CONVEX VMEbus HSP/HIA Subsystem (*io4120*) Diagnostics Manual

Document No. 760-003630-000

First Edition
May 1991

CONVEX Computer Corporation
Richardson, Texas USA

CONVEX VMEbus HSP/HIA Subsystem
(io4120) Diagnostics Manual
Order No. DHW-248
First Edition

Printed in the United States of America

Revision Sheet

CONVEX VMEbus HSP/HIA Subsystem (io4120) Diagnostics Manual

Edition	Document No.	Date	Description
First	760-003630-000	May 1991	First release. Contains the *io4120* diagnostic test information from the *CONVEX PBUS I/O Systems Diagnostics Manual.*

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

Appendixes

A Reporting Problems

List of Tables

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

Preface

Purpose and Intended Audience

This manual explains how to run the *io4120* diagnostic, which verifies the operation of a High-Speed Parallel (HSP) Channel Control Unit (CCU) and HSP Interface Adapter (HIA). This document is not a tutorial, but rather a reference for the users of the *io4120* diagnostics, including field service and manufacturing test personnel, as well as the diagnostics sustaining staff. In addition, CONVEX customers can use this manual to execute the *io4120* diagnostic.

Scope

This manual applies to all CONVEX computers.

Organization

This document consists of the following:

- **Chapter 1. Diagnostics Environment**—Introduces the theories and concepts that underlie I/O diagnostics on CONVEX machines as well as the basic overview, philosophy, and structure of I/O diagnostics.
- **Chapter 2. EGOS Overview**—Provides a brief overview of the Event Governed Operating System (EGOS) and how it relates to device and peripheral diagnostics testing.
- **Chapter 3. Dshell Overview**—Provides a brief overview of and a general introduction to the *dshell* utility.
- **Chapter 4. HSP/HIA Subsystem Test (*io4120*)**—Describes how to operate the diagnostic, including prerequisites, test invocation, hardware initialization sequence, and class descriptions.
- **Appendix A. Reporting Problems**—Provides an example of the CONVEX ***contact*** utility for reporting minor software and hardware problems.

Notational Conventions

The notational conventions used in this text are listed below:

- Bit numbering is left to right, `N-1` through `0`. The most significant numerical bit is `N-1`, the least significant `0`. The bit numbering represents the binary weight of a position.
- Bit fields are specified using the following convention: *name*<*x..y*> where the bit field is *name* from bits *x* through *y*.
- Individual bit positions within a register are denoted by specific positions separated by commas. For example, REG<15,4,0> denotes bits 15, 4, and 0 of REG.
- Byte numbering is from left to right
- A *bit* is a single binary value or entity
- A *nibble* is 4 bits
- A *byte* is 8 bits
- A *halfword* is 16 bits
- A *word* is 32 bits
- A *longword* is 64 bits
- *Single precision* is a 32-bit floating point word
- *Double precision* is a 64-bit floating point longword
- An *instruction* is a multihalfword operand
- A bit is *set* when it contains a binary value of `1`.
- A bit is *clear* when it contains a binary value of `0`.
- All memory and I/O addresses are written in hexadecimal notation unless explicitly stated otherwise.
- All register contents are written in hexadecimal notation unless explicitly stated otherwise.
- A *register* is a programmer-visible hardware storage element internal to the processor
- *Physical memory* is the physical storage installed in the processor
- *Virtual memory* is the perceived amount of physical memory as seen by the application programmer
- The symbol *K* is an abbreviation for *kilo* or 1,024
- The symbol *M* is an abbreviation for *mega* or 1,048,576
- The symbol *G* is an abbreviation for *giga* or 1,073,741,824
- A *stack* is a linked-list group of words useful for dynamic allocation and deallocation of memory
- A *return block* is a collection of registers that is pushed or popped from a context stack in response to an instruction or other event
- *Reserved* or *undefined* convey what to expect, if anything, from unused fields in registers, reserved memory, or reserved I/O space. Algorithm implementation based on the use of undefined or reserved fields is not recommended.

Warnings

The following are examples of warnings, cautions, and notes and their typical content as used in CONVEX documents:

WARNING

Warnings highlight procedures or information necessary to avoid injury to personnel. A warning immediately precedes the critical information and includes a description of the hazard.

CAUTION

Cautions highlight procedures or information necessary to avoid damage to equipment, loss of data, or invalid test results. A caution immediately precedes the critical information and includes a description of the possible damage.

NOTE

Notes highlight useful information that is supplemental in nature. A note may immediately precede or follow the information that is being highlighted.

Associated Documents

The following is a partial list of other manuals or books that may provide more detailed information on the topics presented in this manual:

- *CONVEX Processor Diagnostics Manual (C1, C120)*, Order No. DHW-071
- *CONVEX Processor Diagnostics Manual (C200 Series)*, Order No. DHW-081
- *CONVEX Architecture Reference*, Order No. DHW-005
- *CONVEX SPU UNIX Utilities Manual*, Order No. DHW-021
- *CONVEX Processor Operation Guide (C100 Series, C200 Series)*, Order No. DHW-015
- *CONVEX Diagnostic Utilities Manual (C1, C120)*, Order No. DHW-072
- *CONVEX Diagnostic Utilities Manual (C200 Series)*, Order No. DHW-082
- *CONVEX UNIX Tutorial Papers*, Order No. DSW-002
- *The C Programming Language*, Kernighan & Ritchie, Order No. DSW-046

Ordering Documentation

To order the most current version of this or any other CONVEX document, use the product number. If the product number is not known, order by the exact title. In some situations, the most current version may not be desired. To receive a specific version of a manual, order the manual by its document number, or part number, which can be obtained by contacting the local CONVEX office or by calling the Technical Assistance Center.

The product number for this manual is DHW-248.
The document number for this manual is 760-003630-000.

CONVEX documents can be ordered by mail by sending a request to:

CONVEX Computer Corporation
Customer Service
PO Box 833851
Richardson TX 75083-3851 USA

Technical Assistance

Hardware and software support can be obtained through the CONVEX Technical Assistance Center (TAC):

- From all locations in the continental United States, call 1(800)952-0379.
- From locations in Alaska, Hawaii, and Canada, call 1(214)497-4379.
- From all other locations, contact the nearest CONVEX office.

Reader's Forum

If you wish to mail your comments to us, please use the form at the end of this manual and list the document page number with your questions and comments. Thank you.

Chapter 1
Diagnostics Environment

1.1 Overview

CONVEX system diagnostics consist of a suite of test programs designed (except where noted) to execute under the Service Processor operating system, SPU UNIX. These programs utilize the capabilities of the Service Processor to test the operation of one or more of the functions of the system and report any errors detected. All of the diagnostics in this manual are intended to be executed "off-line"; that is, while CONVEX UNIX is not being executed by any of the Central Processing Units (CPUs) in the system.

The Service Processor, together with SPU UNIX, various diagnostic utilities, and the test programs, themselves, comprise the CONVEX diagnostic environment. This chapter describes the hardware and software components of this environment and is intended to provide the background necessary to fully utilize the capabilities of the CONVEX processor diagnostics.

For more information about the diagnostic environment refer to the Diagnostic Environment chapter in the *CONVEX Processor Diagnostics Manual (C200 Series)* or the *CONVEX Processor Diagnostics Manual (C1, C120)* depending on the architecture of the machine under test.

1.2 Test Program Naming Conventions

Test program names are in the form *cattypedevnn.suffix* where:

- *cat* is the subsystem being tested
- *type* is the type of test being performed, e.g., standalone, self-test, or offline functional test
- *dev* is the device being tested, e.g., disk, tape, or printer. This segment of the test program name is used *only* if the category is a device.
- *nn* is a CONVEX code used for distinguishing between test programs
- *suffix* is one of three program identifiers:
  - *.t* are programs that execute on SP2
  - *.x00* and *.rnn* are object files for different target processors other than the SP2. The target processor depends on the subject of the test. The test program name must have the test program category (*cat*) at the beginning of the name to determine the target processor.

1.2.1 Test Program Categories

Test program categories include those tests for the CPU, peripheral devices, I/O system, memory system, SP2, and entire system. For example, *cpu4041* is a CPU vector instruction test while *mem4000* is a memory system functional test. The following table lists test program categories:

Table 1–1, Test Program Categories

TEST PROGRAM CATEGORIES	
Test Category (*cat*)	**Description**
cpu	CPU subsystem related test
dev	Peripheral device test
io, idc, tli	I/O subsystem related test
mem	Memory subsystem related test
spu	SP2 subsystem related test

1.2.2 Test Program Types

A test program type describes whether a test is a standalone test, self-test, kernel hardware test, or an offline or online functional test. See the following table for the numbering system and description of test program types:

Table 1–2, Test Program Types

TEST PROGRAM TYPES	
Number (*type*)	**Description**
0	Standalone test
1	Self-test
2	Kernel hardware test
4, 5	Offline functional test

1.2.3 Test Program Device Types

Test programs will test disks, tapes, terminals, printers, and networks. See the following table for the numbering scheme and a description of the test program device types:

Table 1–3, Test Program Device Types

TEST PROGRAM DEVICE TYPES	
Number (*dev*)	**Description**
1	Disk
2	Tape
3	Terminal
4	Printer
5	Network

1.2.4 Examples of Test Program Names

The following table presents some examples using the naming conventions outlined above:

> **NOTE**
>
> In the following table, SOFF stands for Standard Object File Format.

Table 1–4, Example Test Program Names

EXAMPLE TEST PROGRAM NAMES	
Test Program Name	**Description**
cpu4041.t	SP2 object code in *b.out* format for *cpu4041*
cpu4041.rnn	C210 or C220 machine object code in SOFF format (relocatable)
cpu4041.x00	C210 or C220 machine object code in SOFF format (linked to run in segment 0)
mem4000.t	SP2 object code in *b.out* format for *mem4000*
mem4000.x00	C210 or C220 machine object code in SOFF format (linked to run in segment 0)
dev4100.t	SP2 object code in *b.out* format for *dev4100*
dev4100.x00	IOP object code in *b.out* format

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 2
EGOS Overview

2.1 Overview

This chapter provides an overview of the Event Governed Operating System (EGOS) and how it relates to device and peripheral diagnostics testing. There are three basic types of EGOS systems, one for each type of CCU. There is one for the Multibus interface, one for the VME interface, and one for the HIA interface. This chapter will explain the three types of EGOS systems and how EGOS is positioned within the overall operating system environment.

2.2 Purpose of EGOS for Diagnostic Testing

EGOS is basically a simple operating system that the device tests use to handle interrupts, schedule processes, and generally allocate and control IOP/VIOP resources. The diagnostics code uses both EGOS and the Message Based System (MBS) to manipulate test program control over to the CCU side of the test program. MBS is not a part of EGOS but rather a system that allows a common section of memory to be used as a message area between multiple processors. For more information on MBS, refer to the *CONVEX Guide to Writing Device Drivers.*

EGOS initially sets up interrupt tables, determines how many chassis there are, and initializes its windows and resource allocation tables.

2.3 EGOS for the Multibus Interface

EGOS for the Multibus interface supports event driven device drivers. The Multibus version of EGOS takes interrupts that are local to a CCU and channels those errors to the proper piece of code to handle the error. It basically supplies the error interrupt handlers for the CCU error interrupts. It also contains support routines to control allocation of the various CCU-related resources.

2.4 EGOS for HSP Interface, HSP EGOS

EGOS for the HSP interface supports event driven device drivers. The HSP version of EGOS is like the Multibus version. It takes interrupts that are local to a CCU and channels those errors to the proper piece of code to handle the error. It basically supplies the error interrupt handlers for the CCU error interrupts. It also contains support routines to control allocation of the various CCU-related resources.

2.5 EGOS for VME Interface, VIOP EGOS

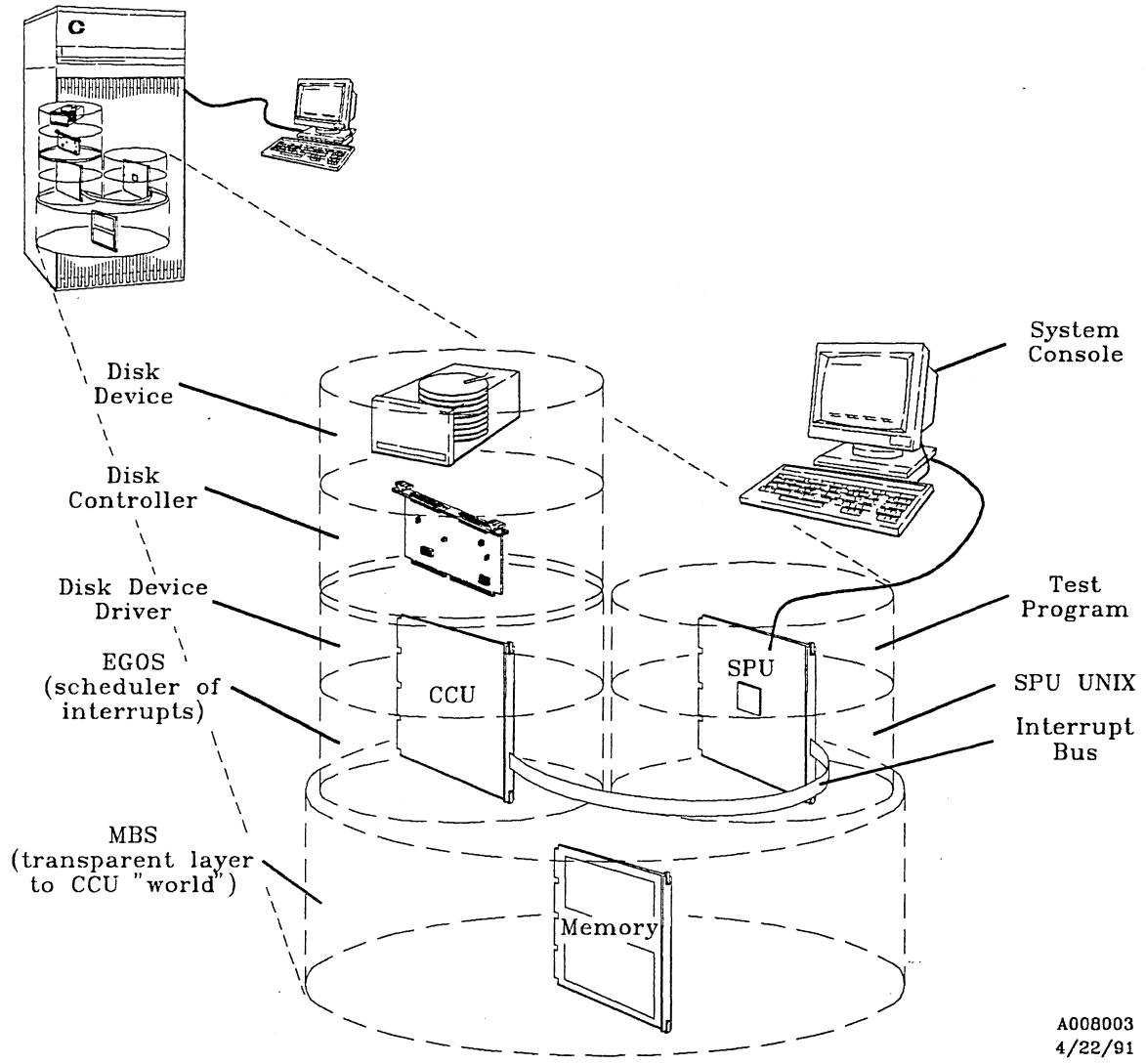
The VME interface version of EGOS is designed with a scheduler for the VIOP and is called VIOP EGOS. VIOP EGOS supports event driven device drivers as well as process type device drivers. VIOP EGOS utilizes a *sleep/wakeup* type of process control that improves efficiency of the device driver and makes it less complicated to create user written device drivers. Each process device driver has a priority level that can be defined relative to other processes. The scheduler supports 32 process priorities and is preemptive for higher priority processes. The VIOP hardware supports 14 device events for event driven device drivers. The 14 levels actually share 2 68020 interrupt levels. Therefore, two is the maximum number of processes at any given time.

2.6 EGOS Position in the Environment

EGOS is positioned in the operating environment between the actual device driver and MBS. MBS is a transparent layer that bridges the CCU and its resources to SPU UNIX. SPU UNIX handles many of the message manipulations that occur during testing. Many error messages that occur during diagnostics testing come from the device driver. When the device driver detects an error from the controller, it calls a routine in EGOS that places a message in the MBS system. This causes SPU UNIX to be interrupted and it retrieves the message from MBS. SPU UNIX then passes a signal to the test program. The test program then prints an error message to the console based on the code that it received.

The following figure illustrates the position of EGOS in the operating system environment.

Figure 2–1, EGOS' Position in the Environment



THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 3
Dshell Overview

3.1 Overview

This chapter provides a brief overview of the *dshell* utility. Included in this overview is an overall explanation of the utility and a list of the utility's commands. For a complete description of this utility, refer to the Dshell chapter of the *CONVEX Diagnostic Utilities Manual (C200 Series)* or the *CONVEX Diagnostic Utilities Manual (C1, C120)* depending on the architecture of the machine under test.

3.2 Diagnostic Shell (*dshell*) Overview

The Diagnostic Shell (*dshell*) is a command interface program that runs on the Service Processor. Most of the diagnostics available for the CONVEX machines are interfaced through the *dshell*. Certain peripheral diagnostics are run as standalone tests. To determine whether a test can be run under the *dshell*, consult the appropriate chapter in this manual.

The *dshell* has two basic functions:

- Selecting diagnostics for execution
- Selecting test options
  - Pause on a failure or at the beginning or end of any specific subtest
  - Loop on a specific type of subtest or on a given set of subtests
  - Select subtest execution order
  - Direct test output to a file or to the screen (or both) to monitor the test as it runs or to analyze test results later
  - Select long or short error messages, or turn messages off
  - Execute either user-created or predefined command scripts

The following table list the various *dshell* commands and their functions.

Table 3–1, *dshell* Commands

COMMAND	FUNCTION
! [command]	This command is used to access, or *fork* a UNIX shell to execute the command that follows *!*.
exit	The *exit* command causes immediate termination of the *dshell* process and any test processes that may have been forked.
quit	The *quit* command causes immediate termination of the *dshell* process and any test processes that may have been forked.
^C	Returns user to the *dshell* command level if no subtest is running.
^B	Immediately terminate the *dshell* and any associated active processes. Core is dumped.
help	The *help* command causes a standard *help* menu to be displayed. The menu describes the correct command syntax for each *dshell* command and gives a terse description of what each command does.
status	The *status* command generates a report on the current state of the *dshell* command options. This report gives the name of each flag, its current value, and an explanation of its current effect.
log [options]	The *log* command provides a mechanism for specifying the number of failures that will be allowed to occur before a test or subtest terminates execution.
loop [options]	The *loop* command causes the *dshell* to repeat the execution of a test or subtest.
msgs [options]	The *msgs* command enables or disables different levels of test, class, and subtest result messages.
pause [options]	The *pause* command returns program control to the *dshell* to the beginning, end, or failure of all or specific subtests.
test [options]	The *test* executes specific tests, and displays test, class, and subtest menus.

3.3 Syntax Help for *dshell* Commands

The syntax for each *dshell* command can be obtained by typing the command with no options and pressing <**CR**>. For example, by entering **loop** and pressing <**CR**>, the syntax help in the following figure will be displayed on the screen:

Figure 3–1, Syntax Help for the *loop* Command

```
: loop
Proper syntax is:

loop off (-s) (-t)                          :disables loop modes
loop -s nnn                                 :loop on subtest nnn
loop -t                                     :loop on test
```

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4
HSP/HIA Subsystem Test (*io4120*)

4.1 Overview

The *io4120* test verifies the operation of a High-Speed Parallel (HSP) Channel Control Unit (CCU) and HSP Interface Adapter (HIA).

The *io4120* test verifies the following local to the HSP:

- 68000 instruction execution
- HSP local memory operations
- HSP local memory protection
- Input bus integrity
- PBUS header generation
- PBUS access verification
- PBUS interrupt operations
- Correct HSP Address Translation Unit (ATU) operations in both physical and virtual address modes
- ATU error detection
- Output bus integrity
- Error status register verification

Additionally, the *io4120* test checks these functions that require the HIA:

- Register accesses
- Interface interrupt test
- Interface parity generation or detection
- Synchronous data transfers in both physical and logical address modes
- Bus arbitration
- Clock selection
- Channel interleave
- Error detection and recovery

4.2 Prerequisites and Required Equipment

The following table lists the required hardware depending on the type of machine under test:

Table 4–1, Hardware Requirements

C1, C120	C200 Series
MCU	Memory System[1]
MAU	CPX
SPU	SP2
HIA	HIA
HSP	HSP
FSE	FSE
	PIA

[1] Memory System consists of a minimum of one pair of memory boards (one odd and one even).

4.3 Test Invocation

The *io4120* test executes under the Diagnostic Shell (*dshell*) and supports all the features of the *dshell*. The *dshell* permits tests to be initiated in any order.

To invoke the *io4120* test, use the procedure shown in the following figure. All responses in **boldface** are entered by the user. The prompts and responses appear sequentially on the screen, one line at a time. All prompts and responses are shown in one figure for convenience.

Figure 4–1, Test Invocation Sequence

```
(spu)> cd /mnt/test RETURN
(spu)> sysreset RETURN
(spu)> mminit -s RETURN
(spu)> dshell RETURN
: test io4120 [-c [class number(s)]] [-s [subtest number(s)]] [+>filename] RETURN
```

NOTE

After entering **dshell**, specific *dshell* parameters may be changed. Refer to the "Dshell Overview" chapter of this manual for more information.

Entering only **test io4120** executes all *io4120* subtests sequentially. Execute a specific class(es) of subtest(s) or one or more individual subtests by using the **-c** or **-s** options, respectively. Detailed information for using these options can be found in the "Dshell Overview" chapter of this manual. The [+>*filename*] option allows the test results to be appended to *filename*.

The following alternate test invocation procedure may be required in some cases:

Figure 4–2, Alternate Test Invocation Sequence

```
(spu)> cd /mnt/test RETURN
(spu)> initall RETURN
(spu)> dshell RETURN
: test io4000 [-c [class number(s)]] [-s [subtest number(s)]] [+> filename] RETURN
```

CAUTION

The user response, **initall**, is typically required if the *initall* utility has not been run since the last power up. However, if any problems have occurred subsequent to the last time *initall* was run, (i.e., system crash, hard error, or failure of previous diagnostic), it should be run again. In this case, failure to run *initall* could result in invalid test results.

NOTE

The *initall* utility requires a significant amount of time (2 to 3 minutes depending on if the control stores have been previously loaded) to execute. If no system abnormalities have occurred subsequent to the last time the system was booted or *initall* was executed, it is not necessary to run *initall*.

4.3.1 Example Test Parameter Menu

Once the test is invoked, a test menu prompt is presented allowing selection of default switches. The following figure shows all prompts, their possible answers (in brackets [ ]), and their default answers (in parentheses ( )). The prompts and responses in the following figure appear sequentially on the screen, one line at a time. All the prompts and responses are shown in one figure for convenience.

Figure 4–3 illustrates *all* questions that can be displayed during test parameter input.

Figure 4–3, Example Test Parameter Menu

```
                              ENTER TEST PARAMETERS

                []      Encloses allowed input ranges or values
                ()      Encloses the default value
                ^       Returns to the previous prompt
                :nn     Returns to the prompt # nn
                :       Returns to the first unsatisfied prompt
                :?      Reviews previous entries

Hsp ccu slot number [0-3]                          (2) ->
Hsp default clock rate [0-3]                       (0) ->
Hia outclk default rate [0-3]                      (0) ->
Hia busclk default rate [0-1]                      (0) ->
Channel mask for compliance test - ls bit is chnl 0
[0x0-0xffff]                                  (0xffff) ->
Default power margin for HIA chassis (+5 only)
[l,n,u]                                            (n) ->
ATU scheme to test ('1' for C1, '2' for C2, 'b' for both)
[1,2,b]                                            (b) ->
Enter OK, or :NN to return to question NN [OK]
                                                  (OK) ->
```

However, some questions may be omitted, depending on answers to previous questions. In all cases, questions are numbered sequentially. However, the numbers displayed on the screen during testing may not correspond to those shown in the example "Example Test Parameter Menu," as the questions illustrated are examples only.

At any time during the test parameter sequence, several options are available as denoted at the top of the Test Parameter Menu. The following list summarizes the available options:

:nn — Returns to an earlier prompt (n is the prompt number)

: — Advances to the next unanswered prompt

:? — Displays (reviews) all responses up to the current prompt

? — Requests help for the current prompt (if available)

^ — Returns to the previous prompt

4.3.2 Prompt Explanations

Each prompt is repeated and explained in the following paragraphs.

> **NOTE**
>
> The CCU slot specified is checked via the diagnostic bus to ensure that the HSP is present. Any invalid entry reprompts for the HSP slot number.

```
Hsp ccu slot number [0-3]                    (2) ->
```

Enter the CCU slot number the HSP resides in.

```
Hsp default clock rate [0-3]                 (0) ->
```

Enter the HSP default clock rate which sets the transfer rate from the HSP to the HIA. Enter **0** to select the highest rate.

```
Hia outclk default rate [0-3]                (0) ->
```

Enter the HIA outclock default rate which sets the transfer rate from the HIA to the HSP. Enter **0** to select the highest rate.

```
Hia busclk default rate [0-1]                (0) ->
```

Enter **0** or RETURN to select the default 40-MHz crystal. This crystal produces a 10-MHz transfer rate. Enter **1** to select a user defined crystal. A user-defined crystal, for example 1 36-MHz crystal producing a 9-MHz transfer rate, would be selected.

```
Channel mask for compliance test - ls bit is chnl 0
[0x0-0xffff]                              (0xffff) ->
```

Enter **0xffff** to select an HIA. See the description of Subtest 4999, *CONVEX Register Compliance Test* in this chapter for more information.

```
Default power margin for HIA chassis (+5 only)
[l,n,u]                                      (n) ->
```

Enter the power margin for the HIA chassis. The power margin may be changed to help debug hardware problems by forcing intermittent errors to hard errors.

- Enter **l** to select the lower power margin, 4.75 V.
- Enter **n** to select the nominal power margin, 5.00 V.
- Enter **u** to select the upper power margin, 5.25 V.

```
ATU scheme to test ('1' for C1, '2' for C2, 'b' for both)
[1,2,b]                                          (b) ->
```

Enter **1** if the test is to execute on a C1 or C120 machine. Enter **2** if the test is to execute on a C200 Series machine. Enter **b** for both ATU schemes to test on a machine.

> **NOTE**
>
> If **b** is entered, some subtests will take twice as long to execute.

```
Enter OK, or :NN to return to question NN [OK]
                                                 (OK) ->
```

If **OK** or RETURN is entered, the "Example Test Parameter Menu" terminates and all inputs are no longer changeable.

When all prompts have been answered, the screen displays a test parameter summary that echos the prompts that have been answered. The following figure illustrates an example of a "Test Parameter Summary" screen. The actual values and responses vary according to the input.

Figure 4–4, Sample Test Parameter Summary

```
                TEST PARAMETER SUMMARY

Hsp ccu slot number                          : 2
Hsp default clock rate                       : 0
Hia outclk default rate                      : 0
Hia busclk default rate                      : 0
Channel mask for compliance test - ls bit is chnl 0
                                             : 0xffff
Default power margin for HIA chassis (+5 only)
                                             : n
ATU scheme to test ('1' for C1, '2' for C2, 'b' for both)
                                             : b
Enter OK, or :NN to return to question NN : OK
```

4.4 Hardware Initialization Sequence

After the last prompt is entered, and before test code execution, the following events occur:

- Calls a routine to read Population Configuration Map (PCM) to determine where main memory is
- Allocates SPU windows for use in test
- Resets system through Scan Interface
- Turns on clocks for HSP
- Sets up software to be aware of hard and soft errors
- SPU local test variables are initialized
- Displays a "Test Parameter Summary"

After all the above events have occurred, the test code is started.

4.5 Class Descriptions

The *io4120* test has four classes of subtests as shown in the following table:

Table 4–2, *io4120* Test Classes

CLASS	DESCRIPTION
1	68000 subsystem tests
2	HSP standalone tests
3	for use by CONVEX manufacturing only
4	HSP/HIA tests
5	HSP/HIA/FSE tests

4.5.1 Class 1 Subtests

Class 1 subtests verify that the HSP can correctly load a program from main memory. Class 1 subtests reside in the HSP EPROM and communicate with the SPU via the LED and the auxiliary test result registers on the HSP scan ring.

NOTE

The following are restrictions on the execution sequence of these subtests:

- Subtest 100 must be the first subtest in this class executed
- Subtests 101 and 102 may be executed in any order and may be executed in a loop until a failure occurs. If a failure occurs, Subtest 100 must be re-executed.
- After Subtest 103 is executed, Subtest 100 must be re-executed.

It is recommended that these tests be executed in numerical order.

The Class 1 subtests are listed in the following table:

Table 4–3, Class 1 Subtests

SUBTEST	DESCRIPTION	TIME (min:sec)
100	HSP Reset	<:02
101	HSP Self-test	01:20
102	HSP Memory Initialization	<:02
103	HSP Boot	<:02

4.5.1.1 Subtest 100, HSP Reset

Subtest 100 resets the HSP via the scan ring. This causes the HSP to execute the initialization code in its EPROM. The initialization code saves the state of the HSP as found on reset and verifies that the HSP 68000 can calculate a checksum. When verified, the HSP calculates a checksum of the EPROM. If these tests pass, the HSP enters a loop and waits for an EPROM command to be placed on the LED register. If this test fails, the HSP informs the SPU by asserting a hard error (a double bus fault) which causes the 68000 to stop executing instructions.

4.5.1.2 Subtest 101, HSP Self-Test

Subtest 101 is the HSP self-test and consists of 10 steps that verify the ability of the HSP to load a program from main memory. The SPU commands the HSP to perform this test by placing a code in the HSP LED registers via the scan ring. When this code is detected, the HSP performs the steps listed below. If a failure is detected, the HSP returns the step number and other relevant data via the LED and auxiliary test result register.

Step 1: Verify Instructions Needed for Memory Test

Step 1 performs the following:

- Tests HSP 68000 registers
- Checks addressing modes
- Verifies jump and branch instructions
- The following operations are executed:
  - Integer arithmetic tests
  - Data movement operations
  - Shift and rotate instructions
  - Bit manipulation

Step 2: Initial Memory Operations Test

Step 2 determines the size of memory by writing to the first word in each bank until a bus error occurs. A walking '1' and '0' test is performed on the first word after the ROM and on the first word of the following banks. Then a uniqueness test and a true/complement pattern test is performed on the last 4 Kbytes of installed memory. Memory parity is checked by writing a word with inverted parity and then reading it to generate a parity error. The word is then rewritten with the correct parity.

Step 3: Verify Remaining 68000 Instructions

Step 3 uses the last 4 Kbytes of installed memory as a stack and checks the 68000 instructions not tested in Subtest 100 or in step 1 of this subtest. However, the *stop*, *trace*, and *reset* instructions are not checked. Step 3 also checks privilege violations and address errors.

Step 4: Test Remaining Memory

Step 4 is similar to step 2, but it checks memory from immediately after the ROM to the word preceding the last 4 Kbytes of installed memory. A parity test is not performed.

Step 5: Verify HSP Local Memory Protection

Step 5 performs a walking '1' and '0' test on the memory protection register at `0xff8000`. A uniqueness test and a true/complement pattern test is performed on all 256 registers. The test copies the HSP ROM into its memory, modifies the protection for installed memory to valid, read, write, and execute. Then the registers are cleared for uninstalled memory. Memory protection is turned on, and the valid, read, write, and execute bits are verified from the supervisor state and from the user state. At the end of the test, memory protection is disabled.

Step 6: Input Bus Test

Step 6 tests input bus integrity by pattern testing a longword in ATU memory. ATU memory is in two different address spaces in the HSP. The first address space checks the lower 32 bits of the input bus, and the second checks the upper 32 bits. The same 32 bit memory location is checked in both address spaces.

Step 7: PMAP Register Test

Step 7 tests the memory that stores the PMAP register data by performing column functionality, uniqueness, and bit functionality tests. Bad parity detection is then checked. Then a functionality test of the valid bit in the PMAP register is performed.

Step 8: Not Used

Step 8 is not used in the self-test in order to be able to distinguish a failure of a given self-test step from a HSP that has not started execution of its self-test. The value of 8 is used by the SPU to command the HSP to perform its self-test. As the HSP self-test executes, the value is incremented. By skipping step 8, the SPU can determine if the HSP started execution of its self-test.

Step 9: PBUS Header Generation

Step 9 checks the HSP PBUS header generation function by asserting the PBUS_TEST bit in the FDR register. Headers are generated for physical addresses of zero, all ones, walking zero, and walking one for both reads and writes in 8-bit and 16-bit access modes. When a header is generated, it is verified by inspection of the PBUS_HDR register. PBUS data is not transferred in during this test.

Step 10: PBUS Access Test

Step 10 performs a PBUS access test that reads, verifies, and echos a pattern placed in main memory by the SPU. This test verifies that the SPU and the HSP agree on the location of main memory and on the proper byte ordering in that memory. On C100 Series machines, the first access to the PBUS made by the HSP is to read the PCM. The PCM is used to locate the test pattern in main memory. On C200 Series machines, a register on the PBUS Interface Adapter (PIA) is read to locate the test pattern in main memory.

4.5.1.3 Subtest 102, HSP Memory Initialization

Subtest 102 initializes HSP memory after the local memory size is determined. The local memory is cleared from the word following the end of the EPROM to the end of installed memory. ATU memory and the device buffer memory is then initialized. The first PMAP register is pointed to the PCM and is searched for the first 2 Mbytes of physical memory. Once located, the map registers are initialized to point to the first 1 Mbyte of installed memory. The CCU slot number is used to index into a table in main memory. This table contains a 32-bit pattern initialized by the SPU. The HSP reads this pattern and returns it to main memory. The returned pattern is verified by the SPU. The pattern is the current time, in seconds, logically ANDed with a mask of `0xffffff00` and logically ORed with the CCU slot number.

4.5.1.4 Subtest 103, HSP Boot

> **NOTE**
>
> Before executing Subtest 103, Subtest 102 must have completed successfully. It is not possible to loop on this subtest.

Subtest 103 determines the size of local memory, and copies the EPROM to the last 16 Kbytes of installed local memory. A checksum is performed on local memory to verify it. The test sets memory protection for installed local memory to valid, read, write, and execute. The memory protection registers for uninstalled local memory are cleared. The interrupt status register is then cleared, and the HSP jumps to the local memory copy of the EPROM. Memory protection is enabled. The HSP slot number is used as an index into a table in main memory. The table is located in the first 1 Mbyte of installed main memory.

After the test begins, the HSP enters a loop while waiting for the SPU to place commands in main memory. The valid commands are listed in the following table:

Table 4–4, Valid SPU Commands for Subtest 103

COMMAND	DESCRIPTION
load	Copies main memory to HSP memory, a byte at a time. As each byte is moved, *load* tallies it into a checksum. After main memory is moved, it checks for parity errors. If no errors, the checksum is placed in main memory; otherwise, negation of calculated checksum goes in main memory. At completion, *load* waits for more commands.
reset	Simulates a reset by loading the supervisor-stack pointer (ssp) from memory location 0 and the *pc* from memory location 4.
jump	Jumps to the address specified in main memory. The *ssp* is set to the top of installed local memory.

> **NOTE**
>
> Subtest 103 checks only the *load* command.

4.5.2 Class 2 Subtests

Class 2 subtests test the HSP without a device attached to the HSP. This class of subtests is loaded into the HSP local RAM from main memory. The following functions are tested:

- PBUS interrupt operations
- HSP address translation
- Address translation error detection
- Output bus integrity
- Error status register verification
- Line clock interrupt operation

The following table lists the Class 2 subtests:

Table 4–5, Class 2 Subtests

SUBTEST	DESCRIPTION	TIME (min:sec)
210	PBUS Interrupt	00:36
220	PBUS Test-and-Set	<:02
230	ATU Memory Pattern	<:02
231	CSR Memory Pattern	<:02
232	Output Bus	<:02
233	Device Buffer Memory Pattern	<:02
240	ATU Local Translation—Physical Addresses	<:02
241	ATU Local Translation—Virtual Addresses	<:02
250	ATU Virtual Address Translation	<:02
251	ATU Memory Parity Error Detection	<:02
252	ATU *pte* Error Detection	<:02
270	Status Register Verification	<:02
280	Line Clock Interrupt	00:10

4.5.2.1 Subtest 210, PBUS Interrupt

Subtest 210 verifies the ability of the HSP to request and use the PBUS interrupt bus. It consists of three steps: an interrupt receive test, an interrupt transmit test, and an interrupt loopback test.

> **NOTE**
>
> The terms receive and transmit are from the perspective of the HSP.

Step 1: Receive Test

In the receive test, all 256 PBUS interrupts are sent by the SPU to the HSP, one at a time. After each interrupt is sent, the HSP verifies that only one interrupt was received and that the interrupt was received by the proper group. The group in which each

interrupt is received is determined by taking the modulo 4 residue of the interrupt number.

Step 2: Interrupt Transmit Test

In the interrupt transmit test, the HSP sends the SPU interrupt numbers 8, 9, 10, and 11, the four interrupts that the SPU is capable of receiving. The HSP verifies that the acknowledge for each interrupt is received and the SPU verifies the reception of the four interrupts.

Step 3: Interrupt Loopback Test

The interrupt loopback test causes all 256 interrupts to be sent and received by the HSP in each of the four possible groups. Like the receive test, as each interrupt is sent the HSP verifies that only one interrupt is received and that the acknowledge is also received.

4.5.2.2 Subtest 220, PBUS Test-and-Set

Subtest 220 verifies that all PMAP registers operate in test-and-set mode by placing each PMAP register in test-and-set mode in sequence. The PMAP register that logically follows the register under test is prepared for nontest-and-set operation and it verifies the operation.

A byte in the main memory command table is set to zero through the check window. It is read through the test window and the value returned is checked for a zero. The check window reads the byte in main memory and the value read is expected to be `0xff`. A second access through the test window verifies that the returned value is also `0xff`.

4.5.2.3 Subtest 230, ATU Memory Pattern

Subtest 230 performs column functionality, uniqueness, and bit functionality tests on the memory used by the Address Translation Unit (ATU) controller. It verifies that memory parity checking is operational by writing a byte with bad parity, reading it, and verifying that a parity interrupt occurs.

Since the PMAP registers and the ATU share a common memory, a uniqueness test over the two address spaces is also performed.

4.5.2.4 Subtest 231, CSR Memory Pattern

Subtest 231 tests the channel status memory by performing column functionality, uniqueness, and bit functionality tests. There is no parity on the Channel Status Register (CSR) memory.

4.5.2.5 Subtest 232, Output Bus

Subtest 232 checks the output bus by pattern testing two 32-bit words in the buffer memory used in device transfers. The first 32-bit location tests the bus in bit locations <31..0>, and the second checks the bus in bits <63..32>. Patterns of zero, all ones, alternating ones and zeros, walking ones, and walking zero are used in the subtest.

4.5.2.6 Subtest 233, Device Buffer Memory Pattern

Subtest 233 checks the device buffer memory for column functionality, uniqueness, and bit functionality. Parity is checked in the same way as in Subtest 230, ATU Memory Pattern.

4.5.2.7 Subtest 240, ATU Physical Address Mode Header

Subtest 240 checks the ability of the ATU controller to format PBUS headers while in physical address mode. The 68000 uses the File Description Register (FDR) register control bits to instruct the ATU to create PBUS headers.

ATU memory is initialized with the start address and byte count field such that the PBUS header is loaded with all zeros, all ones, a walking one, and a walking zero. As each header is generated, the ATU_OREG register is inspected for the correct header.

This subtest asserts the PBUS_TEST bit in the FDR register to inhibit the generated headers from being sent to the Multibus Control Unit (MCU) when the ATU is not functioning correctly.

4.5.2.8 Subtest 241, ATU Virtual Address Mode Header

Subtest 241 uses the FDR register to force the ATU to construct PBUS headers. The ATU memory is initialized so that the ATU constructs headers for data transfers, *pte* transfers, and *sdr* transfers in increasing levels of complexity. This is done by walking a one in the logical address that the ATU is translating. As each address is translated, the ATU_OREG register or the PBUS_HDR register, as appropriate, is checked for valid header contents.

This subtest asserts the PBUS_TEST bit in the FDR register to inhibit the generated headers from being sent to the MCU.

4.5.2.9 Subtest 250, ATU Virtual Address Translation

Subtest 250 checks the ATU logical address translation. The SPU fills a set of *sdr*s and *pte*s in main memory with pointers to nonexistent main memory. As the HSP walks a one through the logical address, it initializes only those locations in main memory and ATU memory that are required to complete the translation. As each translation occurs, the HSP verifies that the ATU only modified the needed locations in its memory. The *sdr*s and *pte*s in main memory are then restored to their initial state before the next logical address is checked.

Subtest 250 is executed for each ATU channel.

4.5.2.10 Subtest 251, ATU Memory Parity Error Detection

Subtest 251 tests the ATU parity error detection. It is tested by introducing a memory parity error into ATU register 6, the next address register, starting a translation, and verifying the error was logged in the CSR after the translation.

4.5.2.11 Subtest 252, ATU *pte* Error Detection

All *pte* entries used for HSP logical address translation must have bit <11> set to show that the selected *pte* is valid for I/O. Subtest 252 verifies that this bit is being checked by placing a *pte* level 2 in main memory without the bit set. A translation is started that reads the corrupted *pte*. The CSR register is checked for the global channel error flag set, as well as the I/O protection error.

The subtest is repeated with a corrupted *pte* level 1.

4.5.2.12 Subtest 270, Status Register Verification

Subtest 270 checks HSP error log recording functions by forcing various error conditions and verifying that they have been correctly reported. The subtest forces read, write, and execute errors on HSP local memory and verifies that they are reflected in the BERRLOG, ADDRHI, and ADDRLO registers.

PBUS read, write, and test and set errors are generated and the BERRLOG, PBERRLOG, ADRHI, and ADRLO registers are verified.

Finally, a DTACK timeout error is forced and the BERRLOG, ADRHI, and ADRLO registers are verified.

4.5.2.13 Subtest 280, Line Clock Interrupt

Subtest 280 enables the line clock interrupt on the HSP and waits for 600 line clock interrupts to occur. The SPU UNIX time of day is checked to ensure that the 600 interrupts took 10 seconds to occur.

4.5.3 Class 4 Subtests

Class 4 subtests verify the functionality of the HSP/HIA interface. These subtests are loaded into the HSP local memory from main memory and include tests for the following:

- Asynchronous and synchronous data transfers
- Address translation
- Clock selection
- Error detection and recovery

Table 4–6, Class 4 Subtests

SUBTEST	DESCRIPTION	TIME (min:sec)
4100	HIA Reset	00:22
4101	HIA Voltage	00:22
4110	Register Access—HSP clock 0	00:04
4111	Register Access—HSP clock 1	00:04
4112	Register Access—HSP clock 2	00:04
4113	Register Access—HSP clock 3	00:04
4120	Illegal Register Request	<:02
4130	Register Access Parity Error	<:02
4140	HIA Channel Interrupt	<:02
4180	Physical Address Read—HIA buffer 0	00:06
4181	Physical Address Read—HIA buffer 1	00:06
4182	Physical Address Write—HIA buffer 0	00:09
4183	Physical Address Write—HIA buffer 1	00:09
4184	Partial Longword Transfers	00:41
4190	Virtual Address Read Test—1 longword	00:12
4191	Virtual Address Read Test—2 longword	00:12
4192	Virtual Address Read Test—512 longword	00:12
4193	Virtual Address Read Test—513 longword	00:12
4194	Virtual Address Read Test—1000 longword	<:15
4195	Virtual Address Write Test—1 longword	00:26
4196	Virtual Address Write Test—2 longword	00:26
4197	Virtual Address Write Test—512 longword	00:25
4198	Virtual Address Write Test—513 longword	00:26
4199	Virtual Address Write Test—1000 longword	00:26
4200	HSP/HIA Clock Selection	04:16
4210	HSP Channel Functionality	00:11
4220	HSP ATU Parity Error Detection	<:02
4230	HSP I/O Access Bit Verification	<:02
4240	ATU PBUS Error Detection	00:06
4250	HSP Checkword Verification	<:02
4260	No Transfer Response	<:02
4270	Illegal Command Detection	<:02
4280	Data Parity Error Transfer	<:02
4990	HIA Internal Loopback—Buffer 0	<:02
4991	HIA Internal Loopback—Buffer 1	<:02
4992	HIA External Loopback—Buffer 0	<:02
4993	HIA External Loopback—Buffer 1	<:02
4994	HIA Local Slave Mode Transfer—Read	00:47
4995	HIA Local Slave Mode Transfer—Write	03:54
4999	CONVEX Register Compliance	<:02

4.5.3.1 Subtest 4100, HIA Reset

Subtest 4100 verifies that the HSP can reset the HIA. The HSP forces the HIA main controller to its idle condition. It then sets the DMAENA bit in the HIA CONTROL register and then asserts the reset signal to the HIA. The CONTROL register is then read and the subtest verifies that the DMAENA bit has been reset.

4.5.3.2 Subtest 4101, HIA Voltage

Subtest 4101 measures and verifies the four voltages in the HIA chassis to ensure that they are within 5% of their nominal value. The nominal values used are +12V and -12V, and +5V and -5V.

4.5.3.3 Subtests 4110–4113, Register Access

Subtests 4110 through 4113 test the HSP asynchronous data transfer function (device register access) by pattern testing the following:

- HIA buffer memories in both 8-bit and 16-bit modes
- HIA parity memory in 8-bit mode
- The Direct Memory Access (DMA) register file in 16-bit mode
- The GLOMSK register in 16-bit mode

Uniqueness, bit, and column functionality tests are performed on each of the above. The interface clock is changed from 0 through 3 in Subtests 4110 to 4113 respectively.

4.5.3.4 Subtest 4120, Illegal Register Request

The HSP makes illegal requests to the HIA and expects the HIA to return an illegal register response code. Subtest 4120 verifies that the error is correctly logged in the HSP BERRLOG and ADDRHI/LO registers, and also in the HIA REGERR register. The following accesses are made to force the errors:

- 16-bit access to the HIA parity memory
- 8-bit access to the DMA register file
- 8-bit access to the HIA CONTROL register
- 8-bit and 16-bit accesses to a nonexistent register in the HIA address space

4.5.3.5 Subtest 4130, Register Access Parity Error

Subtest 4130 verifies that parity generation and detection is working for register accesses. The HSP makes a register access request with bad parity in the opcode, address, and data fields. For each request, the HSP verifies that the HIA sends a command parity error response and that the register request ends in a bus error.

The HSP then corrupts the parity on a longword in the HIA buffer memory. Then a register access is made to the corrupted location and the subtest expects the register access to result in a bus error and the HIA to have sent a fatal error response to the HSP.

4.5.3.6 Subtest 4140, HIA Channel Interrupt

The HSP resets the HIA by asserting the reset line on the HSP interface. The HSP then reads the HIA GLOINT register and clears the HIA FIR to clear any pending interrupts. The HSP then disables device interrupts in the HSP IER.

The subtest then uses the HIA FIR register to force a channel interrupt for each channel. For channel zero, the subtest verifies that the interrupt does not occur until the interrupt enable bits in the HIA CONTROL register and the HSP IER register are set. For all channels, the subtest verifies that the interrupt does not occur until the channel interrupt is enabled in the HIA IER.

4.5.3.7 Subtests 4180–4183, Synchronous Data Transfers with Physical Addresses

These subtests exercise the HSP synchronous data transfer functions without address translation. The amount of data and the addresses used in the subtests varies. These combinations test the HSP end of logical page detection and data buffer valid flush operations. The subtests are divided into reads and writes. A data equals address pattern is used in all subtests. After each transfer, the destination area is checked to verify that all the data was transferred and that no additional data was written. All data lengths and addresses are longword aligned. Each subtest tests the data transfers with a sliding one and zero in the least significant 12 bits of the address.

4.5.3.8 Subtest 4184, Partial Longword Transfers

Correct handling of partial longword transfers is verified by programming the HIA to transfer various transfers each with a different starting address and byte count. The addresses range from logical address 0 to logical address 16 and the byte count varies from 1 to 16. This causes all combinations of leading partial, block transfer, and trailing partial to be tested.

4.5.3.9 Subtests 4190–4199, Virtual Address Read and Write

These subtests exercise the HSP synchronous data transfer logical address translation operation. They vary in the amount of data they transfer and in the direction of the transfer. All data is longword aligned. The subtest virtual addresses are generated by sliding a one from bit <3> through bit <31>. This causes the address translation logic to be checked in an incremental fashion. For all transfers, only the required ATU memory is initialized with valid data. The remaining portions are initialized with pointers to nonexistent main memory. A data equals address pattern is used for all transfers. The destination is initialized to the negation of these patterns for each transfer attempted. After each transfer, ATU memory registers 0, 1, 3, 4, 5, and 11 are checked for the expected values. The destination area is checked for valid data.

4.5.3.10 Subtest 4200, HSP/HIA Clock Selection

Subtest 4200 tests the various data rates available for synchronous data transfers. For each possible clock combination, a 32-Kbyte transfer is initiated. After the transfer, the destination is read to verify data integrity. Reads and writes are both tested.

4.5.3.11 Subtest 4210, HSP Channel Functionality

Subtest 4210 verifies that all HSP channels are functional by starting a 32 Kbyte transfer on each one. After the transfer, the destination is read to verify data integrity. Reads and writes are both tested.

4.5.3.12 Subtest 4220, HSP ATU Parity Error Detection

Subtest 4220 verifies ATU parity error detection synchronous data transfers. The 68000 initializes ATU memory in a normal fashion except that register 11 is initialized with inverted parity. The HIA is programmed to perform one longword read from main memory. After the transfer, the subtest verifies that the CSR logs the error as a memory parity error during a Segment Descriptor Register (SDR) read. Additionally, the CSR is checked to ensure that the NEW_SDRS bit is set, the channel enable is reset, and that the global error bit is set.

4.5.3.13 Subtest 4230, HSP I/O Access Bit Verification

Subtest 4230 verifies that the I/O access bit of *pte* levels 1 and 2 is processed correctly. For each *pte* level, a set of *pte*s is initialized in main memory with one valid entry and all others invalid. A 1-longword transfer is started from a valid logical address. The subtest verifies transfer completion without error. For PBUS reads, this verifies that a prefetch from a protected area does not produce an error response by the HSP if the data was not sent to the HIA. A second 2-longword transfer is then started. This causes the HSP to try to send data from a page without I/O access enabled. The subtest expects that the CSR will reflect an I/O protection error, that the NEW_SDRS bit is set along with the global error flag, that the *pte* level where the error occurred is properly encoded in the CSR, that the direction of the transfer is also properly saved in the CSR, and that the channel enable bit is reset. The subtest verifies both reads and writes.

4.5.3.14 Subtest 4240, ATU PBUS Error Detection

Subtest 4240 verifies that the HSP can recover from PBUS errors received during various stages of processing. Initially, a pair of *pte*s level 2 are initialized in main memory. The first contains a pointer to a valid location in main memory, the second contains a pointer to nonexistent memory. A one longword transfer is started such that the data used comes from the valid page and the prefetch (on reads) comes from the nonexistent page. Then the subtest verifies that the transfer completes without error. For reads, the subtest verifies that the NEW_SDRS bit is set in the CSR.

The operation is restarted with a longword count that causes the data from the second page to be used. The CSR is checked to verify that it has the PBUS_BUS_ERROR bit set, that the CSR shows the error as having occurred during a data access, and that the CSR has the enable bit turned off and the NEW_SDRS bit on.

The above procedure is repeated moving the bad memory pointer from a *pte*2 to a *pte*1, *sdr*, and finally to the pointers to the *sdr*s changing the expected error condition appropriately. Both reads and writes are verified.

4.5.3.15 Subtest 4250, HSP Checkword Verification

Subtest 4250 verifies the proper operation of the HSP checkword by causing the HIA to do a write to main memory of a single longword with an incorrect checkword. The checkword is corrupted one bit at a time for each of the 64 bits of a longword. After each transfer, the ATU CSR is checked to verify that the checkword error bit is set.

4.5.3.16 Subtest 4260, No Transfer Response

Subtest 4260 verifies the functionality of the channel enable bit by having the HIA start a transfer on a channel without the enable bit set. It also verifies that the HSP sent a no transfer response code.

4.5.3.17 Subtest 4270, Illegal Command Detection

Subtest 4270 causes the HIA to send a partial write request for an illegal address/byte count combination. The subtest verifies that the HSP sends an invalid command response to the HIA. The CSR is checked to make sure that an error is not logged in the CSR.

4.5.3.18 Subtest 4280, Data Parity Error Transfer

Subtest 4280 verifies that the HSP can detect data parity errors during synchronous transfers. The subtest programs the HIA to send data with inverted parity. Each of the 8 bytes in a longword is tested in succession. After each transfer, the ATU CSR is checked to ensure that the parity error was logged, the HSP is expected to have sent the HIA a block checksum parity error response code, and the IBPAR register on the HSP is read to verify that the correct byte in error was logged.

4.5.3.19 Subtest 4990, 4991, HIA Internal Loopback

These subtests verify the integrity of the data alignment bus internal to the HIA. One of the two data buffers in the HIA is initialized with a pattern of walking '1's and walking '0's. The HIA is programmed to transfer the initialized buffer into the other HIA buffer. The destination buffer is then checked for data integrity. Subtest 4990 copies HIA buffer 0 to buffer 1. Subtest 4991 copies buffer 1 to buffer 0.

4.5.3.20 Subtest 4992, 4993, HIA External Loopback

These subtests verify data bus integrity on the user interface. One of the two HIA data buffers is initialized to a pattern of walking '1's and walking '0's. The initialized buffer is then transferred to the other buffer and the destination is checked for data integrity. The test is repeated for all four data port sizes. For data port sizes less than 64 bits, the unused portions of the data bus are initialized to the contents of the used portion, thus, the data pattern is modified in the loopback process. Subtest 4992 copies HIA buffer 0 to buffer 1. Subtest 4993 copies buffer 1 to buffer 0.

4.5.3.21 Subtest 4994, 4995, HIA Local Slave Mode Transfers

These subtests verify that the HIA can transfer data in a pseudo slave mode. The subtest tests all data port sizes for logical addresses from 0 to 7, for byte counts of 1 to 16, 4072 to 4089, and 8000 to 8015. After each transfer, the subtest verifies that no errors were logged and that the data was transferred correctly. Subtest 4994 tests transfers to the HIA, and Subtest 4995 tests transfers to main memory.

4.5.3.22 Subtest 4999, CONVEX Register Compliance

Subtest 4999 verifies that the device attached to the HSP complies with the HIA standards. The register at `0xc00000` is read and its contents is verified to be one of, `0x0080`, `0x0010`, `0x0020`, `0x0020`, or `0x0030`. All valid values will cause a device interrupt test to be performed in the same fashion as Subtest 4140, Channel Interrupt Test, except that the user may specify on which channels the test is to be performed by answering the correct prompt when *io4120* is invoked. If the value read from the id register at `0xc00000` is `0x0020`, `0x0040`, or `0x0080`, a read of `0xc00086` is made to verify the presence of the GLOINT register. If the value read from the id register is `0x0030`, `0x0040`, or `0x0080`, a read of a portion of the DMA channel memory is made for all channels specified by the user. Each channel will have a read made for registers at offsets of `0` through `0xe`.

4.5.4 Class 5 Subtests

Class 5 tests verify the functionality of the user interface on the HIA, including:

- Register access to user device address space
- User device register error detection and correction
- User interface interrupts
- Data transfers to and from the user device in all data port sizes and data alignment addresses
- Data transfers in all four data transfer modes
- User interface handshake tests
- Data modulation tests
- User interface error detection and recovery
- Correct operation at all possible clock rates

The Class 5 subtests are listed in the following table:

Table 4–7, Class 5 Subtests

SUBTEST	DESCRIPTION	TIME (min:sec)
5000	User Interface Reset	<:02
5010	FSE RAM	<:02
5020	User Register Error	<:02
5030	User Interrupt	<:02
5180	HIA/FSE Small Read–64-bit datum	<:02
5181	HIA/FSE Small Read–32-bit datum	00:04
5182	HIA/FSE Small Read–16-bit datum	00:09
5183	HIA/FSE Small Read–8-bit datum	00:18
5184	HIA/FSE Large Read–64-bit datum	00:03
5185	HIA/FSE Large Read–32-bit datum	00:06
5186	HIA/FSE Large Read–16-bit datum	00:14
5187	HIA/FSE Large Read–8-bit datum	00:39
5190	HIA/FSE Small Write–64-bit datum	<:02
5191	HIA/FSE Small Write–32-bit datum	00:05
5192	HIA/FSE Small Write–16-bit datum	00:11
5193	HIA/FSE Small Write–8-bit datum	00:24
5194	HIA/FSE Large Write–64-bit datum	00:03
5195	HIA/FSE Large Write–32-bit datum	00:07
5196	HIA/FSE Large Write–16-bit datum	00:17
5197	HIA/FSE Large Write–8-bit datum	00:42
5200	32k Slave Mode Read–64-bit datum	00:05
5201	32k Slave Mode Read–32-bit datum	00:03
5202	32k Slave Mode Read–16-bit datum	00:03
5203	32k Slave Mode Read–8-bit datum	00:03
5204	32k Slave Mode Write–64-bit datum	00:04
5205	32k Slave Mode Write–32-bit datum	00:04
5206	32k Slave Mode Write–16-bit datum	00:04
5207	32k Slave Mode Write–8-bit datum	00:03
5300	32k Chain Mode Read–64-bit datum	00:04
5301	32k Chain Mode Read–32-bit datum	00:04
5302	32k Chain Mode Read–16-bit datum	00:04
5303	32k Chain Mode Read–8-bit datum	00:03
5304	32k Chain Mode Write–64-bit datum	00:05
5305	32k Chain Mode Write–32-bit datum	00:04
5306	32k Chain Mode Write–16-bit datum	00:03
5307	32k Chain Mode Write–8-bit datum	00:03
5400	32k Normal Mode Transfer–64-bit datum	00:08
5401	32k Normal Mode Transfer–32-bit datum	00:06
5402	32k Normal Mode Transfer–16-bit datum	00:06
5403	32k Normal Mode Transfer–8-bit datum	00:06
5500	32k Extended Mode Transfer–64-bit datum	00:07
5501	32k Extended Mode Transfer–32-bit datum	00:06
5502	32k Extended Mode Transfer–16-bit datum	00:06
5503	32k Extended Mode Transfer–8-bit datum	00:06

Table 4–7, Class 5 Subtests (continued)

SUBTEST	DESCRIPTION	TIME (hr:min:sec)
5600	Kill Channel	00:04
5610	Data Modulation	00:55
5620	DMA Enable Bit	00:01
5640	Data Parity Detection	00:05
5650	Transfer Error from HSP	00:20
5660	User Read Parity Error Signal	<:02
5670	Clock Selection	4:03:06

4.5.4.1 Subtest 5000, User Interface Reset

Subtest 5000 verifies that the HSP can reset the HIA. The HSP forces the HIA main controller to its idle condition. It then sets the DMAENA bit in the HIA CONTROL register and then asserts the reset signal to the HIA. The CONTROL register is then read and the subtest verifies that the DMAENA bit has been reset.

Subtest 5000 then reads the FSE errstat register and verifies that the power-up bit is cleared. The subtest then resets the user device by clearing the HIA CONTROL register, and setting the HIA CONTROL user run bit. The FSE errstat register is then reread and the subtest expects the power up bit to be set.

4.5.4.2 Subtest 5010, FSE RAM

Subtest 5010 consists of bit, column, and uniqueness tests for the FSE data buffer RAM and the FSE command queue. The data buffer RAM tests are performed twice, once using 16-bit accesses and once using 8-bit accesses. The command queue ram is only tested with 16-bit accesses.

4.5.4.3 Subtest 5020, User Register Error

The NORESPON, BADACK, and BADERR registers on the FSE are read to force a register access error. This subtest verifies that each access results in a bus error.

The subtest then reads register addresses starting with `c20001` and slides the one bit until the address is `c30000`. For each access, the subtest expects a bus error. After each access, the FSE LASTADRH and LASTADRL registers are read and verification is made that the address was latched correctly.

The parity on a 16-bit word in the FSE data buffer is then corrupted and the subtest verifies that reading that location results in a bus error. Each of the two parity bits in the word is checked individually.

4.5.4.4 Subtest 5030, User Interrupt

Subtest 5030 uses the FSE interrupt register to force an interrupt condition to the HIA. When first asserted, user interrupts are masked in the HIA GLOMSK register. The subtest verifies that no interrupt is sent to the HSP and that the interrupt is logged in the HIA GLOINT register. User interrupts are then unmasked, and the subtest verifies that the HSP receives the interrupt and that it can clear the interrupt by writing to the FSE interrupt register.

4.5.4.5 Subtest 5180–5187, 5190–5197, HIA/FSE Local Transfers

This series of subtests verify that the HIA and FSE can transfer data across the user interface. No synchronous data is transferred to or from the HSP. The subtests vary in the number of bytes transferred, the direction of the transfer, and the data port size used. The subtests listed as small transfers in the Class 5 Subtests table use byte counts of 1 to 16, inclusive. Subtests listed as large use byte counts of 4,080 to 4,096. For each iteration of the byte count, the subtest varies the logical address used from zero to seven, inclusive, in order to test all possible data alignments.

The HSP initializes the source and destination memory areas by register access. The HIA is then programmed to perform the transfer and the transfer is started. After the transfer, the HSP checks the HSP and HIA error registers and then reads the destination area to verify the data was transferred correctly.

4.5.4.6 Subtest 5200–5207, Slave Mode Transfers

This series of subtests verify that the HSP/HIA/FSE can transfer data in slave mode. Each subtest transfers 32-Kbytes. The subtests vary in the direction of the transfer and in the data port size used. Each subtest varies the logical address used from zero to seven, inclusive, in order to test all possible data alignments.

The HSP initializes the source and destination memory areas by register access. The HSP/HIA/FSE is then programmed to perform the transfer. After the transfer, the HSP checks the HSP/HIA/FSE error registers and then reads the destination area to verify the data was transferred correctly. All transfers take place on channel zero.

4.5.4.7 Subtest 5300–5307, Chain Mode Transfers

This series of subtests verify that the HSP/HIA/FSE can transfer data in chain mode. Each subtest transfers 32k bytes. The subtests vary in the direction of the transfer and in the data port size used. Each subtest varies the logical address used from zero to seven, inclusive, in order to test all possible data alignments.

The HSP initializes the source and destination memory areas by register access. The HSP/HIA/FSE is then programmed to perform the transfer. Each of the 16 channels is programmed to transfer one-sixteenth of the data. The last channel in the chain is disabled in the HIA. The transfer is then started on the first 15 channels. After the transfer, the subtest verifies that the first 15 channels finished without any error. The enable bit on the last channel is then set, and the subtest expects the last channel to transfer its data. After the transfer, the HSP checks for errors, and then reads the destination area to verify the data was transferred correctly.

4.5.4.8 Subtest 5400–5403, Normal Mode Transfers

This series of subtests verify that the HSP/HIA/FSE can transfer data in normal mode. Each subtest transfers 32-Kbytes. The subtests vary in the the data port size used. All subtests in this range are read/write subtests. Each subtest varies the logical address used from zero to seven, inclusive, in order to test all possible data alignments.

These subtests program the HSP/HIA/FSE to transfer data from main memory to the FSE, and then from the FSE back to a second location in main memory. The channels are programmed to transfer the data in such a manner that a read is followed by a write, then another read, and then another write. This verifies that all read/write combinations are functional.

After the transfer, the subtest verifies that the data in the FSE is correct and then it verifies that the second copy in main memory is correct.

4.5.4.9 Subtest 5500–5503, Extended Mode Transfers

This series of subtests verify that the HSP/HIA/FSE can transfer data in extended mode. Each subtest transfers 32-Kbytes. The subtests vary in the data port size used. All subtests in this range are read/write subtests. Each subtest varies the logical address used from zero to seven, inclusive, in order to test all possible data alignments.

These subtests program the HSP/HIA/FSE to transfer data from main memory to the FSE, and then from the FSE back to a second location in main memory. The channels are programmed to transfer the data in such a a manner that a read is followed by a write, then another read, and then another write. This verifies that all read/write combinations are functional.

After the transfer, the subtest verifies that the data in the FSE is correct and then it verifies that the second copy in main memory is correct.

4.5.4.10 Subtest 5600, Kill Channel

Subtest 5600 verifies that the HIA can process kill channel signals correctly. Kill channel processing is tested by programming transfers in all four transfer modes, and then having the FSE assert kill channel. Each transfer mode is tested for both reads and writes. After the signal has been asserted, the subtest verifies that the appropriate fields in the HIA and FSE were updated correctly. Transfers in slave and chain mode are expected to terminate the transfer on all subsequent channels after the signal is asserted. Transfers in normal and extended mode are expected to terminate only the channel on which the signal is received. Subsequent channels are expected to complete without error.

4.5.4.11 Subtest 5610, Data Modulation

Subtest 5610 verifies that the HIA can process the read/write signals from FSE. The correct handling of the read/write handshake lines on the user interface is tested by programming the FSE to transfer data while modulating the handshake lines at varying intervals. After the transfer, the subtest verifies that the data was transferred correctly.

4.5.4.12 Subtest 5620, DMA Enable Bit

Subtest 5620 verifies that turning off the DMA enable bit on an active channel will shut down a transfer in a graceful manner. The FSE is programmed to perform a transfer and then have it hold off sending or receiving data for many clock cycles to give the HSP a chance to turn off the DMA enable bit. The subtest then verifies that the data transferred before the hold command is correct. Both reads and writes are tested in normal and extended transfer modes.

4.5.4.13 Subtest 5640, Data Parity Detection

Subtest 5640 verifies that HIA data parity checkers on the user interface are operational. The FSE is programmed to send the HIA data with bad parity in it. The test is made in each of the four transfer modes. For slave and chain modes, the subtest verifies that the transfer stops after the error is encountered and that the HIA resets the DMA enable bit in its control register. For normal and extended mode, the subtest verifies that only the channel where the error was made is disabled and the rest of the transfer finishes normally. For all transfer modes, the register file is examined to verify that the error is properly logged.

4.5.4.14 Subtest 5650, Transfer Error from HSP

Subtest 5650 verifies that the HIA can process transfer error responses from the HSP. Two transfers are programmed which result in the HSP sending a *no transfer* response and a *fatal error* response. These two error responses are generated for reads and writes in each of the four transfer modes for both reads and writes. Errors in slave and chain modes are expected to terminate the transfer. Errors in normal and extended mode are expected to terminate the transfer on the channel receiving the error and continue with the remaining channels.

4.5.4.15 Subtest 5660, User Read Parity Error Signal

Subtest 5660 verifies that the HIA can process the user read parity error signal correctly. A read transfer is programmed in which the FSE asserts the ReadParErr signal. The subtest verifies that the HIA logs the error correctly. Errors in slave and chain modes are expected to terminate the transfer. Errors in normal and extended mode are expected to terminate the transfer on the channel receiving the error and continue with the remaining channels.

4.5.4.16 Subtest 5670, Clock Selection

Subtest 5670 performs Subtests 5200 through 5507 varying the HSP clock, HIA outclk, and HIA busclk over all possible clock combinations. It does not change the system backplane clock. Two passes through all the possible clock rates are made. The first pass forces the HIA to throttle the data, and the second pass does not.

Appendix A
Reporting Problems

A.1 Overview

This appendix introduces the CONVEX Technical Assistance Center (TAC) and the *contact* utility. The *contact* utility is an online system for reporting problems to the TAC. To learn *contact* by using it, enter **contact** at the system prompt and then answer the questions as they appear on the screen. To find out more about using *contact*, read through this appendix. It describes prerequisites and tips for using *contact* and the step-by-step process *contact* takes you through.

A.2 Technical Assistance Center

The CONVEX Technical Assistance Center (TAC) is staffed by technical specialists who can address the diverse questions and problems that arise in a supercomputing environment. If you have a hardware, software, or documentation problem, contact the TAC. This group stands ready to solve such problems.

A.3 The *contact* Utility

The TAC recommends using the *contact* utility to report a hardware, software, or documentation problem. The *contact* utility is an interactive utility that helps the TAC track reports and route them to the the CONVEX personnel most qualified to fix them.

After invoking *contact*, it prompts for information about the problem. When you finish your report, *contact* electronically mails it to the TAC. You are notified within 48 hours that the TAC has received your report.

A.4 Prerequisites

To use *contact* requires

- a UNIX-to-UNIX Communication Protocol (UUCP) connection to the TAC
- the full path name of the program or utility in question
- the version number of the program or utility in question

A.4.1 UUCP Connection

Before using *contact*, check with your system administrator to be sure there is a UUCP connection to the TAC. A UUCP connection allows files to be copied from one UNIX system to another. The *uucp* (UNIX-to-UNIX copy) command relies on either a dial-up or hard-wired UUCP communication line.

A.4.2 Finding the Program Path Name

To determine the full path name of the program or utility in question, use the *which* command. The following screen illustrates using the *which* command to find the full path name of the loader (*ld*) utility:

```
>which ld
/bin/ld
>
```

In this example, the full path name of the loader is */bin/ld*.

For more information on the *which* command, refer to the *which*(1) man page. You can also use the *info* online information system. Enter **info which** at the system prompt. If you use the C shell (*csh*), you can also use the *whence* command to find the program path name. The *whence* command works like *which*, only faster.

A.4.3 Finding the Program Version Number

To determine the version number of the program or utility in question, use the *vers* command. The following screen illustrates using the *vers* command (enter **vers**, then the path name of the program or utility) to find the version number of the loader (*ld*) utility.

```
>vers /bin/ld
/bin/ld: 7.0
>
```

In this example, the loader utility version number is 7.0.

For more information on the *vers* command, refer to the *vers*(1) man page. You can also use the *info* online information system. To do so, enter **info vers** at the system prompt.

A.5 Tips on Using the *contact* Utility

The *contact* utility is interactive and easy to use. This section lists tips to help use it efficiently. In particular, this section tells how to

- use a *.contact* file
- abort a contact session
- resubmit an aborted report
- suspend a contact session
- move from one prompt to another
- use tilde-escape sequences in the *contact* utility

A.5.1 Using a *.contact* File

When invoked, *contact* prompts for information regarding the problem. The first prompt is for your name, title, phone number, and company name. You can, however, create a *.contact* file to skip this first prompt. Follow these steps:

1. Create a *.contact* file in your home directory.
2. Enter your name, job title, phone number, and company name, each on a new line.

When you invoke *contact*, it automatically includes the *.contact* file as input for the first prompt and proceeds to the next prompt.

A.5.2 Aborting the Report

To abort a contact report, either enter the interrupt key (usually CTRL-C) or choose the abort option when prompted by the *contact* utility. Using CTRL-C to abort does not save the contents of the report. Using the abort option saves the contents of the report in a file named *dead.report* in your home directory.

A.5.3 Submitting the *dead.report* File

When aborting a contact session, the *contact* utility saves the report in a file named *dead.report* in your home directory. Using the *contact* command with the *-r* option automatically merges the contents of the *dead.report* file into the new contact session. Enter

```
contact -r
```

and *contact* finds the *dead.report* file in your home directory and merges it into the contact report. You can then edit the report. When you end the editing session, *contact* returns to the final prompt, which asks you to review, edit, submit, or abort the report.

A.5.4 Suspending a Report

Sometimes it is necessary to stop in the middle of a contact report and return to the shell (for instance, to suspend the contact session to find the program path name or version number). To suspend the contact session, press CTRL-Z. To return to the contact session, enter **fg**. Using CTRL-Z and the *fg* (foreground) command lets you switch back and forth between the *contact* utility and the shell. You cannot, however, use CTRL-Z and *fg* to switch back and forth if you are using a Bourne shell (*sh*).

A.5.5 Ending a Response

The *contact* utility prompts for information pertinent to your hardware, software, or documentation question. Some prompts require one-line responses; to move to the next prompt, press RETURN. Other prompts require more than a one-line response; to move to the next prompt, press CTRL-D.

A.5.6 Tilde-Escape Sequences

The *contact* utility treats input beginning with a tilde (~) as a special sequence. The character following the tilde is considered a request for a special function. The following tilde sequences are recognized by *contact*:

~e	Start the text editor (defined in your EDITOR environment variable).
~h	Display a list of available tilde-escape sequences.
~p	Print the contact report to the terminal screen.
~r *filename*	Read the contents of *filename* as a response to the current prompt. Some prompts require only a one-line response. This tilde-escape sequence only works for prompts that allow more a than one-line response.
~~	Insert a single tilde as the first character in the line.

A.6 Using the *contact* Utility

The *contact* utility prompts for the following information:

- your name, title, phone number, and corporate name
- the name and version of the product involved
- a one-line summary of the problem
- a detailed description of the problem
- the priority of the problem
- instructions on how to reproduce the problem
- comments about the problem
- comments about the documentation supporting the problem
- files to include in the contact report

The following is a step-by-step discussion of these prompts:

1a. To invoke the *contact* utility, enter **contact** at the system prompt. The system responds with a welcome message and a series of questions regarding your hardware, software, or documentation question. The following screen illustrates the *contact* command and the system response:

```
>contact
Welcome to contact version 0.11 ()

Enter your name, title, phone number, and corporate name (^D to terminate)
>
```

1b. If there is a *.contact* file in your home directory, *contact* skips the first prompt. The following screen illustrates the *contact* command and the system response when a *.contact* file is in your home directory:

```
>contact
Welcome to contact version 0.11 ()

Enter the name of the product involved
>
```

2. The *contact* utility prompts for the version number of the product. If you do not know the version number, use CTRL-Z to suspend the session. Use the *which* (or *whence* if using *csh*) and *vers* commands to find the version number of the product. Use the *fg* command to return to the session and enter the version number in the form X.X or X.X.X.X.
3. The *contact* utility prompts for a one-line summary of the problem. This summary is the subject header in any further correspondence regarding the problem. Make this summary as descriptive as possible in one line.
4. The *contact* utility prompts for a detailed description of the problem. Make this description as complete as possible. Include source code and a stack backtrace whenever possible. (Refer to the *adb*(1) or *csd*(1) man page for information on obtaining a stack backtrace.) The more information provided, the quicker the TAC can isolate and solve the problem.
5. The *contact* utility prompts for the priority of the problem. The following screen illustrates this prompt and the priority levels from which to choose; you must enter a priority number.

```
Enter a problem priority, based on the following:
1) Critical    - work cannot proceed until the problem is resolved.
2) Serious     - work can proceed around the problem, with difficulty.
3) Necessary   - problem has to be fixed.
4) Annoying    - problem is bothersome.
5) Enhancement - requested enhancement.
6) Informative - for informational purposes only.
>
```

6. The *contact* utility prompts for an explanation of how to reproduce the problem. Include the command syntax and options you used and anything else you did to make your program run.
7. The *contact* utility prompts for any other pertinent comments. Include any relevant information.
8. The *contact* utility prompts for suggestions regarding the documentation supporting the product. Indicate if the documentation could be revised to address the question.
9. The *contact* utility asks for the names of files necessary to reproduce the problem. The following screen illustrates the *contact* prompt and sample user response:

```
Are there any files that should be included in this report (yes | no)?
>yes
Please enter the names of the files, one to a line (^D to terminate)
>test.f
>~/subroutines/sub.f
>
```

> **NOTE**
>
> Tilde-escape sequences are not recognized in responses to this prompt. Instead, *contact* treats a tilde in this section to mean your home directory. This convention is based on use of the tilde for expanding file names in *csh*.

If the files specified are small text files, they are automatically included in the contact report. If the files are too big to be included in this report, *contact* gives further instructions on how to submit these files.

To specify a directory, combine the directory files into a single file using the *tar* command (refer to the *tar*(1) man page for further information) or enter each file name in the directory on a single line in the contact report.

10. The *contact* utility prompts you to review, edit, submit, or abort the contact report. The following screen illustrates this prompt:

```
Please select one of the following options:
1) Review the problem report.
2) Edit the problem report.
3) Submit the problem report.
4) Abort the problem report.
>
```

Choose the number of the option you want to select. These options let you do the following:

Review — Review the text of your contact report. You are then prompted again to select an option.

Edit — Edit the text of the contact report. If you choose to edit the report, *contact* puts you in your default text editor.

Submit — Send the report to the CONVEX TAC. You are notified within 48 hours that the TAC has received the report. This option exits the *contact* utility and returns you to the shell environment.

Abort — Save the text of your report in a file named *dead.report* in your home directory. This option exits the *contact* utility and returns you to the shell environment.

Index

Numeric

A

C

D

E

F

H

I

K

L

M

N

O

P

R

S

T

U

V

W

CONVEX HSP/HIA Subsystem Test (*io4120*) Diagnostics Manual
Document No. 760-003630-000
First Edition

Reader's Forum

Please use this form to submit comments or questions concerning the clarity and service of this manual. Constructive critical comments are most welcome and help us continue in our efforts to generate quality customer documentation. Please list the page number for questions or comments.

_____

_____

_____

_____

_____

_____

_____

_____

From:

Name_____Title_____

Company_____Date_____

Address and Phone No._____

FOR ADDITIONAL INFORMATION OR DOCUMENTATION:

Location	Phone Number
From all locations in continental U.S.	1(800)952-0379
From locations in Alaska & Hawaii	1(214)497-4379
From locations in Canada	1(800)345-2384
From all other locations	Contact nearest CONVEX office

Direct mail orders to:

CONVEX Computer Corporation
Customer Service
PO Box 833851
Richardson TX 75083-3851 USA

(Fold Here First)

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1046 RICHARDSON, TEXAS

POSTAGE WILL BE PAID BY ADDRESSEE

CONVEX Computer Corporation
Customer Service
PO Box 833851
Richardson TX 75083-3851

(Fold Here Second)

(Tape or Staple)