoints in the current context. The **on**, **off**, and **active** fields are not currently used.

break requires a current context. Either set the context with the **context** command or set it as part of the **break** command.

Before you can set a data breakpoint the execution point of your program must be "inside" the user program; that is, the execution point must be in the user's address space. To get your execution point inside a program, issue the **step** command. This requirement is unnecessary for execution breakpoints.

BREAK *(cont.)*

BREAK*(cont.)*

Examples

1. Set a breakpoint at line number 84 in the current source file.

```
decon> break (host:100) #84
```

2. Set a breakpoint at line #51 in the source file *prompt.f*, when this file is not the current source file. Notice that the current context is (host:100) and appears in the DECON prompt.

```
(host:100)> break prompt{}#51
```

3. Set a breakpoint at the routine *f()* in the node program *node.f*. This breakpoint occurs whenever the node program calls *f()*. Note that the context is still (host:100), and you must specify the node context as part of the **break** command.

```
(host:100)> break (nodes:0) f()
```

4. Display the current breakpoints. Note that the **break** command displays only those breakpoints in the current context. If you want to display the breakpoints in a particular context, you can specify the context in the **break** command.

```
(host:100) > break
  (host:100)
        bp no   active  action  object                          on      off
        =====   ======  ======  ======                          ==      ===
        1       1       break   host.f{}MAIN()#84
        2       1       break   prompt.f{}userinpt()#51
 (host:100) > break (nodes:0)
  (nodes:0)
        bp no   active  action  object                          on      off
        =====   ======  ======  ======                          ==      ===
        3       1       break   f()                             All     None
(host:100) >
```

CONTEXT CONTEXT

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

Syntax

context [({**all** | **nodes** | **host** | *nodelist*} : {**all** | *pidlist*})]

Arguments

all When used as the first argument (left side of the colon), **all** specifies the host and all the nodes. On the right, it specifies all loaded processes.

nodes Specifies all the nodes.

host Specifies the host.

nodelist *node* |,*node..node* [,*nodelist*]... where *node* is the node number and *node1..node2* has *node2* > *node1*.

pidlist *pid* |,*pid..pid* [,*pidlist*]... where *pid* is the NX/2 process ID and *pid1..pid2* has *pid2* > *pid1*.

Notes

You can set up a default context with the **context** command. This is called the *current* context. You can also specify a debug context with most DECON commands that is valid only for the duration of that command. This is called the *specified* context.

Examples

1. Set the context to process 100 on the SRM.

```
decon> context(host:100)
(host:100) >
```

2. Set the context to process 0 on all nodes 0, 1, and 2.

```
decon> context (0..2:0)
(nodes:0) >
```

DECON

Invokes the DECON debugger.

Syntax

decon

Arguments

None

Notes

When you invoke DECON, DECON automatically executes commands in its configuration file. This file must be called *.deconcf* (note the leading period). It must reside in the current directory or in your home directory. It is an ASCII file containing DECON commands.

Examples

1. Invoke DECON. Note that DECON echoes the commands in its configuration file. It places a ++ in front of the command to show that it came from a DECON command file.

```
% decon
        **** iPSC/2 Concurrent Debugger R3.1 xxx
decon> ++ hload 100 host
        decon: loading host program ......
        decon: program loaded ......
decon> ++ load node
        decon: loading node program ......
        decon: program loaded ......
decon> ++ context (host:100)
(host:100) > ++ break #84
(host:100) > ++ break #90
(host:100) > ++ context (nodes:0)
(nodes:0) > ++ break #93
(nodes:0) > ++ break #99
(nodes:0) >
```

DECON *(cont.)*

DECON *(cont.)*

The configuration file in the preceding example is as follows:

```
hload 100 host
load node
context (host:100)
break #84
break #90
context (nodes:0)
break #93
break #99
```

DIMENSION

Displays the number of nodes and the dimension of the allocated cube.

Syntax

dimension

Arguments

None

Notes

Note that **dimension** displays both the number of nodes and the cube's dimension.

Examples

1. Determine how many nodes are in the current cube.

```
decon> dimension
        **** Nodes Used: 64, Dimension: 6
decon>
```

DISPLAY DISPLAY

Displays the value of the specified variable.

Syntax

display [*context*] *variable* [,*variable*] ...

Arguments

context	This is the context as defined in the **context** command. It is specified as follows: ({**all** | **nodes** | **host** | *nodelist*}:{**all** | *pidlist*}) Refer to the description of the **context** command for more information.
variable	The symbolic name of the variable that you want to display. Note that if the variable name in your program has a dollar sign ($), you must replace the $ with an underscore (_) when referring to the variable from within DECON. This is because the C and Fortran compilers convert $ to _.

Notes

When displaying Fortran variables, enter them in lowercase, even if they are uppercase in your source code. By default, the F77 Fortran compiler converts all symbol names to lowercase. Note, however, that if you compile your Fortran source code with the **-U** switch, the code is compiled as case-sensitive, and DECON treats it as such.

When displaying the elements of a Fortran array, use [ ] to bracket the array index, rather than ( ).

You can display any equivalenced Fortran variables.

When displaying Fortran common variables, enter them as you would any other variable. However, you must use the symbolic name in the COMMON statement that is in the routine containing your execution point.

When displaying C variables, enter them as they appear in your source code. DECON treats C variables as case sensitive. Also, the field reference operators **.** and **->** are used to refer to structured and union variables in C.

DISPLAY *(cont.)*

DISPLAY *(cont.)*

Examples

1. Display the variable named *size* in the host program with NX/2 pid 100.

```
(host:100) > display size
        (host:100) size = 4
```

2. Display the array called *msg*.

```
(host:100) > display msg
        (host:100) msg
                (1) = 0    (2) = 0    (3) = 0
                (4) = 1072693248    (5) = 100
```

Notice that the elements of *msg* are displayed as **integer*4** because that's how msg is declared. Refer to the following Fortran code fragment. However, *b* is double precision and equivalenced to *msg(3)*. What you see in *msg(3)* and *msg(4)* is the bit pattern for the **double precision** number *b* shown as an **integer*4**.

```
        integer*4 msg(5)

        double precision  integral, a, b
c
c       equivalence a, b, and points to elements of msg
c
        equivalence (msg(1), a)
        equivalence (msg(3), b)
        equivalence (msg(5), points)
```

DISPLAY *(cont.)*

DISPLAY *(cont.)*

3. Display the fifth element of the array *msg*.

```
(host:100) > display msg[5]
        (host:100) msg(5) = 100
```

4. Assume that in your C program the structure *msg* is defined as follows:

```
21  struct msg_type {       /* structure for parameters of integration */
22          double  a,      /* lower limit of integration */
23                  b;      /* upper limit of integration */
24          long  points;   /* number of points in quadrature rule */
25  };
26
27  struct msg_type msg;    /* integration parameters */
```

Assume that the current context is (host:100). Display the structure *msg*.

```
(host:100) > display msg
        (host:100) msg{
                        a  = 0.00000000e+00
                        b  = 1.00000000e+00
                        points  = 100
                }
```

Now display the field *b* in the structure *msg*.

```
(host:100) > display msg.b
        (host:100) msg.b = 1.00000000e+00
(host:100) >
```

EXEC

Executes a file containing DECON commands.

Syntax

exec [step] *filename*

Arguments

step Causes the DECON command file to be executed line by line. The screen displays each command before executing it. You can chose to execute it by pressing **<Return>**. Then, the next command appears on the screen.

filename The file name of the DECON command file. This may be a path name.

Notes

When you invoke DECON, it executes the default command file called *.deconcf*. This file must be located in the current directory or your home directory. The *.deconcf* file is often used to define configuration information, such as a list of convenient aliases.

The DECON commands executed from a command file are always echoed on the screen.

When executing a command file in **step** mode, you may enter any other commands when DECON is waiting for a **<Return>**; your commands are executed immediately. DECON resumes the exec file commands when you enter **<Enter>** on an empty line.

Examples

1. Execute the command file *picf*.

```
decon> exec picf
decon> ++ hload 100 host
        decon: loading host program ......
        decon: program loaded ......
decon> ++ load node
        decon: loading node program ......
        decon: program loaded ......
decon> ++ context (host:100)
(host:100) > ++ break #84
(host:100) > ++ break #90
(host:100) > ++ context (nodes:0)
(nodes:0) > ++ break #93
```

EXEC *(cont.)* EXEC *(cont.)*

```
(nodes:0) > ++ break #99
(nodes:0) >
```

2. Execute the command file *picf* in step mode.

```
decon> exec step picf
decon> ++ hload 100 host
<Return>
            decon: loading host program ......
        decon: program loaded ......
decon> ++ load node
<Return>
        decon: loading node program ......
        decon: program loaded ......
decon> ++ context (host:100)
<Return>
(host:100) > ++ break #84
<Return>
(host:100) > ++ break #90
break(all:all)
  (nodes:0)
        bp no   active  action  object                        on      off
        =====   ======  ======  ======                        ==      ===
  (host:100)
        bp no   active  action  object                        on      off
        =====   ======  ======  ======                        ==      ===
        1       1       break   host.f{}MAIN()#84
(host:100) >
(host:100) > ++ context (nodes:0)
<Return>
(nodes:0) > ++ break #93
<Return>
(nodes:0) > ++ break #99
<Return>
(nodes:0) >
```

FILE

Sets the current list file.

Syntax

file *filename*

Arguments

filename The name of the source code file used by the **list** command. *filename* may be a path name.

Notes

The current source file is the file used by the **list** command.

Examples

1. Change the current list file to *prompt.f* and list lines #17 through #31.

```
(host:100) > file prompt.f
(host:100) > list #17,#31
(host:100)
17      print *
18      print *, '******************** INTEGRATION EXAMPLE **************
19     >*********'
20      print *
21      print *, 'This Example uses the iPSC to calculate pi by integrati
22     >ng    '
23      print *, 'the function:'
24      print *, '               f(x) = 4 / (1 + x**2)'
25      print *, 'between x=0 and x=1.'
26      print *, 'The method used is the n-point rectangle quadrature rul
27     >e.'
```

FILE *(cont.)*

FILE *(cont.)*

```
28      print *
29      write(6,99)
3099    format(' How many points do you want (0 or neg. value quits)? ')
31      read (5, 110) points
(host:100) >
```

Note that if you now issue the list command and specify a context, source lines are still displayed from *prompt.f*.

```
(host:100) > list (nodes:0)
  (nodes:0)
1c
2c   prompt.f 6.1 89/03/30 04:02:21
3c
4c   Function which prompts the user for the integration parameters
5c
6
7       function userinpt (a, b, points, size)
8
9       integer*4 userinpt
10
11      double precision a, b
(host:100) >
```

FRAME

Displays the runtime activation stack in the current or specified context.

Syntax

frame [*context*]

Arguments

context This is the context as defined in the **context** command. It is specified as follows:

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

Refer to the description of the **context** command for more information.

Notes

None

Examples

1. Consider the Fortran pi example. Set a breakpoint in the file *prompt.f*. Issue **run** in the (host:100) context. Then display the activation stack. Notice that it has two entries. The first is the return from the breakpoint. This breakpoint occurs in the routine *userinpt()* in *prompt.f*. The second entry is the return to the main program from *userinpt()*.

```
(host:100) > break prompt{}#25
(host:100) > break
  (host:100)
        bp no   active  action  object                       on      off
        =====   ======  ======  ======                       ==      ===
        1       1       break   host.f{}MAIN()#84
        2       1       break   host.f{}MAIN()#90
        5       1       break   prompt.f{}userinpt()#25
(host:100) > run
 LOADING THE CUBE ...

 ******************** INTEGRATION EXAMPLE ***********************
```

FRAME *(cont.)* FRAME *(cont.)*

```
This Example uses the iPSC to calculate pi by integrating
the function:
                f(x) = 4 / (1 + x**2)
       (host:100) stopped at userinpt()#25   (bp# 5)
(host:100) > frame
userinpt(a,b,points,size)#25
MAIN()#73
(host:100) >
```

2. Now start up the host program. Hit the next breakpoint and observe the activation stack. The program has returned from *userinpt()*. The only entry is the return from the breakpoint.

```
(host:100) > run
 between x=0 and x=1.
 The method used is the n-point rectangle quadrature rule.

 How many points do you want (0 or neg. value quits)?
100
 What cube size (1-  2) should I use ?
2
        (host:100) stopped at MAIN()#84   (bp# 1)
(host:100) > frame
MAIN()#84
(host:100) >
```

HELP HELP

Displays help information.

Syntax

help

Arguments

None

Notes

When you enter help, a summary list of DECON commands appears on your screen. Pressing the <Enter> key displays an additional page.

Examples

1. Display the summary list.

```
decon> help
        alias [new_cmd cmd_str]          -- define/display aliases
        break [context] [[obj]|[assert]]-- set/display breakpoints
        context [context]                -- set/display current context
        dimension                        -- display cube dimension
        display [context] vars           -- display value of variables
        exec [step] fname                -- execute Decon command script
        file fname                       -- change source file name
        frame [context]                  -- display stack frames
        help                             -- list Decon commands
        hload [pid] fname                -- load object file into SRM
        list [context] -f [routine|n..m]-- display source lines/file name
        load [-p pid] [nids] fname       -- load object file into cube
        log [on fname | off]             -- turn on/off log of debug session
        msgq [context] [type]            -- display message queue
        process                          -- display current processes
        quit                             -- exit Decon

                  Press RETURN for Next Page
```

HLOAD HLOAD

Loads a process onto the System Resource Manager, but does not run the process.

Syntax

hload [*pid*] *filename* [> *outfile*] [< *infile*] [*args*]

Arguments

pid The NX/2 process ID (pid) assigned to the host process. This must be the same value as that returned by **mypid()** and set with **setpid()**. Valid *pids* include any non-negative integer value. If you do not specify a *pid*, a value of 0 is assumed.

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

outfile The host program's output file argument.

infile The host program's input file argument.

args The host program's arguments. Each argument must be delimited with single quotes and separated with spaces.

Notes

The *pid* must be the same number used by the **setpid()** system call in the host program.

Examples

1. Load the file *host* on the SRM.

```
decon> hload host
```

2. Load the file *host* on the SRM. Assign it pid 100. Give it the two arguments, *arg1* and *arg2*.

```
decon> hload 100 host 'arg1' 'arg2'
```

LIST

Displays source code.

Syntax

list [*context*] [*routine* | *stmt1*,*stmt2* | *stmt1*[,*count*]] [**-f**]

Arguments

context — This is the context as defined in the **context** command. It is specified as follows:

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

Refer to the description of the **context** command for more information.

routine — The name of a routine. The name must be terminated with two parentheses ( ) and be in the current list file. When you specify a routine, the first 20 statements of the routine are listed.

stmt1,*stmt2* — Statement numbers must be preceded with #. Displays the source code from statement number *stmt1* to *stmt2*. If the range is more than a screenful, the first lines scroll off the screen

stmt1,*count* — A statement number must be preceded with #. *count* lines above and below the specified statement are displayed. The default value of *count* is 10.

-f — Displays the name of the source file being listed, along with the source code.

Notes

list without arguments displays 10 lines, beginning with the next executable statement.

With the **list** command, you can also specify a context. If the context is a single process, the **list** command then displays the source code for that process, beginning at the process's execution point. If the context is several processes, each process must be identical and have the same execution point. The current scope is the file in the current context that contains the execution point.

The **file** specification overrides the **list** specification and the current scope. It continues to override until the execution point changes.

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

Examples

1. Assume that the current context is (host:100). Issue the list command after the host program encounters a breakpoint.

```
(host:100) > run
        (host:100) stopped at MAIN()#84   (bp# 1)
(host:100) > list
  (host:100)
84           call crecv(PARTTYPE, msg, MSGSIZE)
85           integral = a
86
87c clean out unreceived messages
88c          call flushmsg(-1, ALLNODES, APPLPID)
89
90           write(6, 110) integral
91110        format('  pi is approximately : ', F18.16)
92
93c
94c Figure out elapsed time.  Total milliseconds were returned
(host:100) >
```

2. List line #93 of the node program bracketed by three lines.

```
(host:100) > list (nodes:0) #93,3
  (nodes:0)
90c     get the size of the cube to put to work on problem
91c
92100     continue
93      call crecv(SIZETYPE, size, CUBESIZE)
94      worknodes = size
95
96c
(host:100) >
```

LIST *(cont.)*

3. Change the current list file to *prompt.f* and list lines #17 through #31.

```
(host:100) > file prompt.f
(host:100) > list #17,#31
  (host:100)
17      print *
18      print *, '******************** INTEGRATION EXAMPLE **************
19     >*********'
20      print *
21      print *, 'This Example uses the iPSC to calculate pi by integrati
22     >ng     '
23      print *, 'the function:'
24      print *, '              f(x) = 4 / (1 + x**2)'
25      print *, 'between x=0 and x=1.'
26      print *, 'The method used is the n-point rectangle quadrature rul
27     >e.'
28      print *
29      write(6,99)
3099    format(' How many points do you want (0 or neg. value quits)? ')
31      read (5, 110) points
(host:100) >
```

Note that if you now issue the **list** command and specify a context, source lines are still displayed from *prompt.f*.

```
(host:100) > list (nodes:0)
  (nodes:0)
1c
2c   prompt.f 6.1 89/03/30 04:02:21
3c
4c   Function which prompts the user for the integration parameters
5c
6
7       function userinpt (a, b, points, size)
8
9       integer*4 userinpt
10
11      double precision a, b
(host:100) >
```

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

But if you run the node program, causing it to encounter a breakpoint, list once again observes the current and specified context.

```
(host:100) > run (nodes:0)
        (0:0) : stopped at MAIN_()#93   (bp# 3)
(host:100) > list
  (host:100)
84          call crecv(PARTTYPE, msg, MSGSIZE)
85          integral = a
86
87c clean out unreceived messages
88c         call flushmsg(-1, ALLNODES, APPLPID)
89
90          write(6, 110) integral
91110       format('  pi is approximately : ', F18.16)
92
93c
94c Figure out elapsed time.  Total milliseconds were returned
(host:100) > list (nodes:0)
  (nodes:0)
93      call crecv(SIZETYPE, size, CUBESIZE)
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
103c    beginning to wait for another chance to work
(host:100) >
```

LOAD

LOAD

Loads a process onto the nodes, but does not run the process.

Syntax

load [-p *pid*] [*nodes*] *filename*

Arguments

-p *pid* — The NX/2 process ID (pid) assigned to the node process. Valid *pids* include any non-negative integer value. If you do not specify a *pid*, a value of 0 is assumed. To load multiple processes on a single node, you must assign each a unique *pid*.

nodes — List of nodes onto which the process is to be loaded. Separate multiple node numbers with a space. If no nodes are specified, all nodes in the allocated cube are loaded.

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

Notes

None

Examples

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

```
decon> load node
        decon: loading node program ......
        decon: program loaded ......
```

2. Load the file *node* on nodes 0 and 1 and assign it pid 50.

```
decon> load -p 50 0 1 node
        decon: loading node program ......
        decon: program loaded ......
```

LOG

Turns on or off a debug log file. Displays the name of the current log file.

Syntax

Arguments

on *filename*	Specifies the name of the file that will store the debug log. *filename* may be a complete or relative pathname.
off	Turns off logging to the current log file.

Notes

Without arguments, **log** displays the name of the current log file.

Only one log file can be active at a time.

If you specify the name of an existing file, its contents will be overwritten, not appended to.

Only DECON commands and their responses are logged. Application program output and user input to the application program are not logged. Note, however, that output of the DECON **system** command is logged.

Examples

1. Turn on logging to file *log007*.

```
(host:100) > log on log007
```

2. Display the name of the current log file.

```
(host:100) > log
        **** log file = log007
```

3. Turn off logging.

```
(host:100) > log off
```

MSGQ MSGQ

Displays messages currently queued.

Syntax

msgq [*context*] [*type*]

Arguments

context This is the context as defined in the **context** command. It is specified as follows:

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

Refer to the description of the **context** command for more information.

type The message type. When you specify *type*, **msgq** displays only those messages of that type that are in the queue.

Notes

Note that **msgq** displays only the messages in the current or specified context. Messages are in context if the destination (node:pid) is.

msgq displays messages that have been sent but not yet received. Use **recvq** to display which processes have posted receives that have not been satisfied.

MSGQ *(cont.)* MSGQ *(cont.)*

Examples

1. Assume that the current context is (nodes:0), that two messages, as yet unreceived, have been sent to each node, and that your allocated cube has four nodes. Display the message queue for process 0 on all nodes.

```
(nodes:0) > msgq

                For                  From             Type        Size
        =================   =================   =========   =========
  Pid 0:
        (   2:         0)   (host:        100)          0           4
        (   2:         0)   (host:        100)          5          20
        (   3:         0)   (host:        100)          0           4
        (   3:         0)   (host:        100)          5          20
        (   1:         0)   (host:        100)          0           4
        (   1:         0)   (host:        100)          5          20
        (   0:         0)   (host:        100)          0           4
        (   0:         0)   (host:        100)          5          20
(nodes:0) >
```

2. Now display type 5 messages in the message queue for the same process on all nodes.

```
(nodes:0) > msgq 5

                For                  From             Type        Size
        =================   =================   =========   =========
  Pid 0:
        (   2:         0)   (host:        100)          5          20
        (   3:         0)   (host:        100)          5          20
        (   1:         0)   (host:        100)          5          20
        (   0:         0)   (host:        100)          5          20
(nodes:0) >
```

PROCESS PROCESS

Displays information about user processes controlled by DECON. Displays the process status, the load file name, and the NX/2 pid.

Syntax

process

Arguments

None

Notes

None

Examples

1. Display process information after a host breakpoint. Notice that the node program is loaded but has not yet executed.

```
        (host:100) stopped at MAIN()#84    (bp# 1)
(nodes:0) > process
        pid     load file                proc status
        ===     =========                ===========
        100     host                     stopped on HOST
          0     node                     loaded on ALL nodes
```

2. Run the node program, hitting another breakpoint. Display process information. Notice that the node program has executed and is now stopped.

```
(nodes:0) > run
        (0:0) : stopped at MAIN()#93    (bp# 3)
(nodes:0) > process
        pid     load file                proc status
        ===     =========                ===========
        100     host                     stopped on HOST
          0     node                     stopped on ALL nodes
(nodes:0) >
```

QUIT QUIT

Terminates a debug session and exits DECON.

Syntax

quit

Arguments

None

Notes

exit is a synonym for **quit**.

Examples

1. Exit DECON.

```
decon> quit
         **** decon is terminating
%
```

RECVQ

RECVQ

Displays user processes currently waiting for messages.

Syntax

recvq [*context*]

Arguments

context This is the context as defined in the **context** command. It is specified as follows:

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

Refer to the description of the **context** command for more information.

Notes

Note that **recvq** displays only the messages in the current or specified context. Messages are in context if the destination (node:pid) is.

recvq displays the processes that have posted receives that have not been satisfied. Use **msgq** to display messages that have been sent but not received.

Examples

1. Consider the Fortran pi example. Run the node program, but do not start up the host program. The node program stops just before its first receive. Start up the node program again. It blocks at the first receive because the host has not yet sent the message. Send an interrupt signal (usually <Del>, sometimes <CTRL-C>) and regain the DECON prompt. Display the receive queue and notice that both nodes are waiting for a message.

```
(nodes:0) > run
        (0:0) : stopped at MAIN()#93    (bp# 3)
(nodes:0) > msgq
                For                 From            Type        Size
        ================  ================  =========  =========
   Pid 0:
                            Node 1 has no messages
                            Node 0 has no messages
```

RECVQ *(cont.)*

RECVQ *(cont.)*

```
(nodes:0) > run
        **** interrupted ...
(nodes:0) > recvq
                   Waiting Pid            Message Type
        (1)             0                          0
        (0)             0                          0
(nodes:0) >
```

REMOVE

Removes breakpoints.

Syntax

remove [*context*] (*brkpt_no* [,*brkpt_no*] ...| **all**)

Arguments

context — This is the context as defined in the **context** command. It is specified as follows:

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

Refer to the description of the **context** command for more information.

brkpt_no — Specifies the breakpoints to be removed. To determine the breakpoint number, use the **break** command.

Notes

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

Examples

1. Display all current breakpoints. Then remove breakpoints 1 and 2 on the host.

```
(nodes:0) > break (all:all)
  (nodes:0)
        bp no   active  action  object                   on      off
        =====   ======  ======  ======                   ==      ===
        3       1       break   node.f{}MAIN()#93        All     None
        4       1       break   node.f{}MAIN()#99        All     None
  (host:100)
        bp no   active  action  object                   on      off
        =====   ======  ======  ======                   ==      ===
        1       1       break   host.f{}MAIN()#84        All     None
        2       1       break   host.f{}MAIN()#90        All     None
(nodes:0) > remove (host:100) 1,2
(nodes:0) >
```

RUN RUN

Starts or continues execution of processes in the current or specified context.

Syntax

run [*context*]

Arguments

context This is the context as defined in the **context** command. It is specified as follows:

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

Refer to the description of the **context** command for more information.

Notes

A read executed on a node is only successful during a **run** or **step**. Once DECON's user prompt appears, the nodes will not accept keyboard input. You should use breakpoints and context to ensure that no process runs in parallel with a read section in another process, unless the process is guaranteed to hang waiting for a message from the read process.

When you enter **run**, control returns (The DECON prompt appears) when the first event (for example, a breakpoint encounter) occurs. When control returns, other processes may still be running.

It is an error to use the **run** command in a context containing running processes. Use **wait** to ensure that all processes are stopped.

Examples

1. Start up the host process with NX/2 pid 100 when the current context is (nodes:0).

```
(nodes:0) > run(host:100)
(nodes:0) >
```

2. Start up all processes in the current context.

```
(nodes:0) > run
```

SET

Sets or displays debug variables.

Syntax

set [*debug_var string*]

Arguments

debug_var — The symbolic name of the debug variable you are defining. To use a debug variable later, precede *debug_var* with a $.

string — The string that will be represented by the variable.

Notes

Without arguments, **set** displays current debug variables. Use the **unset** command to delete debug variables.

Examples

1. Define the debug varible *nvar* as (nodes:0). Then, use this debug variable in the **context** command.

```
(host:100) > set nvar (nodes:0)
(host:100) > context $nvar
(nodes:0) >
```

2. Display the current debug variables.

```
(nodes:0) > set
        Dvars   Contents
        =====   ========
        nvar    (nodes:0)
```

SOURCE SOURCE

Sets the current source directory. Displays the name of the current list directory.

Syntax

source [*directory*]

Arguments

directory The path name of the directory containing the application source files. The default is the directory in which you invoked DECON.

Notes

Without arguments, **source** displays the name of the current source directory.

Examples

1. Display the current source directory.

```
(nodes:0) > source
        **** Current Source Directory is CURRENT DIRECTORY
(nodes:0) >
```

2. Display the current source directory. Get a list error. Change the current source directory and successfully list the node program.

```
(nodes:0) > source
        **** Current Source Directory is CURRENT DIRECTORY
(nodes:0) > list
        **** error : file '/usr/you/Fpi/node.f' not found
(nodes:0) > source src
(nodes:0) > source
        **** Current Source Directory is src
(nodes:0) > list
  (nodes:0)
57      program node
58
59      include 'fcube.h'
60
61      integer SIZETYPE, INITTYPE, PARTTYPE, MSGSIZE, CUBESIZE,
```

SOURCE *(cont.)*

SOURCE *(cont.)*

```
62     >                    HOST, HOSTPID, APPLPID, DOUBLESIZE
63
64      integer*4 worknodes, mynod, pid, size
65      integer*4 basicpoints, extrapoints, mypoints, i, j
66      integer*4 starttime, points
67      integer*4 msg(5)
(nodes:0) >
```

STATUS STATUS

Displays the status of node programs.

Syntax

status

Arguments

None

Notes

Because DECON regains control as soon as the first process (either a node or host process) encounters a breakpoint or terminates, DECON displays the current messages, but saves other incoming messages in a message buffer. The **status** command displays the messages in this buffer.

Examples

1. Assume that the allocated cube has two nodes. Also assume that both nodes run the same program. Run the node program, hitting a breakpoint. Both hit the breakpoint, and you see the message from process 0. Use **status** to get the other messages.

```
(nodes:0) > run
        (0:0) : stopped at MAIN()#93    (bp# 3)
(nodes:0) > status
        (1:0) : stopped at MAIN()#93    (bp# 3)
```

STEP

Single steps the processes one executable source line in the current or specified debug context.

Syntax

step [*context*] [**call**]

Arguments

context This is the context as defined in the **context** command. It is specified as follows:

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

Refer to the description of the **context** command for more information.

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

Notes

System calls are treated as if the command were **step call**, even if you did not specify **call**.

Unlike **run**, when you step a program, you get all the status messages. There are no extra messages in the message buffer. **run** returns immediately after an event (program termination or breakpoint encounter); **step** does not return until all processes in its context have stepped.

Examples

1. Assume that the node program is stopped at #93, just before a **crecv()**. Step through the **crecv()** and pick up the message. Then, step to the next **crecv()**.

```
(nodes:0) > run
        (0:0) : stopped at MAIN()#93    (bp# 3)
(nodes:0) > list
  (nodes:0)
93      call crecv(SIZETYPE, size, CUBESIZE)
94      worknodes = size
95
96c
```

STEP *(cont.)*

STEP *(cont.)*

```
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
103c    beginning to wait for another chance to work
(nodes:0) > msgq

                For                    From               Type        Size
         =================    =================     =========   =========
  Pid 0:
         (   1:         0)    (host:       100)             0           4
         (   1:         0)    (host:       100)             5          20
         (   0:         0)    (host:       100)             0           4
         (   0:         0)    (host:       100)             5          20
(nodes:0) > step
        (0:0) stopped at MAIN()#94
        (1:0) stopped at MAIN()#94
(nodes:0) > msgq

                For                    From               Type        Size
         =================    =================     =========   =========
  Pid 0:
         (   1:         0)    (host:       100)             5          20
         (   0:         0)    (host:       100)             5          20
(nodes:0) > step
        (0:0) stopped at MAIN()#99
        (1:0) stopped at MAIN()#99
(nodes:0) >
```

STOP

Stops program execution in the current or specified context.

Syntax

stop [*context*]

Arguments

context — This is the context as defined in the **context** command. It is specified as follows:

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

Refer to the description of the **context** command for more information.

Notes

If you don't have a DECON prompt, stopping program execution may require two steps.

1. Send an interrupt signal to allow DECON to regain control. The interrupt signal is user-configurable. It is often **<Del>**, but sometimes **<CTRL-C>**. The DECON prompt appears.
2. Issue the **stop** command.

Examples

1. Run the Fortran pi example. Run the node program until it blocks at its first receive. Send an interrupt signal. Then, issue the **stop** command.

```
% decon
        **** iPSC/2 Concurrent Debugger R3.1
decon> exec picf
decon> ++ hload 100 host
        decon: loading host program ......
        decon: program loaded ......
decon> ++ load node
        decon: loading node program ......
        decon: program loaded ......
decon> ++ context (host:100)
(host:100) > ++ break #84
```

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

```
(host:100) > ++ break #90
(host:100) > ++ context (nodes:0)
(nodes:0) > ++ break #93
(nodes:0) > ++ break #99
(nodes:0) > run(host:100)
 LOADING THE CUBE ...

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

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

 How many points do you want (0 or neg. value quits)?
100
 What cube size (1-  2) should I use ?
2
        (host:100) stopped at MAIN()#84   (bp# 1)
(nodes:0) > run
        (0:0) : stopped at MAIN()#93   (bp# 3)         Nodes encounter a
                                                   breakpoint before first
                                                    receive  but there are
                                                      no messages waiting.

(nodes:0) >process
        pid     load file                   proc status
        ===     =========                   ===========
        100     host                        stopped on HOST

          0     node                        stopped on ALL nodes
(nodes:0) > run                              Node programs block at first receive.
   **** interrupted ...                          Issue a <Del> to regain control.
(nodes:0) > process
        pid     load file                   proc status
        ===     =========                   ===========
        100     host                        stopped on HOST

          0     node                        running on ALL nodes
(nodes:0) > stop
(nodes:0) > process
        pid     load file                   proc status
        ===     =========                   ===========
        100     host                        stopped on HOST
          0     node                        stopped on ALL nodes
(nodes:0) >
```

SYSTEM SYSTEM

Executes a UNIX command.

Syntax

system *command*

Arguments

command The UNIX command to be executed.

Notes

The *command* is not interpreted by DECON. The *command* is passed directly to */bin/sh*.

If a log file is active, output from this command is logged.

Examples

1. Issue the UNIX command ls -l from within DECON.

```
decon > system ls -l /usr/ipsc/examples/c/pi
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 fx.f
-r--r--r--   1 root     other       2880 Mar 30 21:03 host.f
-r--r--r--   1 root     other        475 Mar 30 21:03 makefile
-r--r--r--   1 root     other       4154 Mar 30 21:03 node.f
-r--r--r--   1 root     other       1160 Mar 30 21:03 prompt.f
decon>
```

TYPE TYPE

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

Syntax

type [*context*] [**-p**] *variable* [,*variable*] ...

Arguments

context This is the context as defined in the **context** command. It is specified as follows:

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

Refer to the description of the **context** command for more information.

variable The symbolic name of a program variable. Note that if the variable name in your program has a dollar sign ($), you must replace the $ with an underscore (_) when referring to the variable from within DECON. This is because the C and Fortran compilers convert $ to _.

-p Causes type details of a C **struct** variable to be displayed.

Notes

None

TYPE *(cont.)*

TYPE *(cont.)*

Examples

1. Determine the type of the variable *tms* in process 100 in the host program.

```
(nodes:0) > type (host:100) tms
integer*4 tms
(nodes:0) >
```

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

```
(host:100) > type msg
struct msg_type msg;
(host:100) > type -p msg
struct msg_type  {
        double a;
        double b;
        int points;
} msg;
(host:100) >
```

UNALIAS UNALIAS

Deletes previously defined aliases.

Syntax

unalias *alias_name* [,*alias_name*] ... | **all**

Arguments

alias_name A string that you chose as another name for a DECON command with the **alias** command.

all Removes all currently defined aliases.

Notes

None

Examples

1. Remove the alias **c**.

```
(nodes:0) > alias
        Alias   Contents
        =====   ========
        c       context
(nodes:0) > unalias c
(nodes:0) > alias
        Alias   Contents
        =====   ========
(nodes:0) >
```

UNSET

Deletes previously defined debug variables.

Syntax

unset *debug_var* [,*debug_var*] ... | **all**

Arguments

debug_var	The symbolic name of the debug variable you are deleting. Do not precede it with a $.
all	Removes all currently defined debug variables.

Notes

None

Examples

1. Delete the debug variable *nvar*.

```
(host:100) > set
        Dvars   Contents
        =====   ========
        nvar    (nodes:0)
(host:100) > unset nvar
(host:100) > set
        Dvars   Contents
        =====   ========
(host:100) >
```

WAIT WAIT

Wait until all processes stop running.

Syntax

wait

Arguments

None

Notes

This command is most useful if you have several node processes that don't all finish at the same time. For example, assume you start all the node processes running. The DECON prompt returns after the first process encounters a breakpoint. A **status** command displays the message buffer, but the other processes may not yet have stopped, and their messages are not yet in the message buffer. You can keeping issuing **status** commands until you see all the messages, or you can issue a **wait** command and wait.

wait is not context sensitive.

Examples

1. Issue a **run**. When you see the breakpoint message, issue a **wait**. When all the other processes complete, the rest of the messages appear.

```
(nodes:0) > run
        (0:0)  : stopped at MAIN()#93   (bp# 3)
(nodes:0) > wait
        (1:0)  : stopped at MAIN()#93   (bp# 3)
        (2:0)  : stopped at MAIN()#93   (bp# 3)
        (3:0)  : stopped at MAIN()#93   (bp# 3)
(nodes:0) > wait
       **** All nodes are stopped
```

DECON EXAMPLE 3

INTRODUCTION

This chapter contains an example of how DECON can find what has turned out to be a frequent bug in concurrent applications.

The source code for the Fortran version of the DECON example is listed in Appendix A. It is also contained in */usr/ipsc/examples/f/pi/decon*. The program calculates π. It uses an n-point quadrature rule to evaluate the definite integral,

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

The example consists of a host program that interfaces to the user and loads the same node program on each of the allocated nodes. Each node performs a portion of the integration.

The host program sends two messages to every node program. The first message (type 0) contains the number of nodes that are to work on the problem. The second message (type 5) contains the integration limits and the number of discrete rectangles that approximate the integral. You can improve the accuracy of the calculation by increasing the number of rectangles.

The node program then performs its calculation and executes a global sum operation (**gdsum()**). Node 0 then sends that global sum (the answer) to the host, which displays the result on the screen along with some timing statistics. Figure 3-1 illustrates this control flow.

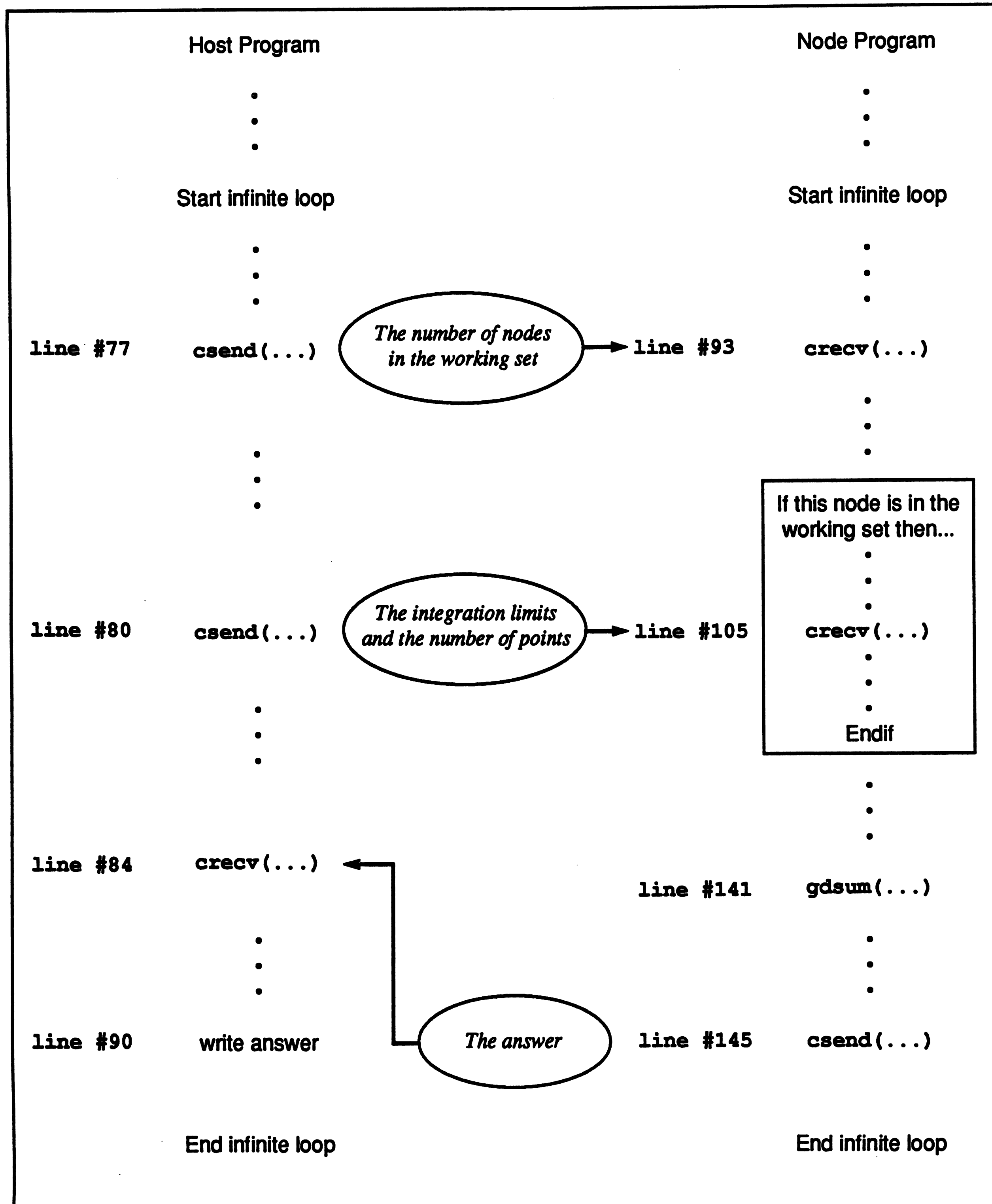


Figure 3-1. Control Flow for the DECON Demonstration Program

PREPARING THE EXAMPLE

Copy the contents of */usr/ipsc/examples/f/pi/decon* to your own area and issue the **make** command. The executable files *dhost* and *dnode* are created.

The file *dnode.f* has the bug. For debugging purposes, the file *dhost.f* has the calls **load()** and **killcube()** commented out.

```
% make
        f77 -B  -c dhost.f
dhost.f:
        f77 -B  -c dprompt.f
dprompt.f:
        f77 -o dhost dhost.o dprompt.o -host
        f77 -B  -c dnode.f
dnode.f:
        f77 -B  -c dfx.f
dfx.f:
        f77 -o dnode dnode.o dfx.o -node
%
```

INVOKING DECON

The example assumes that you allocate a cube with four nodes and invoke DECON from the same directory that contains the example's source and executable files.

```
% getcube -t4
getcube successful: cube type 4m4n0 allocated
% decon
        **** iPSC/2 Concurrent Debugger R3.1
decon>
```

LOADING FILES AND SETTING BREAKPOINTS

Load the host and node programs. The host program's NX/2 pid is 100, and the node program's NX/2 pid is 0.

```
decon> hload 100 dhost
        decon: loading host program ......
        decon: program loaded ......
decon> load dnode
        decon: loading node program ......
        decon: program loaded ......
decon>
```

Set the breakpoints. The breakpoints in *dhost* are at lines #84 and #90. The host program receives the answer from Node 0 and writes it on the screen. Line #84 is the crecv() from Node 0. Line #90 is the write to the screen.

```
decon> context(host:100)
(host:100) > break #84
(host:100) > break #90
(host:100) > context(nodes:0)
(nodes:0) > break #93
(nodes:o) > break #105
(nodes:0) > break(all:all)
      bp no   active  action  object                   on      off
      =====   ======  ======  ======                   ==      ===
  (nodes:0)
        3       1      break   node.f{}MAIN()#93        All     None
        4       1      break   node.f{}MAIN()#105       All     None
  (host:100)
        1       1      break   host.f{}MAIN()#84        All     None
        2       1      break   host.f{}MAIN()#90        All     None
(nodes:0) >
```

RUNNING THE HOST PROGRAM

Run the host program. It requests the number of points and the number of nodes. Choose 100 for the number of points and choose 2 for the number of nodes. Note that the allocated cube has four nodes, but you chose only two. The host program then encounters the breakpoint at its crecv().

```
(nodes:0) > run (host:100)
 LOADING THE CUBE ...

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

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

 How many points do you want (0 or neg. value quits)?
100
 What cube size (1-  4) should I use ?
2
        (host:100) stopped at MAIN()#84    (bp# 1)
(nodes:0) >
```

RUNNING THE NODE PROGRAM

Now run the node program. It stops just before its first receive at line #93. Display the message queue.

```
(nodes:0) > run
        (0:0) : stopped at MAIN()#93    (bp# 3)
(nodes:0) > msgq

                   For                    From                  Type         Size
         ==================    ==================      =========    =========
 Pid 0:
         (    2:           0)  (host:          100)            0            4
         (    2:           0)  (host:          100)            5           20
         (    3:           0)  (host:          100)            0            4
         (    3:           0)  (host:          100)            5           20
         (    1:           0)  (host:          100)            0            4
         (    1:           0)  (host:          100)            5           20
         (    0:           0)  (host:          100)            0            4
         (    0:           0)  (host:          100)            5           20
(nodes:0) >
```

Notice that each node has two waiting messages. Start up the node program again and display the message queue.

```
(nodes:0) > run
        (0:0) : stopped at MAIN()#105    (bp# 4)
(nodes:0) > msgq

                   For                    From                  Type         Size
         ==================    ==================      =========    =========
 Pid 0:
         (    2:           0)  (host:          100)            5           20
         (    3:           0)  (host:          100)            5           20
         (    1:           0)  (host:          100)            5           20
         (    1:           0)  (    3:           0)   1000000002            8
         (    0:           0)  (host:          100)            5           20
         (    0:           0)  (    2:           0)   1000000002            8
(nodes:0) >
```

You expect to see only type 5 messages, but there are two extra messages. To find out what's happening, take a look at the *dnode.f* source. One way to do this from within a DECON session is to issue the system command with vi.

```
(nodes:0) > system vi dnode.f
```

The pertinent area is as follows:

•
•
•

```
 91  c
 92  100     continue
 93          call crecv(SIZETYPE, size, CUBESIZE)
 94          worknodes = size
 95
 96  c
 97  c       if this node is not among the worker nodes it returns to the
 98  c       beginning to wait for another chance to work
 99  c
100          partialint = 0.0
101          if (mynod .lt. worknodes) then
102  c
103  c       receive integration parameters
104  c
105              call crecv(INITTYPE, msg, MSGSIZE)
106
107  c start the clock running
108              starttime = mclock()
109
110  c calculate slice size:
111              slicesize = (b - a) / points
112
113  c calculate # of points per node & local sub-interval:
114              basicpoints = points/worknodes
115              extrapoints = mod(points, worknodes)
116
117              if (mynod.lt.extrapoints) then
118                 mypoints = basicpoints + 1
119                 mya = a + slicesize * mynod * mypoints
120              else
121                 mypoints = basicpoints
122                 mya = a + slicesize * (mynod * mypoints + extrapoints)
123              endif
124
125              myb = mya + slicesize * mypoints
126
```

```
127  c calculate the partial integral of the function over my sub-interval:
128              do 200 i=1, mypoints
129                 x = mya + (i - 0.5D0)*slicesize
130                 partialint = partialint + f(x) * slicesize
131  200         continue
132          endif
133
134  c if we're node 0, gather all the contributions from all the other
135  c nodes, add them up, and send the result to the host
136  c also, figure elapsed time and send in message
137  c
138  c the nodes that didn't do any work also participate in the gdsum(),
139  c but their partialint is 0.0
140
141          call gdsum(partialint, 1, work)
142          if (mynode() .eq. 0) then
```

•
•
•

Notice that because you chose only two of the four nodes to do the work, two unchosen nodes do not enter the **if** to reach the breakpoint at line #105. Nodes 2 and 3 go right to the the **gdsum()** where they block. The message type `1000000002` is a reserved type for NX/2 internal messages and is used by **gdsum()**. The code is written such that even nodes that do not participate in the calculation perform the **gdsum()**.

The important point to notice is that all four nodes are sent type 5 messages, but two of the nodes (2 and 3) go directly to the **gdsum()** without executing a receive.

Blocked processes are still listed as running. Quit **vi** and issue the **process** command to verify this.

```
(nodes:0) > process
       pid     load file                 proc status
       ===     =========                 ===========
       100     host                      stopped on HOST

         0     node                      running on (2..3); stopped on (0..1)
(nodes:0) >
```

Nodes 0 and 1 are stopped at a breakpoint at line #105. Nodes 2 and 3 are blocked at the **gdsum()**. Because the **gdsum()** is a global operation, the nodes cannot progress beyond the **gdsum()** until all nodes reach that point. Start up Nodes 0 and 1. They then drop down to the **gdsum()**. The global sum takes place, and all nodes reach the GOTO and then stop at line #93.

```
(nodes:0) > run(0,1:0)
        (0:0) : stopped at MAIN()#93    (bp# 3)
```

All nodes are now ready to begin a new calculation. But display the message queue and notice that nodes 2 and 3 never picked up their messages.

```
(nodes:0) > msgq

                For                     From              Type         Size
         =================     =================     =========    =========
  Pid 0:
         (    2:          0)   (host:         100)           5           20
         (    3:          0)   (host:         100)           5           20
                                 Node 1 has no messages
                                 Node 0 has no messages
(nodes:0) >
```

BACK TO THE HOST PROGRAM

Change the context to (host:100). List its message queue and notice that it has a waiting message. This is the answer. Restart the host program. It picks up its message and encounters the breakpoint at line #90, just before it writes the answer to the screen. The answer is in the variable *integral.*

```
(nodes:0) > context(host:100)
(host:100) > msgq

                For                     From              Type         Size
         =================     =================     =========    =========
                100                       0                 20           20
(host:100) > run
        (host:100) stopped at MAIN()#90    (bp# 2)
(host:100) > msgq

                For                     From              Type         Size
         =================     =================     =========    =========
                Message Queue Empty
```

```
(host:100) >(host:100) > display integral
        (host:100) integral = 3.14160099e+00
(host:100) >
```

Restart the host program. It prints out the answer and is ready to perform another calculation. So are the node processes.

Terminate the host program, exit DECON, and release the cube.

```
(host:100) > run
  pi is approximately : 3.1416009869231249
  elapsed time =    0 min.  0.004 sec.

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

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

 How many points do you want (0 or neg. value quits)?
0
 PLEASE WAIT WHILE I CLEAN OUT THE CUBE ...
        (host:100) : program terminated
(host:100) > quit
        **** decon is terminating
% relcube
relcube released 1 cube
%
```

THE BUG

The host process and node processes were ready for another iteration. But the node's message queue still contained those two unreceived messages. If the next time around you chose four nodes, and Nodes 2 and 3 execute the second receive, they will get those old messages and not the new ones. This is the bug.

The NX/2 operating system takes great pains not to lose messages. You should design your application so that you don't have unreceived messages hanging around. One possible solution is to place the second **crecv()** outside the **if** statement so that it gets executed by all nodes.

```
                    .
                    .
                    .
c
c       receive integration parameters
c
        call crecv(INITTYPE, msg, MSGSIZE)

c
c       if this node is not among the worker nodes it returns to the
c       beginning to wait for another chance to work
c
        partialint = 0.0
        if (mynod .lt. worknodes) then

c start the clock running
            starttime = mclock()

c calculate slice size:
            slicesize = (b - a) / points

                    .
                    .
                    .
```

SOURCE CODE FOR THE DECON EXAMPLE A

MAKEFILE

```
1  #
2  # makefile 6.1 89/03/30 04:01:54
3  #
4  # This file is used to compile and link the host.f, prompt.f,
5  # fx.f, and node.f files for the pi numerical integration example.
6  #
7  # The command "make all" causes compilation and linking.
8
9
10  F77FLAGS = -B
11  all :    dhost dnode
12
13  dhost:   dhost.o dprompt.o
14           f77 -o dhost dhost.o dprompt.o -host
15
16  dnode:   dnode.o dfx.o
17           f77 -o dnode dnode.o dfx.o -node
18
19  clean:
20           rm dhost dnode dhost.o dnode.o dprompt.o dfx.o
```

DHOST.F

```
1   c
2   c host.f 6.2 89/03/30 15:35:31
3   c
4   c This program calculates the value of pi, using numerical integration
5   c with parallel processing, and clocks the solution time.
6   c
7   c The user selects the number of processors and the number of points
8   c of integration.  By selecting and timing different cube sizes you
9   c obtain a measure of the speedup with perfectly parallel problems.
10  c
11  c
12  c   Program Components:
13  c
14  c   Constants:
15  c
16  c       APPLPID      Application process id
17  c       INITTYPE     Type of initial message to the cube
18  c       MSGSIZE      Size of initial message to the cube (in bytes)
19  c       ALLNODES     Used to load all nodes in cube with a node process
20  c       TIMEFACTOR   Conversion factor to obtain seconds from system clock
21  c
22  c   Variables:
23  c
24  c       points       number of points to use in the integration
25  c       a            lower limit of integration
26  c       b            upper limit of integration
27  c       integral     calculated integral
28  c       msg          message sent to and from cube. Holds a, b, & points
29  c
30  c   Calls:
31  c
32  c       userinpt     to get integration parameters from user
33  c       csend        to send messages to the cube
34  c       crecv        to receive messages from the cube
35  c       killcube     to kill processes in the cube
36  c
37          program host
38
39          include 'fcube.h'
40
41  c       external  userinpt
42          integer*4 userinpt
43
44          integer*4 APPLPID, SIZETYPE, INITTYPE, MSGSIZE,
45       >            CUBESIZE, ALLNODES, HOSTPID, PARTTYPE
46
```

```
47          integer*4 tms, ms, tsec, sec, min, i
48          integer*4 size, points
49          integer*4 msg(5)
50
51          double precision  integral, a, b
52    c
53    c     equivalence a, b, and points to elements of msg
54    c
55          equivalence (msg(1), a)
56          equivalence (msg(3), b)
57          equivalence (msg(5), points)
58
59          data      APPLPID /0/,
60       >            SIZETYPE /0/,
61       >            INITTYPE /5/,
62       >            MSGSIZE /20/,
63       >            CUBESIZE /4/,
64       >            ALLNODES /-1/,
65       >            HOSTPID/100/,
66       >            PARTTYPE /20/
67
68          call setpid(HOSTPID)
69          print *, 'LOADING THE CUBE ...'
70    C     call load ('node', ALLNODES, APPLPID)
71
72    100   size = numnodes()
73          i = userinpt(a,b,points,size)
74          if (i.le.0) go to 200
75
76    c send size to use into cube
77                call csend(SIZETYPE, size, CUBESIZE, ALLNODES, APPLPID)
78
79    c send integration parameters into cube
80                call csend(INITTYPE, msg, MSGSIZE, ALLNODES, APPLPID)
81
82    c receive final integral value from cube (as 'a'), and the time,
83    c in milliseconds (as 'b')
84                call crecv(PARTTYPE, msg, MSGSIZE)
85                integral = a
86
87    c clean out unreceived messages
88    c           call flushmsg(-1, ALLNODES, APPLPID)
89
90                write(6, 110) integral
91    110         format('  pi is approximately : ', F18.16)
92
93    c
94    c Figure out elapsed time.  Total milliseconds were returned
```

```
95   c from node 0 in 'points'
96   c
97                  tms = points
98                  ms  = mod(tms, 1000)
99                  tsec = (tms - ms) / 1000
100                  sec  = mod(tsec, 60)
101                  min  = (tsec - sec) / 60
102
103                  write(6, 111) min, sec, ms
104   111            format('  elapsed time = ',i4,' min. ',i2,'.',i3.3, ' sec.')
105
106                  goto 100
107
108   200       continue
109
110             print *, 'PLEASE WAIT WHILE I CLEAN OUT THE CUBE ...'
111   C         call killcube(ALLNODES, APPLPID)
112             end
```

DPROMPT.F

```
1   c
2   c    prompt.f 6.1 89/03/30 04:02:21
3   c
4   c    Function which prompts the user for the integration parameters
5   c
6
7             function userinpt (a, b, points, size)
8
9             integer*4 userinpt
10
11             double precision a, b
12             integer*4 points
13             integer*4 size
14             integer*4 size1
15             integer*4 i
16
17             print *
18             print *, '******************** INTEGRATION EXAMPLE **************
19          >*********'
20             print *
21             print *, 'This Example uses the iPSC to calculate pi by integrati
22          >ng    '
23             print *, 'the function:'
24             print *, '                    f(x) = 4 / (1 + x**2)'
25             print *, 'between x=0 and x=1.'
26             print *, 'The method used is the n-point rectangle quadrature rul
27          >e.'
28             print *
29             write(6,99)
30   99        format(' How many points do you want (0 or neg. value quits)? ')
31             read (5, 110) points
32   110       format(i10)
33
34             a = 0.0
35             b = 1.0
36             i = 0
37
38             if (points .gt. 0) then
39   112            continue
40                  write(6,199) size
41   199            format(' What cube size (1-',i3,') should I use ? ')
42                  read(5,111) size1
43   111            format(i3)
44                  if (size1.gt.size) goto 112
45                  i = 1
46                  if (size1.le.0) i = 0
```

```
47            else
48                 i = 0
49            endif
50
51            size = size1
52            userinpt = i
53            return
54            end
```

DFX.F

```
1  c
2  c   fx.f 6.1 89/03/30 04:01:31
3  c
4  c   Function to integrate in the integration example.
5  c
6          double precision function f(x)
7          double precision x
8
9          f = 4.D0 / (1.D0 + x*x)
10
11          return
12          end
```

DNODE.F

```
1   c
2   c node.f 6.3 89/03/30 15:35:34
3   c
4   c This program calculates the value of pi, using numerical integration
5   c with parallel processing, and clocks the solution time.
6   c
7   c   This is the node program, which shows how the iPSC/2 nodes can be
8   c   used to calculate the definite integral of a function f(x).
9   c
10  c   Each node:
11  c
12  c       1) receives the integration parameters by invoking crecv,
13  c       2) calculates a partial integral over its sub-interval, and
14  c       3) sends its contribution to node 0, which adds the
15  c          integral and sends the total back to the host.
16  c
17  c
18  c   Program Components:
19  c
20  c   Constants:
21  c
22  c       INITTYPE        Type of initial message to the cube
23  c       PARTTYPE        Type of message holding partial integral
24  c       MSGSIZE         Size of messages to and from the cube
25  c
26  c   Variables:
27  c
28  c       worknodes       total number of participating nodes in the cube
29  c       mynod           node id returned by mynode()
30  c       size            holds cube size
31  c
32  c       points          total number of points to use in integration
33  c       basicpoints     basic number of points per node
34  c       extrapoints     extra points; to be assigned to lower nodes
35  c       mypoints        number of points assigned to a node process
36  c
37  c       a               lower limit of integration
38  c       b               upper limit of integration
39  c       mya             this node's lower limit of partial integration
40  c       myb             this node's upper limit of partial integration
41  c
42  c       msg             initial message received from host. Holds
parameters of
43  c                       integration (a, b, points)
44  c       x               independent variable of integration
45  c       f               function to integrate (external)
```

```
46  c       slicesize       size of integration slice
47  c       partialint      partial integral calculated on node
48  c       i               loop counter
49  c
50  c   External Routines:
51  c
52  c       f               to evaluate the function f(x) being integrated
53  c       mynode          to obtain the node id for this process
54  c       mypid           to obtain the process id of this process
55  c       numnodes        to determine the size of cube being used
56  c
57
58          program node
59
60          include 'fcube.h'
61
62          integer SIZETYPE, INITTYPE, PARTTYPE, MSGSIZE, CUBESIZE,
63        >           HOST, HOSTPID, APPLPID, DOUBLESIZE
64
65          integer*4 worknodes, mynod, pid, size
66          integer*4 basicpoints, extrapoints, mypoints, i, j
67          integer*4 starttime, points
68          integer*4 msg(5)
69
70          double precision x, slicesize, partialint, a, b, mya, myb
71          double precision work, sum
72
73          external  f
74          double precision f
75  c
76  c       make explicit the structure of msg:
77  c
78          equivalence (msg(1), a)
79          equivalence (msg(3), b)
80          equivalence (msg(5), points)
81
82          data SIZETYPE /0/, INITTYPE /5/, PARTTYPE /20/, MSGSIZE /20/,
83        >      CUBESIZE /4/, VECTORLEN / 1/, HOSTPID /100/,
84        >      APPLPID /0/, DOUBLESIZE /8/
85
86          HOST = myhost()
87          pid = mypid()
88          mynod = mynode()
89
90  c
91  c       get the size of the cube to put to work on problem
92  c
93  100     call crecv(SIZETYPE, size, CUBESIZE)
```

```
94          worknodes = size
95
96  c
97  c       if this node is not among the worker nodes it returns to the
98  c       beginning to wait for another chance to work
99  c
100          partialint = 0.0
101          if (mynod .lt. worknodes) then
102  c
103  c       receive integration parameters
104  c
105              call crecv(INITTYPE, msg, MSGSIZE)
106
107  c start the clock running
108              starttime = mclock()
109
110  c calculate slice size:
111              slicesize = (b - a) / points
112
113  c calculate # of points per node & local sub-interval:
114              basicpoints = points/worknodes
115              extrapoints = mod(points, worknodes)
116
117              if (mynod.lt.extrapoints) then
118                 mypoints = basicpoints + 1
119                 mya = a + slicesize * mynod * mypoints
120              else
121                 mypoints = basicpoints
122                 mya = a + slicesize * (mynod * mypoints + extrapoints
123              endif
124
125              myb = mya + slicesize * mypoints
126
127  c calculate the partial integral of the function over my sub-interval:
128              do 200 i=1, mypoints
129                 x = mya + (i - 0.5D0)*slicesize
130                 partialint = partialint + f(x) * slicesize
131  200         continue
132           endif
133
134  c if we're node 0, gather all the contributions from all the other
135  c nodes, add them up, and send the result to the host
136  c also, figure elapsed time and send in message
137  c
138  c the nodes that didn't do any work also participate in the gdsum(),
139  c but their partialint is 0.0
140
141           call gdsum(partialint, 1, work)
```

```
142         if (mynode() .eq. 0) then
143             a = partialint
144             points = mclock() - starttime
145             call csend(PARTTYPE, msg, MSGSIZE, HOST, HOSTPID)
146         endif
147
148         goto 100
149         end
150
```

INDEX

A

allocating a cube 3-3
array
    display 2-13
assign 2-4–2-5

B

break 2-6–2-8
breakpoint number 2-36

C

command file 2-16
configuration file 2-10
context 2-9
    current 2-9
    default 2-9

D

data breakpoint 2-7
decon 2-10–2-11
deconcf 2-10
dimension 2-12
display 2-13–2-15

E

exec 2-16
execution breakpoint 2-7

F

file 2-18–2-19, 2-24
Fortran -U 2-4, 2-13
Fortran common 2-13
frame 2-20–2-21

G

gdsum() 3-7
global sum 3-1

H

help 2-22
hload 2-23

I

invoking DECON 3-3

L

list 2-24–2-27
load 2-28
loading files 3-3
log 2-29
log file 2-46
lowercase symbols 2-4, 2-13

M

msgq 2-30–2-31, 2-34

P

preparing the example 3-3
process 2-32

Q

quit 2-33

R

recvq 2-30, 2-34–2-35
remove 2-36
run 2-37
running host program 3-4
running node program 3-5

S

set 2-38
setting breakpoints 3-4
source 2-39–2-40
status 2-41
step 2-7, 2-42–2-43
stop 2-44–2-45
system 2-29, 2-46
System Resource Manager 2-23

T

type 2-47

U

unalias 2-49
unset 2-38, 2-50

W

wait 2-37, 2-51