CONVEX Multibus I/O Subsystem (*io4000*) Diagnostics Manual

Document No. 760-001830-000

First Edition
May 1991

CONVEX Computer Corporation
Richardson, Texas USA

CONVEX Multibus I/O Subsystem (io4000) Diagnostics Manual
Order No. DHW-230
First Edition

Printed in the United States of America

Revision Sheet

CONVEX Multibus I/O Subsystem (io4000) Diagnostics Manual

Edition	Document No.	Date	Description
First	760-001830-000	May 1991	First release. Contains the *io4000* diagnostic test information from the *CONVEX PBUS I/O Systems Diagnostics Manual.*

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

Appendixes

List of Tables

List of Figures

Preface

Purpose and Intended Audience

This manual explains how to run the *io4000* diagnostic, which verifies the functionality of a specified Input Output Processor (IOP). This document is not a tutorial, but rather a reference for the users of the *io4000* 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 *io4000* 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. Multibus I/O Subsystem Test (*io4000*)**—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-230.
The document number for this manual is 760-001830-000.

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

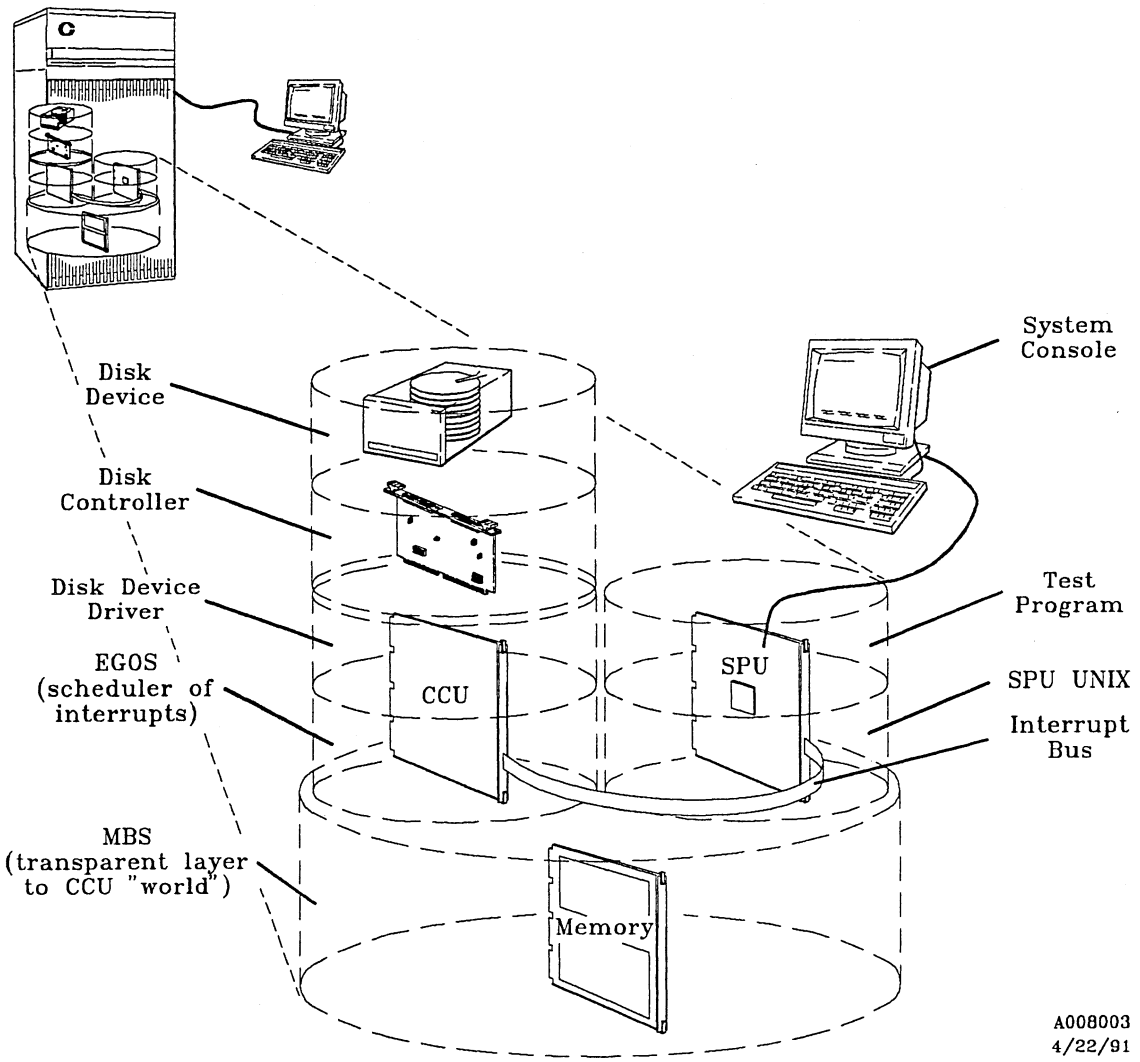
The VME interface version of EGOS is designed with a scheduler for the VIOP and is called VIOP EGOS. VIOP EGOS supports event driven device drivers as well as process type device drivers. VIOP EGOS utilizes a *sleep/wakeup* type of process control that improves efficiency of the device driver and makes 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
Multibus I/O Subsystem Test (*io4000*)

4.1 Overview

The *io4000* test verifies the functionality of a specified Input Output Processor (IOP). The *io4000* test verifies the following on the IOP(s) specified:

- The 68000 on the IOP can correctly execute instructions
- The IOP local memory is functional
- The memory protection registers are operational
- The cache memory can be written to and read from, and the associated cache control bits are functional
- The four Multibus interfaces operate in loopback mode
- The IOP can boot a program from memory
- The voltages on all installed Multibus chassis are within tolerance

The functional test uses main memory as a communication port from the Service Processor Unit (SPU) to or·from the IOP(s) being tested.

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
IOP	IOP
MBCU	MBCU
	PIA

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

4.3 Test Invocation

The *io4000* 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 *io4000* test, use the procedure shown in the following figure:

Figure 4–1, Test Invocation Sequence

```
(spu)> cd /mnt/test RETURN
(spu)> sysreset RETURN
(spu)> mminit -s RETURN
(spu)> dshell RETURN
: test io4000 [-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.

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.

Entering only **test io4000** executes all *io4000* 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.

> **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 whether 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*.

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
```

4.3.1 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 ( )):

Figure 4–3, 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

1: Enter IOP numbers [3,5-7][1]                          (5)  ->
2: Do you wish to run the multibus voltage tests?
   [y,n]                                                 (n)  ->
3: Enter OK, or :NN to return to question NN [OK]     (OK) ->
```

[1] The available selections for this prompt will vary depending on which IOP(s) are present.

The prompts and responses appear sequentially on the screen, one line at a time. All the prompts and responses are shown in one figure for convenience. The Test Parameter Menu illustrates *all* questions that can be displayed during test parameter input. 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 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

A description of the meaning of each prompt follows:

```
Enter IOP numbers [3,5-7]                                (3,5-7) ->
```

This prompt allows for the selection of the IOP(s) to be scanned. (For more than one IOP, enter the numbers separated by commas or a range separated by a hyphen, for example 3,4,5 or 3-5). After entering the IOP number(s), each IOP entered is scanned to see if it exists. The test will reprompt for valid IOP(s) if no valid IOP(s) are specified.

```
Do you wish to run the multibus voltage tests?  [y,n]          (n) ->
```

This prompt selects whether Subtest 600, Multibus Voltage Test should be executed. If the default is selected, the Multibus voltage test does not execute.

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

If **OK** or (RETURN) is entered, the test parameter menu terminates and all inputs are no longer changeable.

4.4 Hardware Initialization Sequence

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

- Signals and interrupts for hard and soft errors are set
- Determines whether the IOPs can run cache accelerated mode
- Initializes SPU local test variables

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

4.5 Class Descriptions

The *io4000* test contains the following four classes of subtests as shown in the following table. The individual class descriptions that follow list the time required to execute each subtest.

> **NOTE**
>
> The subtest execution times shown in this test description were executed at 10 Mhz on a system with one IOP and 16 Mbytes of main memory. The execution times are approximate and depend on factors such as memory size and number of IOP(s) under test.

Table 4–2, *io4000* Test Classes

Class	Description
1	68000 subsystem tests
2	RAM pattern tests
3	Cache functionality tests
6	Multibus tests

> **NOTE**
>
> The first time one of the subtests in the 200–600 range is executed, the IOP(s) under test must be loaded with the test program, which adds about 20 seconds to the test times.

4.5.1 Class 1 Subtests

Class 1 subtests verify that the IOP can correctly load a program from main memory. These tests communicate with the SPU via the IOP LED register. The SPU places a command on the LEDs through the IOP scan ring. The IOP then reads the LED register, executes the specified command, and places the result in the LED register.

In general, a return code of zero indicates the test completed without error. When the test fails, a nonzero value is returned, which identifies the step in the subtest that failed. The SPU reads this code and prints the appropriate diagnostic message. Class 1 subtests are listed in the following table:

> **NOTE**
>
> The subtest execution times shown in this section were executed at 10 Mhz on a system with one IOP and 16 Mbytes of main memory. The execution times are approximate and depend on factors such as memory size and number of IOP(s) under test.

Table 4–3, Class 1 Subtests

Subtest	Description	Time (min:sec)
100	IOP Reset	:02
101	IOP Self-test	:40
102	IOP Initialization Command	:01
103	IOP Boot Command	:02

> **NOTE**
>
> Subtest 100 must be run before subtests 101, 102, and 103. Also, subtest 103 cannot be looped.

4.5.1.1 Subtest 100, IOP Reset

Subtest 100 resets all the IOPs being tested.

> **NOTE**
>
> This subtest must be executed before subtests 101, 102, or 103.

Each IOP then performs the following:

- Tests operation of the instructions:
  - beq
  - bne
  - move <ea>,cc
  - cmpi #<data>,<ea> for all operand lengths
  - cmpa <ea>,an
  - cmp<ea>,dn
- Tests uniqueness of registers a0, d0-d1
- Tests functionality of all bits in a0, d0-d1
- Tests ability to calculate a checksum

If all of the previously listed tests pass, then the IOP performs a checksum test on the EPROM. If the checksum test passes, the IOP then enters a loop waiting for commands from the SPU to be passed through the IOP scan ring. If any of the above tests fail, an error signal is sent back to the SPU via the send-error bit in the miscellaneous diagnostic register, and the IOP 68000 performs a double-bus fault and dies.

4.5.1.2 Subtest 101, IOP Self-test

Subtest 101 places a self-test command on the red LEDs of the IOPs under test. Each IOP being tested then performs a series of seven tests (CPU2, RAM1, CPU3, RAM2, MAPTST, CACHETST, and MBLBTST). The following are descriptions of the tests:

CPU2

This test checks register functionality for the various 68000 registers, checks the addressing modes, verifies the jump and branch instructions, performs integer arithmetic tests, data movement operations, shift and rotate instructions, and bit-manipulation operations.

RAM1

This test determines the size of local memory by writing to the first word in each bank until a bus error occurs. It then performs a walking '1's and '0's test on the first word after the ROM and on the first word of all the following banks. Then, on the last 4 Kbytes of installed local memory, it performs a uniqueness test and a true/complement pattern test. 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.

CPU3

This test checks the 68000 instructions not tested in subtest 100 or CPU2 by using the last 4 Kbytes of installed local memory as a stack. It also checks privilege violations and address errors. The *stop*, *trace*, and *reset* instructions are not checked.

RAM2

This test is similar to RAM1 except that it checks local memory from immediately after the ROM to the word preceding the last 4 Kbytes of installed local memory. A parity test is not performed.

MAPTST

This test performs a walking '1's and '0's test on the memory protection register at `0xff8000`. Then, it performs a uniqueness test and a true/complement pattern test for all the 256 registers. Next, it copies ROM to local memory and modifies the protection for installed local memory to valid, read, write, and execute. It clears the registers for uninstalled local memory. Then, memory protection is turned on, and the valid, read, write, and execute bits are verified from supervisor and user state. Finally, at the end of the test, memory protection is again disabled.

CACHETST

This test clears the cache after disabling cache parity checking and clearing the longword or byte dirty flags and the tag flag registers. Then, it reclears the tag flag registers and enables cache parity checking. Any pending cache or PBUS interrupts are cleared, and cache interrupts are enabled. The local memory copy of the EPROM is mapped in, and a checksum is performed on it. Next, the test sets the map registers to point to the last Mbyte of the PBUS. A unique pattern is written to the first 64 shortwords in each main memory 4-Kbyte window. As each word is written, the dirty bits are checked to see that they are updated appropriately. The test then clears the longword or byte dirty flags and marks all tag registers as loaded. Finally, the pattern written to each cache location is verified.

MBLBTST

This test resets the Multibus cable drivers and places them in loopback mode. In each Multibus window, the local memory copy of the EPROM is mapped in, and a checksum test is performed. The pattern left by the cache test is then checked through each of the Multibus windows.

4.5.1.3 Subtest 102, IOP Initialization Command

Subtest 102 determines the size of the installed local memory, and then clears local memory from the word following the EPROM to the end. The test then disables cache parity checking and, for each of the map registers, performs the following:

- Clears the dirty 'a' and 'b' bits
- Clears the corresponding 128 cache bytes
- Clears the dirty 'a' and 'b' bits again

Next, the test enables cache parity checking, and points the first map register to the population configuration map on the MCU. The Population Configuration Map (PCM) is searched in order to locate the first 2 Mbytes of main memory. Once located, the map registers are initialized to point to the first half of this 2 Mbyte block. The motherboard slot number is used to index into a table in main memory. This table contains a pattern, previously initialized by the SPU, that is to be echoed by the IOP. The pattern is the current time, in seconds, logically 'anded' with a mask of `0xffffff00`. When the IOP echoes this pattern, the subtest ends.

4.5.1.4 Subtest 103, IOP Boot Command

> **NOTE**
>
> Before executing Subtest 103, Subtest 102 must have completed successfully. This subtest checks only the SPU *load* command. Also, it is not possible to loop on this subtest.

Initially, 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 IOP jumps to the local memory copy of the EPROM. Memory protection is enabled. The IOP slot number is used as an index into a table in main memory. The table is located in the first Mbyte of installed main memory.

After the test begins, the IOP 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 IOP 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 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.

4.5.2 Class 2 Subtests

Class 2 subtests are listed in the following table:

> **NOTE**
>
> The subtest execution times shown in this section were executed at 10 Mhz on a system with one IOP and 16 Mbytes of main memory. The execution times are approximate and depend on factors such as memory size and number of IOP(s) under test.

Table 4–5, Class 2 Subtests

Subtest	Description	Time (min:sec)
200	PBUS Interrupt	0:09
201	PBUS Test-and-set	0:01
202	IOP Memory Access	0:01

4.5.2.1 Subtest 200, PBUS Interrupt

Subtest 200 verifies the ability of the IOP to request and use the PBUS interrupt bus. The subtest consists of the following three steps:

Step 1: Interrupt Receive Test

In the receive test, all 256 PBUS interrupts are sent by the SPU to the IOP, one at a time. After each interrupt is sent, the IOP verifies that only one interrupt was received and that the interrupt was received by the proper group. The group is determined by taking the modulo 4 residue of the interrupt number.

Step 2: Interrupt Transmit Test

In the interrupt transmit test, the IOP sends the SPU interrupt numbers 8, 9, 10, and 11 (the four interrupts that the SPU is capable of receiving). The IOP 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 IOP in each of the four possible groups. Like the receive test, as each interrupt is sent, the IOP verifies that only one interrupt is received and that the acknowledge is also received.

> **NOTE**
>
> The terms receive and transmit are from the IOP's aspect.

4.5.2.2 Subtest 201, PBUS Test-and-set

Subtest 201 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 non test-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 202, IOP Memory Access

Subtest 202 verifies the IOP's ability to access all of allocated main memory. The test determines which blocks in memory are allocated by reading the PCM on the SPU, memory is allocated in 2 Mbyte blocks. The test writes and then reads a single unique integer value in each allocated block, and the test then re-reads each of the blocks to verify proper addressing.

4.5.3 Class 3 Subtests

Class 3 subtests verify the functionality of the IOP cache memory. The current subtests verify the cache in accelerated mode, normal mode, and in bypass mode.

> **NOTE**
>
> Not all of the subtests in Class 3 are currently implemented. Only the subtests that are currently available are listed in the following table.

Table 4–6, Class 3 Subtests

Subtest	Description	Time (min:sec)
300	IOP Cache Accelerate Read	2:26
301	IOP Cache Accelerate Write	3:42
302	IOP Cache Bypass Read	3:38
303	IOP Cache Bypass Write	3:36

> **NOTE**
>
> The subtest execution times shown in this section were executed at 10 Mhz on a system with one IOP and 16 Mbytes of main memory. The execution times are approximate and depend on factors such as memory size and number of IOP(s) under test.

4.5.3.1 Subtest 300, IOP Cache Accelerate Read

First subtest 300 sets the accelerate bit in the PMAP register. Then it sets the PMAP register that logically follows the window under test to nonaccelerate, nonbypass mode.

The IOP slot number is used to index into a table in main memory that contains patterns and pointers to main memory, which are used to test the accelerate function. The SPU initializes the table before the accelerate test command is given to the IOP(s). The table is accessed through the nonaccelerated PMAP register as described in the previous paragraph. The table has checksums and an IOP ID that are checked to ensure the integrity of the data.

Using the information from the table, the test checks the PMAP window under test. Before any location in the page under test is referenced, an initial 64-byte pattern corresponding to buffer locations `0` to `3f` is initialized to a known pattern through the nonaccelerated window. The pattern used for the first byte in the page is the IOP slot number plus the page number of the page under test, modulo 256. The test initializes each successive byte to the value of the previous byte plus 1, modulo 256.

Then a loop is entered in which each of the 4096 bytes in the page is read and checked for valid data. Additionally, each time byte `0` of each of the two cache buffers is read, the PTAG register for the page is checked to ensure the following:

- The accelerate-on reference bit is set.
- The loaded bit is set.
- The buffer dirty bit is clear.
- The TAG value and its parity are properly set.

Then, if the previous conditions are met, the test uses the nonaccelerated window to negate the current pattern in main memory. The original pattern remains in the cache. The pattern for the next 64-byte group then is initialized through the nonaccelerated window. When byte `1` of a buffer is read, the test checks the following:

- The accelerate-on reference bit is clear.
- The loaded bit is set for both buffers.
- The TAG value for the current buffer has not changed, and the TAG value for the other buffer is properly set.

4.5.3.2 Subtest 301, IOP Cache Accelerate Write

Subtest 301 initializes the main memory area used for the write test to the negation of the pattern to be written there. The test then uses the sum of the IOP slot number plus the page number of the page under test, modulo 256, as the pattern for the first byte. Next, it enters a loop into which each of the 4096 bytes in the page is written, one byte at a time. As each byte is written, the dirty bits for that longword are checked to make sure that they are set to the expected value. The test then checks the PTAG register for valid data. When byte `0x3f` of each buffer is written, the corresponding area of main memory is read through the nonaccelerated window to ensure that the pattern has *not* yet been written.

Every time byte `0` of a cache buffer is written, beginning with byte `0x40` in the page under test, the PTAG register is checked to verify that the accelerate-on reference bit is set. When byte `1` of a buffer is written, the nonaccelerated window reads main memory to verify that the pattern for

the previous 64-byte block is written to memory and to verify that the accelerate-on reference bit resets.

After the last 64-byte block is written, the window under test is flushed so that all data is written to memory. Finally, the test again checks the pattern in the entire 4096-byte block for integrity through the nonaccelerated window.

4.5.3.3 Subtest 302, IOP Cache Bypass Read

Subtest 302 tests each of the windows in the IOP cache. As in the accelerate read test, this test places the window that logically follows the window under test in nonaccelerate, nonbypass mode and verifies the operation of the window under test. Then it initializes the main memory area for the bypass test to the negation of the pattern to be written there. It uses the same patterns as those listed for Subtest 300, IOP Cache Accelerate Read Test.

For each of the 4096 bytes in the window, the test first uses the nonaccelerate window to write a one byte pattern to main memory. Then it reads the byte just written through the window under test and verifies the data. Finally it negates the byte in main memory through the nonbypass window.

4.5.3.4 Subtest 303, IOP Cache Bypass Write

Subtest 303 , the logical complement of Subtest 302, tests each window in the IOP cache. It initializes the main memory area used to test the cache to the negation of the pattern that is to be written there. It uses the same pattern described in Subtest 301, IOP Cache Accelerate Write Test.

For each of the 4096 bytes in the window, the test first uses the window under test to write a 1-byte pattern to main memory. Then it uses the nonbypassed window to verify that the pattern was written to main memory.

4.5.4 Class 6 Subtests

Class 6 subtests test the operation of portions of the Multibus Control Units (MBCUs) that are installed on the IOP(s) being tested. The system I/O configuration file, */ioconfig*, is read to determine what MBCUs are present.

> **NOTE**
>
> Not all of the subtests in this class are currently implemented. The subtest that is currently available is listed in the following table.

Table 4–7, Class 6 Subtest

Subtest	Description	Time (min:sec)
600	Multibus Voltages	:01

NOTE

The subtest execution times shown in this section were executed at 10 Mhz on a system with one IOP and 16 Mbytes of main memory. The execution times are approximate and depend on factors such as memory size and number of IOP(s) under test.

Subtest 600 reads and tests the four voltages of each Multibus that is present for out-of-tolerance conditions. The voltages and tolerances for a Multibus are listed in the following table:

Table 4–8, Multibus Voltages and Tolerances

Voltage	Minimum	Tolerance
+12.10	+11.40	+12.60
+05.10	+04.75	+05.25
-05.10	-05.75	-04.75
-12.10	-12.60	-11.40

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

R

S

T

U

V

W

CONVEX Multibus I/O Subsystem
(*io4000*) Diagnostics Manual
Document No. 760-001830-000
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)