March, 1988
Order Number: 311569-001

# iPSC®/2

# CONCURRENT DEBUGGER

*(Preliminary)*

**ADVANCE INFORMATION**

This is a draft copy of the manual. Material in this document is for informational purposes only and is subject to change without notice. Intel Corporation assumes no responsibility for any errors which may appear in this document.

intel Corporation

Copyright © 1988 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 makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update or to keep current the information contained in this document.

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

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

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

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

EXOS is a trademark or equipment designator of Excelan, Inc.
XENIX is a trademark of Microsoft Corp.
UNIX is a trademark of AT&T
VAST-2 is a registered trademark of Pacific-Sierra Research Corp.
iPSC/2 is a registered trademark of Intel Corporation



| REV. | REVISION HISTORY | DATE |
|---|---|---|
| -001 | Original issue | 03/88 |

**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.

**CHAPTER 1 - THE CONCURRENT DEBUGGER**

Introduction 1-1

Contents 1-1

Overview 1-2

**CHAPTER 2 - DEBUGGER CAPABILITIES**

Introduction 2-1

The Debug Context 2-2

Inspecting Message Queues 2-3

Listing Outstanding Receive Requests 2-4

Setting Breakpoints & Tracepoints 2-5

Accessing and Changing Data 2-6

Other Capabilities 2-6

**CHAPTER 3 - DEBUGGER COMMANDS**

Symbols 3-2

Data Types 3-3

Debug Context 3-3

Process Loading Commands 3-4

Program Execution Control Commands 3-5

Data and Status Access Commands 3-8

Debugger Control Commands 3-11

**CHAPTER 4 - SAMPLE SESSIONS**

Introduction 4-1

Sample Session #1 4-2

Sample Session #2 4-3

# CHAPTER 1

# THE CONCURRENT DEBUGGER

## INTRODUCTION

This preliminary manual describes the iPSC/2 Concurrent Debugger, referred to as *"DECON"*, which is a tool designed to help developers debug their concurrent applications on iPSC/2 systems.

## CONTENTS

This manual is divided into the following chapters:


| | | |
|---|---|---|
| Chapter 1 | Introduction | Presents an overview of the debugger |
| Chapter 2 | Capabilities | Discusses major capabilities of the debugger |
| Chapter 3 | Commands | Discusses symbols, data types, and commands |
| Chapter 4 | Sample Session | Provides a sample debug session |

## OVERVIEW

The Concurrent Debugger offers developers a source-level debugging tool for their C and FORTRAN applications on iPSC/2 systems. It provides source-level debug support for user processes running on both the system resource manager (SRM) and the nodes, with minimum interference.

The Concurrent Debugger provides two levels of debugging capabilities:

- The first level concentrates on the sequential nature of the processes and provides debug capabilities on the user-defined objects usually seen in a sequential applications debugger. These include such functions as data access, conditional breakpointing/tracing, and execution monitoring. This level of debugging support can be used to debug one process at a time or processes which have little or no intercommunications.
- The second level provides a set of features at the *process* level designed specifically to assist you in debugging applications in the concurrent environment. This level can be used to debug process-level synchronization and communications activities in the user program. Among these additional features are process monitoring, process communications control, and node/message status monitoring.

In order to use the debugger, you must perform the following two steps:

1. Compile your C/FORTRAN programs using the -g and -X18 compiler option. This option directs the compiler to include symbolic information in the load module. The -X18 compiler option turns some of the compiler optimization phase.
2. You must not load your node programs in your host program. Comment out any ***load*** call to your host program.

# CHAPTER 2

# DEBUGGER CAPABILITIES

## INTRODUCTION

DECON supports all of the major capabilities found in advanced source level sequential debuggers, such as code, data and conditional breakpoints and tracepoints, single stepping program execution, symbolic program references, data and status interrogation, and other standard debugging capabilities.

This chapter, however, concentrates on those capabilities and features that are unique to DECON and which directly support the major debugging challenges and activities described above.

## THE DEBUG CONTEXT

The debug context defines the set of processes which are the target for each debug command. On the iPSC/2, each process is uniquely identified by the node id (nid) it is executing on and its assigned process id (pid). Processes are specified by providing their (nid:pid) combination. A context is then defined by providing a list of one or more (nid:pid) pairs to the context command.

For example, the command "context(127:1)" instructs DECON to apply subsequent debug commands to process 1 executing on node 127. Within DECON, the command (in bold characters) might be typed after the default "decon" prompt, like this:

decon> **context (127:1)**

To remind you of the new target for debug commands, DECON will replace its previous prompt, "decon", with the context set with the context command. Thus, after the above command the prompt will be:

(127:1)>

The context command allows you to easily and quickly change the focus of your attention to other processes. Thus, to change your focus to node 0, process 1, you simply type:

(127:1)> **context (0:1)**<br>
(0:1)>

To zoom out, and focus on processes with pid = 1 executing on all nodes type:

(0:1)> **context (nodes:1)**<br>
(nodes:1)>

You can also target all processes executing on all nodes simultaneously, including the host, by typing:

(0:1)> **context (all:all)**<br>
(all:all)>

## INSPECTING MESSAGE QUEUES

DECON lets you inspect the system message buffers where all messages arriving at a destination node are stored until they are received by the proper destination process. Any lost messages would be stored in these system buffers and can be detected by using the **msgq** (for message queue) command.

For example, to check for messages queued up for all processes executing on node 10 type:

decon> **msgq(10:all)**

or, to check for messages queued up on all nodes, including the host, type:

decon> **msgq(all:all)**

The msgq command not only lets you find lost messages, but also lets you trace which process(es) caused the messages to be lost in the first place. It does this by providing information on each message's originating process, intended destination, type, and size. The following example shows typical output from the msgq command:

decon> **msgq(nodes:all)**

| For | From | Type | Size |
|---|---|---|---|
| (0:0) | (host:1) | 10 | 4 |
| (0:1) | Message Queue Empty | | |
| (1:0) | (0:0) | 20 | 512 |
| (2:0) | Message Queue Empty | | |
| (3:0) | Message Queue Empty | | |

In this example, information is requested about messages queued for all process on the nodes of the system. In return, the messages queued on node 0, for process 0, or (0:0), and on node 1, for process 0, or (1:0) are shown. The processes originating each of the messages, each message's type, and size in bytes are also listed.

In the previous examples the msgq command is issued after the standard "decon" prompt. However, if a context had been set previously, the msgq command, as well as most other debug commands, will apply to the context reflected in the prompt. Thus, if the context had been previously set to (nodes:all), the following msgq command would have the same effect as the above msgq command:

(nodes:all)> **msgq**

An explicit context given in the msgq command overrides the "current context" displayed by the prompt. Thus, to examine the message queues for node 0 only, type:

(nodes:all)> **msgq(0:all)**

## LISTING OUTSTANDING RECEIVE REQUESTS

The **recvq** (for receive queue) command lists requests to receive messages that have not yet been satisfied with the arrival of the corresponding message. It is used in combination with the msgq command to identify lost messages and faulty requests for messages.

For example, to find out which processes on the nodes have not yet received messages they expected type:

(nodes:all) > **recvq**

| **Node** | **Waiting Pid** | **Message Type** |
|---|---|---|
| 0 | 0 | 20 |
| 1 | 0 | 30 |
| 2 | Wait Queue Empty | |
| 3 | Wait Queue Empty | |

As shown above, DECON lists which processes have outstanding requests to receive messages. The list includes information on the waiting process id, the node the process is on, and the type of the message requested. This allows incomplete requests to be easily matched with appropriate messages. In the example, process 0 on node 0 is waiting for a message of type 20, and process 0 on node 1 waiting for a message of type 30.

Together, the msgq and recvq commands allow the iPSC/2 programmer to quickly and easily detect most, if not all, of the message related bugs in iPSC/2 applications.

## SETTING BREAKPOINTS AND TRACEPOINTS

Code breakpoints and tracepoints can be set on statements, subroutines, or labels of multiple processes simultaneously. For example,

decon> **break (0..3:1) sub1()**

sets breakpoints at subroutine sub1() on processes with pid = 1 executing on nodes 0, 1, 2, 3.

You can also specify assertion-driven code breakpoints and tracepoints, such as:

decon> **break (nodes:0) #50 after 1000**

This means "break, on all nodes with a process 0, at statement #50 after it has executed 1000 times."

Data breakpoints and tracepoints can be set on user variables, either static or dynamic, and allow various event qualifiers, such as read, write or access. For example, the command

decon> **break (1:2) write flag**

will cause process 2, on node 1, to stop when the variable flag is written to. Similarly,

decon> **break (0:1,2) read count**

will cause processes 1 and 2 executing on node 0 to stop when variable count is read. And

decon> **break (1..8:0) var**

will cause process 0, on nodes 1 through 8, to stop when variable var is read or written.

Conditional breakpoints and tracepoints can also be set on variables as follows:

decon> **break (nodes:0) when i = 100**

This will stop process 0 on all nodes when the variable "i" is assigned the value 100.

The above capabilities have proven very useful in controlling, monitoring, and synchronizing the multiple processes executing concurrently on iPSC/2 nodes. They are also particularly useful in debugging loops and algebraic expressions found in concurrent applications.

## ACCESSING AND CHANGING DATA

Values of variables can be examined with the display command. As before, all variable references are symbolic. For example,

```
decon> display (1:0) A[2..10]
```

shows the values of elements 2 through 10 of array A on process 0 on node 1.

New values can be assigned to variables using the assign command, in two different ways. The first way lets you view the current value of the variable before changing it. This is illustrated in the following example:

```
decon> assign (1:0) A[2..10]
(1:0)  A[2]   = 10.0 <- 100.0
       A[3]   = 10.0 <- 100.0
       ...
       A[10]  = 10.0  <- 100.0
```

To change the current value (10.0), the new value (100.0) is typed in after the <- prompt. Otherwise, a return is typed after the <-, and the current value is not changed. This command continues until the last element is being accessed. You can use "$" to stop the command.

To deal with very large data structures or variables repeated across many processors, DECON also supports an "automatic" version of the assign command. This is illustrated in the following example:

```
decon> assign (1:0) A[2..10] 100.0
```

This command automatically assigns the value 100.0 to the elements 2 through 10 of array A in process 0 on node 1. Similarly, to assign a new value to a variable repeated across many nodes type

```
decon> assign (nodes:0) A[1..100] 0.0
```

This assigns the value 0.0 to elements 1 through 100 of array A in process 0 on all nodes of the system.

## OTHER CAPABILITIES

Other capabilities are also supported within DECON, including the ability to execute debug scripts that collect a set of commonly used debug commands, load, start and reset node and host processes, list source code, execute UNIX commands, keep a log of the debug session for later review, obtain on-line help on how to use each of the debug commands, and set aliases.

# CHAPTER 3

# DEBUGGER COMMANDS

## INTRODUCTION

This chapter defines the Concurrent Debugger's (*decon*) commands and command syntax. The following topics are covered in this chapter:

- Symbols
- Data Types
- Debug Context
- Process Loading Commands
- Program Execution Control Commands
- Data and Status Access Commands
- Debugger Control Commands

## SYMBOLS

*Decon* uses the following symbols to unambiguously interpret user objects.

| | | |
|---|---|---|
| <file> | = | <file_id> "{}" |
| <function> | = | <function_id> "()" |
| <statement> | = | "#" <number> |
| <label> | = | ":" <id> |
| <var_name> | = | {<file> \| <function> \| '@'} <var_name> |

First, an explanation of how *decon* interprets the user variables. Because there are three possible classes of storage for a user variable in C: local (auto), static, and public (extern), the user variable is represented as shown above.

If no qualifier is provided, *decon* will search for the specified variable id in the current block, then current function, then current file, and then the whole application. As soon as there is a match, the search stops. The variable found may be either local, static or public.

The <function> qualifier is used to identify local variables defined in the specified function. The specified function can be different from the current function but must have a frame on the run-time stack.

The <file> qualifier is used to identify static variables defined in the specified source file. The file id must be a valid source file name. The '@' qualifier is used to identify public variables visible to the whole application. These two qualifiers are useful for identifying variables with the same names.

Sometimes it is not desirable to address a complete structured variable when only a subrange of the variable is of interest. *decon* borrows the special symbols of ".." from Pascal to support addressing a subrange of components within a struct variable or an array variable.

<sub_array> = <array_id>("[" ¦ ")" )<starting_index> ".." <ending_index>" ("]" ¦ ")" )
<sub_struct> = <struct_id>"."<field_1> ".."<field_2>

<array_id> and <struct_id> are identifiers defined in the user's program. <starting_index> and <ending_index> are element indexes to an array variable. <field_1> and <field_2> are field identifiers of a structured variable.

The source statement is represented as follows:

{<file> | <function>} <statement>

If no <file> or <function> is provided, *decon* uses the source file in which the current block/current function is defined.

The <file> <statement> format identifies a source statement in the specified source file.

The <function> <statement> format identifies a source statement in the source file in which the specified function is defined.

The user-defined label is represented as follows:

{<function>} <label>

If no <function> is provided, *decon* tries to find the label from the current block/current function. Otherwise, the label defined in the specified function is used.

There are two file representations within decon. First, the <file> symbol defined in this section is mainly used to qualify variables and statements in the user program. The second file representation which does not require the curley brackets is used in commands such as load, exec, and log. It should be enclosed in single quotes if its name begins with special characters or numbers.

## DATA TYPES

*Decon* supports all data types defined both in C and FORTRAN 77, and follows the same scoping and formatting rules. In addition, *decon* allows you to display or modify a subrange of an array variable or the whole array variable with one debug command. There are also some differences. These are described below:

- When displaying a C union variable, the same data is applied to each member of the variable.

- Display of a printer variable of any type causes the address of the object it points to or 'nil' if it points to nothing. Also, decon supports only limited addressing operators. Allowable operators are: '*', '&', '[', ']'. For example, if list is defined as "char * list[10]", "list[1]. Note that list[1], *list[1], and **list[1] are legal, but *(list + 1) is not.

## DEBUG CONTEXT

Debugging in a concurrent multi-process environment often requires the ability to refer to several processes simultaneously. Because each user process can be uniquely identified by the node id (nid) it is running on and its process id (pid), the *debug context* is defined to be a set of the tuples (nid:pid). One example might be (0:1), which identifies process 1 on node 0. *nid* has some predefined strings: **nodes** means all nodes, **host** means system resource manager (host), and **all** means both nodes and host. *pid* has the predefined string **all**, which means all processes.

## PROCESS LOADING COMMANDS

**load** [-p <*pid*>] [<*nids*>] <*file*>

loads the user process into the cube but does not execute it. This is the same syntax that you normally use to load node processes. <*pid*> is the process number associated with the program and has the same value as mypid( ). If <*pid*> is not specified, the default process id 0 is assigned to the process. <*file*> is the name of the file to be loaded.

**hload** [<*pid*>] <*file*> [<<*ifile*>|><*ofile*>|<*args*>]

loads the user process into the system resource manager (or host). <*pid*> is the process number associated with the program and has the same value as mypid( ). If <*pid*> is not specified, the default process id 0 is assigned to the process. <*file*> is the name of the load module to be loaded. <*ifile*>, <*ofile*>, and <*args*> are input file, output file, and arguments respectively to the user program. Note that <*args*> are arguments to the host program and must be surrounded by single quotes.

**reset** [<*debug_context*>]

resets user processes. This command will put the process into a known state, i.e., as if it were just loaded. All previously set breakpoints are intact.

**kill** [<*debug_context*>]

terminates user process on the specified debug context. The current debug context is used when <*debug_context*> is not provided. This command can terminate process in any state, e.g. just loaded or running.

## PROGRAM EXECUTION CONTROL COMMANDS

**run** [<*debug_context*>]

starts or continues execution of programs on the specified debug context. If no debug context is provided, programs on the current debug context are started. Execution continues until either a breakpoint is encountered or the program terminates.

**stop** [<*debug_context*>]

stops program execution on the specified debug context. Stopping a running program requires two steps:

1. Use controlC to allow *decon* to regain control.
2. Use the stop command to stop program execution.

**step** [<*debug_context*>] [call] [<*nos*>]

single steps the programs on the specified debug context. If no debug context is provided, programs on the current debug context are stepped. <*nos*> indicates the number of source statements to be stepped through. **call** directs *decon* to treat subroutine calls as single statements. If it is not specified, the called routine will be entered and its statements will be stepped through.

## PROGRAM EXECUTION CONTROL COMMANDS (continued)

**break** [<*debug_context*>]
**break** [<*debug_context*>] (<*stmt_no*> | *label_no*> | <*routine_name*>) [after <exp>]
**break** [<*debug_context*>] [read | write] <*variable*>
**break** [<*debug_context*>] when <*condition* >

sets or displays breakpoints in the user programs. A breakpoint number will be assigned to the corresponding breakpoint and can be used in the **remove**, **act**, and **deact** commands.

The first form is used to display all set breakpoints under the specified debug context.

bp no  active  action  object  on  off

The <bp no> is a breakpoint number assigned by *decon* to the corresponding breakpoint or tracepoint (see "*trace*" on the next page) and can be used in the ***remove***, ***act***, and ***deact*** commands.

The <active> field is dictated if the breakpoint is currently active. A breakpoint can be set inactive with ***the deact*** command.

The <action> field shows "break" if it is a breakpoint or "trace" if it is a tracepoint.

The <object> field identifies the break object. A breakpoint can be set on individual nodes, a subset of nodes, or the entire cube. The <on> field can be used to show the domain where the breakpoint is defined. For example, if a breakpoint is set on node 0 of a 4-node subcube, the <on> field will show (0) and the <off> field will show (1...3). These two fields are meaningful to node processes only.

The second form sets a breakpoint at a source line number, a label, or a routine so that execution stops prior to that statement/label/routine being executed. Source line numbers must be preceded by the name of the source file if the file is not current, e.g., foo{}#10. Labels must be preceded by the name of the subroutine where they are defined if the subroutine is not currently being executed. The <after> clause instructs the breakpoint to occur every <exp> occurrence. Note that setting breakpoints on non-executable statements will cause an error.

The third form sets a breakpoint at a user variable (data breakpoint). The user variable can be either public or local. The attributes **read** and **write** further qualify the breakpoint such that execution should stop when the variable is being read or written. Note that execution stops after the variable has been accessed.

The fourth form sets the conditional breakpoints. The simple_var can be an integer type only. The <condition> has the following syntax:

<condition> ::= <*simple_var*> = <*exp*>

## PROGRAM EXECUTION CONTROL COMMANDS (continued)

**trace** [<*debug_context*>]
**trace** [<*debug_context*>] (<*stmt_no*> | <*label_no*> | <*routine_name*>)
**trace** [<*debug_context*>] [**read** | **write**] <*variable*>
**trace** [<*debug_context*>] when <*condition*>

sets or displays tracepoints in user programs. A breakpoint number will be assigned to the corresponding tracepoint and can be used in the **remove, act,** and **deact** commands. This command's function is identical to the **break** command with one exception: encountering a tracepoint causes tracing information to be displayed and program execution does not stop.

**act** [<*debug_context*> ] (*bkpt_no* {, *bkpt_no*} | all)

reactivates user breakpoints and tracepoints. If *bkpt_nos* are provided, only those breakpoints are reactivated. If **all** is specified, all previously deactivated breakpoints/tracepoints are reactivated.

**deact** [<*debug_context*> ] (*bkpt_no* {, *bkpt_n o*} | all)

deactivates user breakpoints and tracepoints. If bkpt_nos are provided, only those breakpoints are deactivated. If all is specified, all breakpoints/tracepoints are deactivated.

**remove** [<*debug_context*> ] (*bkpt_no* {, *bkpt_ no*} | all)

removes user breakpoints and tracepoints. If *bkpt_nos* are provided, only those breakpoints are removed. If **all** is specified, all breakpoints/tracepoints are removed.

## DATA AND STATUS ACCESS COMMANDS

**assign** [<*debug_context*> ] *variable*
**assign** [<*debug_context*> ] *variable* [ = <*expr*>]

assigns a new value to the specified variable. The first form displays the current value of the variable and then prompts you for a new value. If the *variable* is a structured variable, e.g., an array, each component is modified in turn. Enter carriage return if you don't want to change its value, and use "**$**" to terminate this command.

The second form silently assigns the value of <*expr*> to the variable without displaying the old value. If the variable is an array, every element will be assigned the new value. Note that this command format can't be used for structured variable assignment. FORTRAN complex variables are considered structured.

When assigning to a FORTRAN logical variable, use integers 0 and 1 (0 = false, 1 = true).

**dimension**

displays the dimension of the cube in which *decon* is running.

**display** [<*debug_context*> ] *variable* {, *variable*}

displays the values of the specified variables. If *variable* is a structured variable, each component is displayed in turn. The numeric value of a variable is always in decimal format.

When displaying character variable and FORTRAN logical variable, the number in parentheses which follows the character indicates the character's hexical number.

The field reference operators "." and "→" are used to refer to structured and union variables in C, and common second equivalence variables in FORTRAN.
**CAUTION**: Because the FORTRAN compiler does not produce enough information in the symbol table, only the variable which requires more space in an equivalenced variable can be displayed. If they have the same size, only the variables listed second in the equivalence statement can be found.

**list** [<*debug_context*>] [<*routine* | <*range*>]

displays the program listing in the specified debug context. If a routine name is specified, the first 20 statements of the whole routine are listed. <*range*> can be either "#stmt1,#stmt2" or "#stmt1 [,cnt]". The first format defines the range from stmt1 to stmt2. The second format defines **cnt** lines above and below stmt1. The default is 10 if **cnt** is not specified. Invoking list with no arguments will print 10 lines with the next executable statement.

## DATA AND STATUS ACCESS COMMANDS (continued)

**file** [<*file_id*>]

sets or displays the current source file. If <*file_id*> is provided, it becomes the current source file and succeeding list commands will use this file to display source statements. If <*file_id*> is not provided, the name of current source file is displayed.

**frame** [<*debug_context*>]

displays run-time stack activation chain in the specified debug context.

**source** [<*dir_str*>]

sets or displays the directory where the user's program source files are located. If <*dir_str*> is provided, it is set to be the new source directory. Otherwise, the name of the current directory is displayed. <dir_str> does not have to be quoted.

**scope** [<*debug_context*>]

displays current program scope in the specified debug context

**msgq** [<*debug_context*>] [*type_no*]

displays the messages currently queued in the system. If *type_no* is provided, only those messages of the specified type are displayed. Each message in the message queue is displayed as follows:

For From Type Size

The <for> field identifies the destination (nid:pid) context. The <from> field shows the context that sends the message. The <type> <size> indicate the message type and size. If there is no message, the message "MESSAGE QUEUE EMPTY" will be displayed.

## DATA AND STATUS ACCESS COMMANDS (continued)

**recvq** [<*debug_context*>]

displays user processes currently waiting for messages. Each entry of the display looks like:

waiting pid     message type

shows the message type the process is waiting for. The message "WAIT QUEUE EMPTY" is displayed if there is no process waiting for messages.

**status**

displays the status of node programs. Because *decon* regains control as soon as the first process (on node or host) hits a breakpoint or terminates, it saves other incoming messages sent by other processes in a message buffer. This command can be used to display messages in this buffer which contain the status of other processes.

**type** [<*debug_context*> ] *variable* {, *variable*}

displays the types of the specified variables. Note that the type "Complex" implies complex*8 and the type "Dcomplex" implies complex*16 in FORTRAN.

## DEBUGGER CONTROL COMMANDS

**alias** [<*new_cmd*> <*old_str*>]

defines or lists aliases. <*new_cmd*> is the name of the new command. <*old_str*> is the command name and its arguments which is to be aliased. It must be surrounded by single quotes. For example, the command "alias l 'load -p 1'" enables *decon* to recognize 'l' as a command to load a program with process id 1 into the nodes.

**context** [<*debug_context*>] {, <*debug_context*>}

sets or displays the current debug context. Invoking this command with no parameter displays the current context. The current context initially is set to empty, thus any *decon* command that requires a context will cause an error if invoked before the context is set.

**exec** [step] <*cmd_file*>

executes a debug command script. Normally the control will not return to *decon* until execution of the command script is completed. If the **step** option is specified, each command in the script <*cmd_file*> is echoed and waits for a carriage return <CR> before it is executed. Note that <*cmd_file*> must exist and contain legal debug commands. Note also that the command file cannot be nested and the maximum line length for a command in an exec file is 130 characters.

**help**

lists all *decon* commands.

**log** [on <*file*> | off]

turns on or off the logging of a debug session. If **on** is specified, debug information is written into the specified file. If **off** is specified, the logging is turned off. Invoking this command with no argument shows the name of the log file currently logged to. Note that if the specified file <*file*> exists, it will be overwritten.

**process**

prints out user processes currently controlled by *decon*. Displayed information includes the process status, filename, and its context.

## DEBUGGER CONTROL COMMANDS (continued)

### quit

terminates a debug session and exits *decon*.

### set [<*debug_var*> <*old_str*>]

defines or lists debug variables that can be used to replace long names. *decon* recognizes a debug variable when a variable is prefixed with a dollar sign. <*debug_var*> is the name of the newly defined debug variable. <*old_str*> is the variable name being aliased and must be surrounded by single quotes.

### system <*unix_cmd*>

passes <*unix_cmd*> to UNIX for services. The <*unix_cmd*> is not interpreted by *decon* and must be surrounded by single quotes.

### unalias (*new_cmd* {, *new_cmd*} | all)

undefines aliases. This command undefines either the specified <*new_cmd*> or all aliases.

### unset (*debug_var* {, *debug_var*} | all)

undefines debug variables. This command undefines either the specified *debug_var* or all debug variables.

# CHAPTER 4

# SAMPLE SESSIONS

## INTRODUCTION

This chapter presents two sample debugging sessions.

The first example is a relatively simple session used to illustrate the syntax of some of the debug commands and how they work.

The second example is a more detailed session based on an application using a ring topology.

## SAMPLE SESSION #1

Assume that the source program **test.c** is under directory /usr/abc/src, the executable file **test** is under directory /usr/abc/bin, and we are currently in directory /usr/abc/bin. The following simple session is given to illustrate the syntax of some debug commands and how they work. *Decon* outputs are not shown.

| Command | Comment |
| --- | --- |
| **cdbg> hload 0 foo** | -- load foo on host, given pid 0 (default) |
| **cdbg> context (host:0)** | -- set current environment to pid 0 on host |
| **(host:0) list** | --try to display the source program |
| **(host:0) system 'pwd'** | -- see where we are |
| **(host:0) source '../src'** | -- specify where the source programs are located |
| **(host:0) list** | -- then display the source again |
| **(host:0) break #14** | -- set a breakpoint at statement **14** |
| **(host:0) run** | -- start execution |
| **(host:0) display p->i** | -- see value of field i in object that p points to |
| **(host:0) run** | -- continue |
| **(host:0) display p->i** | -- see the value of p->i again |
| **(host:0) break a()** | -- set a breakpoint at routine **a** |
| **(host:0) break** | -- see where breakpoints have been set |
| **(host:0) remove 1** | -- remove first breakpoint, i.e. at statement **14** |
| **(host:0) run** | -- continue, should stop at routine **a** |
| **(host:0) step** | -- do a single step in routine **a** |
| **(host:0) frame** | -- see what routines are currently active |
| **(host:0) display i** | -- see the value of i in routine **a** |
| **(host:0) display main( ) i** | -- see the value of i in main |
| **(host:0) remove 2** | -- remove second breakpoint, i.e. at routine **a** |
| **(host:0) run** | -- finish execution |
| **(host:0) quit** | -- quit the debugger |

## SAMPLE SESSION #2

Imagine an application with a ring topology. Each node in the ring receives messages from its "left" neighbor and sends messages to its" right" neighbor. A host process initially sends a message into the ring via node 0 and then waits for a message that is sent from node 0 when the circuit is completed. Refer to Figure 4-1 below:

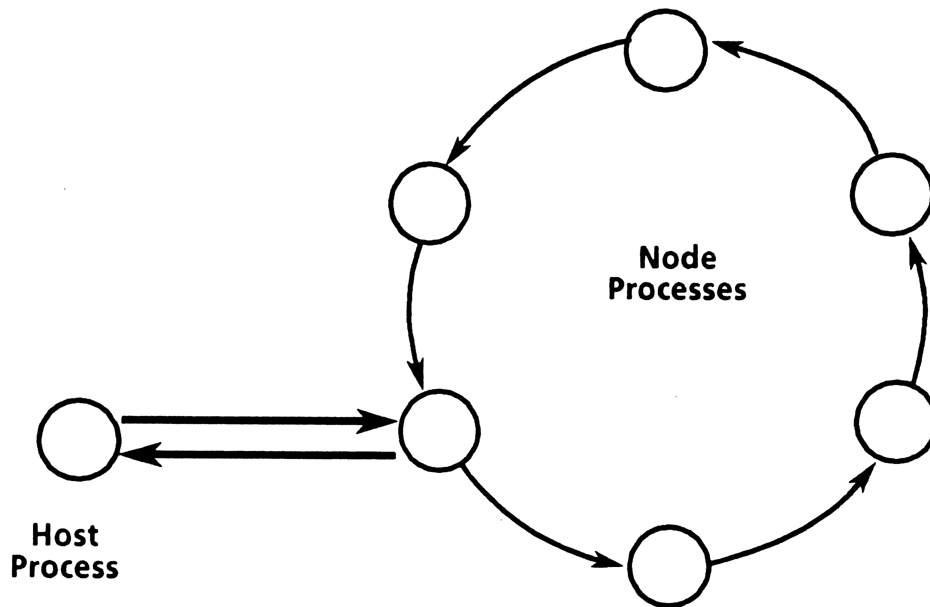


**Figure 4-1**
**Application With Ring Topology**

Relevant parts of the application source code are listed below, (in C):

| Statement | Host Source Code | Comments |
|---|---|---|
| ... | ... | |
| 34 | csend (starttype, ...); | Send "starttype" message into ring |
| 35 | crecv(endtype, ...); | Receive an "endtype" message |
| ... | ... | |

| Statement | Node Source Code | Comments |
|---|---|---|
| ... | ... | |
| 44 | if (mynode() = = 0) { | If this executes on node 0 then: |
| 45 | crecv(starttype, ...); | Receive a "starttype" message (from host) |
| ... | ... | ... and |
| 55 | csend(ringtype, ...); | Send a "ringtype" message (to "right" node) |
| | } | |
| | else { | Execute this code on all other nodes: |
| 56 | crecv(ringtype, ...); | Receive a "ringtype" message (from "left" node) |
| ... | ... | ... and |
| 66 | csend(ringtype,...); | Send a "ringtype" message (to "right" node) |
| | } | |
| ... | ... | ... |

The following debug session illustrates the flexibility and ease of use of DECON. Note: **bold** characters indicate input from the user.

| SESSION | COMMENTS |
|---|---|
| % **decon** | Start DECON from UNIX |
| decon> **load -p 1 node** | Load the 'node' program and assign it pid = 1. |
| decon: loading node program | By default program is loaded into all nodes |
| decon: program loaded | allocated to user. |
| decon> **context (nodes:1)** | Set the context to be process 1 executing on all nodes. |
| (nodes: 1)> **break #44** | Set breakpoints at statement 44 in process 1 on all nodes. |
| (nodes: 1)> **run** | Begin executing process 1 on all nodes. |
| (0:1) : stopped at main() #44 (bpt# 1) | Node 0, process 1 reports hitting a breakpoint. |
| (nodes: 1)> **status** | Request status of nodes. |
| (1:1) : stopped at main() # 44 (bpt# 1) | Nodes report hitting breakpoint number 1. |
| (2:1) : stopped at main() #44 (bpt# 1) | |
| (3:1) : stopped at main() # 44 (bpt# 1) | |
| (nodes: 1) > **msgq** | List messages queued up at nodes for process 1 |

| For | From | Type | Size |
|---|---|---|---|
| (0:1) | Message Queue Empty | | |
| (1:1) | Message Queue Empty | | |
| (2:1) | Message Queue Empty | | |
| (3:1) | Message Queue Empty | | |

(nodes: 1)> **recvq** — List outstanding requests to receive messages from process 1 on all nodes.

| Node | Waiting Pid | Message Type |
|---|---|---|
| 0 | Wait Queue Empty | |
| 1 | Wait Queue Empty | |
| 2 | Wait Queue Empty | |
| 3 | Wait Queue Empty | |

Processes with pid = 1 on nodes have no outstanding requests to receive messages. (They have not yet been issued)

| SESSION | COMMENTS |
|---|---|
| (nodes: 1)> **hload 1 host** | Load the host program with pid = 1 |
| decon : loading host program | |
| decon : program loaded | |
| (nodes: 1)> **context (host:1)** | Set context to be process 1 on the host |
| (host: 1)> **break #35** | Set breakpoint at statement 35 on process 1 on host |

| Session | Explanation |
|---|---|
| (host: 1)> **run** | Start the host process |
| (host: 1) : stopped at main() #35 | Host process reports hitting breakpoint at statement 35 |
| (host: 1)> **context (nodes:1)** | Set the context back to the nodes |
| (nodes: 1)> **msgq** | Request information on nodes message queues |
| **For** **From** **Type** **Size** | |
| (0:1) (host: 1) 10 4 | Node 0 has queued a message from host |
| (1:1) Message Queue Empty | Other nodes have no queued messages |
| (2:1) Message Queue Empty | |
| (3:1) Message Queue Empty | |
| (nodes: 1)> **display (host:1) starttype** | Display the value of the variable "starttype" in |
| (host: 1) startttype = 10 | host process. |
| (nodes: 1)> **run (0:1)** | Run process 1 on node 0 |
| (nodes: 1)> **msgq** | Request information on nodes message queues |
| **For** **From** **Type** **Size** | |
| (0:1) Message Queue Empty | |
| (1:1) (0:1) 20 8 | There is a message for process 1 on node 1 |
| (2:1) Message Queue Empty | |
| (3:1) Message Queue Empty | |
| (nodes: 1)> **display ringtype** | Display the value of the variable "ringtype" in |
| (0:1) ringtype = 20 | process 1 on nodes |
| (1:1) ringtype = 20 | |
| (2:1) ringtype = 20 | |
| (3:1) ringtype = 20 | |
| (nodes: 1)> **remove 1** | Remove breakpoint #1 (previously set on nodes) |
| (nodes: 1)> **run** | Run process 1 on nodes (previously stopped) |
| (nodes: 1)> **msgq(host:1)** | Show messages queued up for host process |
| **For** **From** **Type** **Size** | |
| (host: 1) (0:1) 30 8 | Host has a type 30 message from node 0 (indicates ring circuit has been completed) |
| (nodes: 1)> **context (host:1)** | Change context back to host process |
| (host: 1)> **run** | Execute host process (it was at breakpoint) |

| Command / Output | Comment |
|---|---|
| (host:1)> **msgq** | Again, list messages queued for host |

| For | From | Type | Size | |
|---|---|---|---|---|
| (host:1) | (0:1) | 30 | 8 | Previous message was not consumed by host. |

| Command / Output | Comment |
|---|---|
| (host:1)> **recvq** | List outstanding requests to receive messages |

| Node | Waiting Pid | Message Type | |
|---|---|---|---|
| Host | 1 | 20 | Host process (1) is waiting for a type 20 message; however, the queued message is of type 30 ! |

| Command / Output | Comment |
|---|---|
| (host:1)> **display endtype** | Confirm suspected source of error |
| (host:1) endtype = 20 | Suspicion confirmed; endtype should be 30, not 20 |
| (host:1)> **quit** | Exit DECON and return to UNIX (to fix error). |