CONVEX VMEbus I/O Processor (*io5000*) Diagnostics Manual

Document No. 760-002930-000

First Edition
May 1991

CONVEX Computer Corporation
Richardson, Texas USA

CONVEX VMEbus I/O Processor (io5000) Diagnostics Manual
Order No. DHW-241
First Edition

© 1991 CONVEX Computer Corporation
All rights reserved.

This document is copyrighted. All rights reserved. This document may not, in whole or part, be copied, duplicated, reproduced, translated, electronically stored or reduced to machine readable form without prior written consent from CONVEX Computer Corporation (CONVEX).

Although the material contained herein has been carefully reviewed, CONVEX does not warrant it to be free of errors or omissions. CONVEX reserves the right to make corrections, updates, revisions, or changes to the information contained herein. CONVEX does not warrant the material described herein to be free of patent infringement.

UNLESS PROVIDED OTHERWISE IN WRITING WITH CONVEX COMPUTER CORPORATION (CONVEX), THE EQUIPMENT DESCRIBED HEREIN IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES. THE ABOVE EXCLUSION MAY NOT BE APPLICABLE TO ALL PURCHASERS BECAUSE WARRANTY RIGHTS CAN VARY FROM STATE TO STATE. IN NO EVENT WILL CONVEX BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING ANY LOST PROFITS OR LOST SAVINGS, ARISING OUT OF THE USE OR INABILITY TO USE THIS EQUIPMENT. CONVEX WILL NOT BE LIABLE EVEN IF IT HAS BEEN NOTIFIED OF THE POSSIBILITY OF SUCH DAMAGE BY THE PURCHASER OR ANY THIRD PARTY.

CONVEX and the CONVEX logo ("C") are registered trademarks of CONVEX Computer Corporation
C1, C120, C201, C202, C210, C220, C230 and C240 are trademarks of CONVEX Computer Corporation
C100 Series and C200 Series are trademarks of CONVEX Computer Corporation
UNIX is a registered trademark of AT&T Bell Laboratories
ConvexOS is a registered trademark of CONVEX Computer Corporation

Printed in the United States of America

Revision Sheet

CONVEX VMEbus I/O Processor (io5000) Diagnostics Manual

Edition	Document No.	Date	Description
First	760-002930-000	May 1991	First release. Contains the *io5000* diagnostic test information from the *CONVEX PBUS I/O Systems Diagnostics Manual.*

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

1 Diagnostics Environment

1.1 Overview 1-1
1.2 Test Program Naming Conventions 1-1
1.2.1 Test Program Categories 1-1
1.2.2 Test Program Types 1-2
1.2.3 Test Program Device Types 1-2
1.2.4 Examples of Test Program Names 1-3

2 EGOS Overview

2.1 Overview 2-1
2.2 Purpose of EGOS for Diagnostic Testing 2-1
2.3 EGOS for the Multibus Interface 2-1
2.4 EGOS for HSP Interface, HSP EGOS 2-1
2.5 EGOS for VME Interface, VIOP EGOS 2-2
2.6 EGOS Position in the Environment 2-2

3 Dshell Overview

3.1 Overview 3-1
3.2 Diagnostic Shell (*dshell*) Overview 3-1
3.3 Syntax Help for *dshell* Commands 3-3

4 VMEbus I/O Processor Test (*io5000*)

4.1 Overview 4-1
4.2 Prerequisites and Required Equipment 4-1
4.3 Test Invocation 4-1
4.3.1 Test Parameter Menu 4-3
4.3.2 Prompt Explanations 4-4
4.4 Hardware Initialization Sequence 4-4
4.5 Class Descriptions 4-5
4.5.1 Class 1 Subtests 4-5
4.5.1.1 Subtest 100, VIOP Reset 4-6
4.5.1.2 Subtest 101, VIOP Self-test 4-6
4.5.1.3 Subtest 102, VIOP Initialization Command 4-8
4.5.1.4 Subtest 103, VIOP Boot Command 4-8
4.5.2 Class 2 Subtests 4-9
4.5.2.1 Subtest 200, PBUS Communication 4-10
4.5.2.2 Subtest 220, PBUS Test-and-set 4-10
4.5.2.3 Subtest 221, PBUS Test-and-clear 4-10
4.5.2.4 Subtest 230, Line Clock Interrupt 4-10
4.5.2.5 Subtest 250, VIOP Microprocessor Clock Margin 4-11
4.5.2.6 Subtest 251, VIOP Cache Buffer Tag 4-11
4.5.2.7 Subtest 261, Parity Checker 4-11
4.5.3 Class 3 Subtests 4-12
4.5.3.1 Subtest 300, VIOP Cache Accelerate Read 4-12
4.5.3.2 Subtest 301, VIOP Cache Accelerate Write 4-13
4.5.3.3 Subtest 302, VIOP Cache Bypass Read 4-13
4.5.3.4 Subtest 303, VIOP Cache Bypass Write 4-14
4.5.3.5 Subtest 304, VIOP Cache Protection 4-14
4.5.3.6 Subtest 310, VIOP Cache Test of Dirty Bytes in Longwords 4-14

4.5.3.7 Subtest 311, VIOP Cache Test of Dirty Longwords in a Buffer 4-14
4.5.3.8 Subtest 312, VIOP Cache Test of Dirty Buffers in a Page 4-14
4.5.4 Class 4 Subtests 4-15
4.5.4.1 Subtest 400, PBUS Interrupt 4-15
4.5.5 Class 5 Subtests 4-15
4.5.5.1 Subtest 500, VBCU Cable Pattern 4-16
4.5.5.2 Subtest 501, VBCU Forced Interrupt 4-16
4.5.6 Class 6 Subtests 4-16
4.5.6.1 Subtest 600, VMEbus Voltage 4-16
4.5.6.2 Subtest 601, VME Chassis Power Supply Margin 4-17

Appendixes

A Reporting Problems
A.1 Overview A-1
A.2 Technical Assistance Center A-1
A.3 The *contact* Utility A-1
A.4 Prerequisites A-1
A.4.1 UUCP Connection A-1
A.4.2 Finding the Program Path Name A-2
A.4.3 Finding the Program Version Number A-2
A.5 Tips on Using the *contact* Utility A-2
A.5.1 Using a *.contact* File A-3
A.5.2 Aborting the Report A-3
A.5.3 Submitting the *dead.report* File A-3
A.5.4 Suspending a Report A-3
A.5.5 Ending a Response A-3
A.5.6 Tilde-Escape Sequences A-4
A.6 Using the *contact* Utility A-4

List of Tables

1–1 Test Program Categories 1-2
1–2 Test Program Types 1-2
1–3 Test Program Device Types 1-3
1–4 Example Test Program Names 1-3
3–1 *dshell* Commands 3-2
4–1 Hardware Requirements 4-1
4–2 *io5000* Test Classes 4-5
4–3 Class 1 Subtests 4-6
4–4 Valid SPU Commands for Subtest 103 4-9
4–5 Class 2 Subtests 4-9
4–6 Class 3 Subtests 4-12
4–7 Class 4 Subtests 4-15
4–8 Class 5 Subtests 4-16
4–9 Class 6 Subtests 4-16
4–10 VMEbus Voltages and Tolerances 4-16

List of Figures

2–1 EGOS' Position in the Environment 2-3
3–1 Syntax Help for the *loop* Command 3-3
4–1 Test Invocation Sequence 4-2
4–2 Alternate Test Invocation Sequence 4-3
4–3 Test Parameter Menu 4-3

THIS PAGE INTENTIONALLY LEFT BLANK

Preface

Purpose and Intended Audience

This manual explains how to run the *io5000* diagnostic, which verifies the functionality of a specified VIOP. This document is not a tutorial, but rather a reference for the users of the *io5000* 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 *io5000* 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. VMEbus I/O Processor Test (*io5000*)**—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-241.
The document number for this manual is 760-002930-000.

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

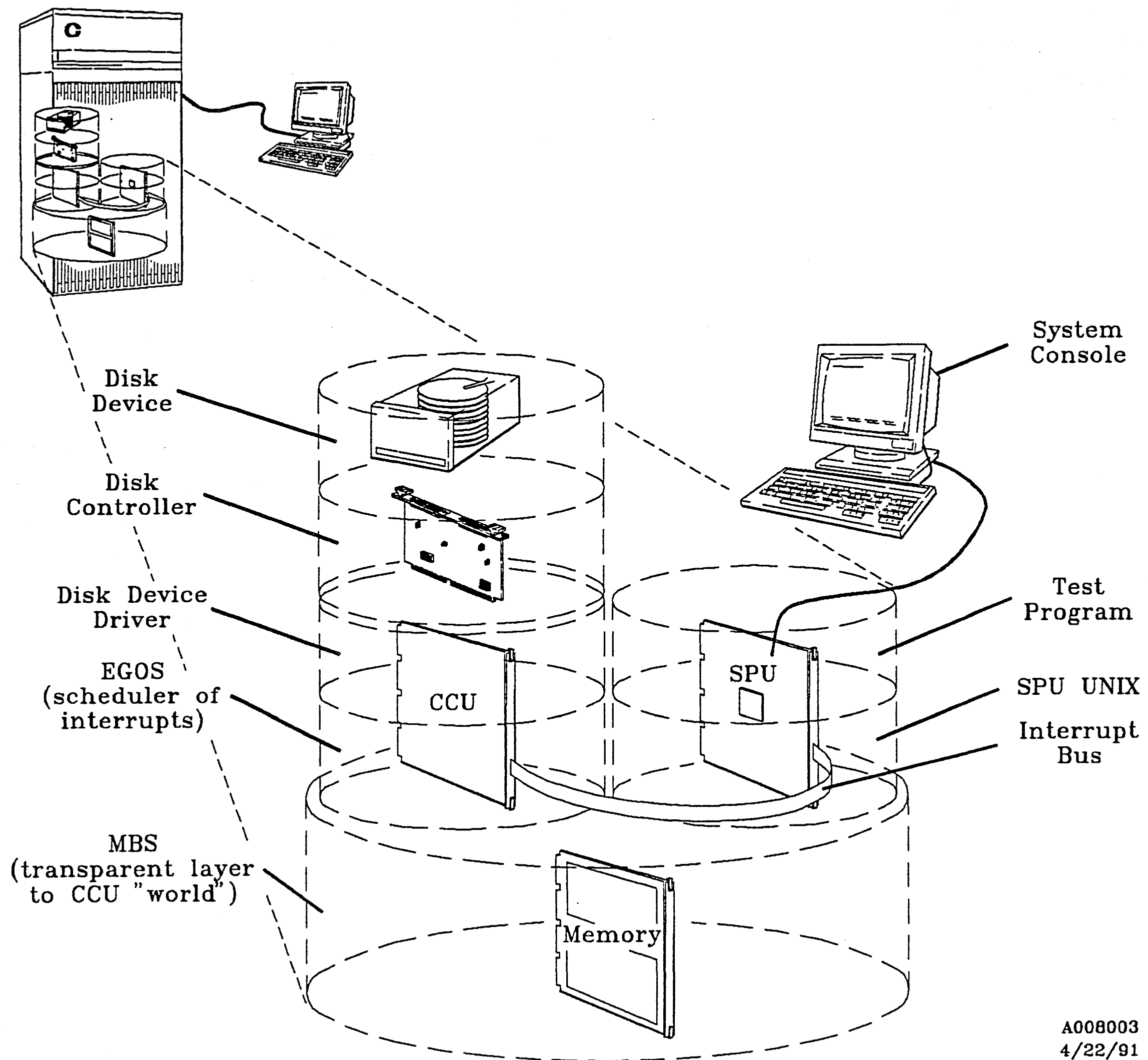
The VME interface version of EGOS is designed with a scheduler for the VIOP and is called VIOP EGOS. VIOP EGOS supports event driven device drivers as well as process type device drivers. VIOP EGOS utilizes a *sleep/wakeup* type of process control that improves efficiency of the device driver and makes 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
VMEbus I/O Processor Test (*io5000*)

4.1 Overview

The *io5000* test verifies the functionality of a specified VIOP. The test verifies the following on the VIOP(s) specified:

- 68020 on the VIOP can correctly execute instructions
- VIOP local memory is functional
- Memory protection and mapping registers are operational
- Cache memory can be written to and read from, and the associated cache control bits are functional
- Two VMEbus cable interfaces operate in loopback mode
- VIOP can boot a program from memory
- Voltages on all installed VMEbus chassis are within tolerance

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
VIOP	VIOP
VBCU	VBCU
	PIA

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

4.3 Test Invocation

The *io5000* 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 *io5000* 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 io5000 [-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 io5000** executes all *io5000* 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 io5000 [-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 ( )). 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.

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.

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 VIOP numbers [0-3]1                       (1)  ->
  2: Enter value for Disable_68k_cache flag [0,1]    (0)  ->
  3: Enter OK, or :NN to return to question NN [OK]  (OK) ->
```

[1] The possible selections for this prompt can vary depending on which VIOP(s) are present.

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 VIOP numbers [0-3]                    (1) ->
```

This prompt allows selection of the VIOP(s) to be scanned. For more than one VIOP, enter the numbers separated by commas or a range separated by a hyphen, for example 0,1,2 or 0-3.) After entering the VIOP number(s), each VIOP entered is scanned to see if it exists. The test will reprompt for valid VIOP(s) if no valid IOP(s) are specified.

```
Enter value for Disable_68k_cache flag [0,1]          (0) ->
```

This prompt allows modification of the setup of the VIOP scan ring bit that can force the 68020 to run with its internal instruction cache disabled. Disabling the 68020 cache forces a memory reference for each instruction fetch.

```
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:

- SPU windows to main memory are initialized
- SPU local test variables are initialized
- The SPU verifies that the system has been reset since power-up

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

4.5 Class Descriptions

The *io5000* contains the following six classes of subtests:

Table 4–2, *io5000* Test Classes

CLASS	DESCRIPTION
1	68020 subsystem tests
2	VIOP miscellaneous tests
3	Cache functionality tests
4	Interrupt tests
5	VBCU interface tests
6	VMEbus voltage tests

NOTE

The first time one of the subtests in the 200 through 601 range is executed, the VIOP(s) under test must be loaded with the test program, which adds about 10 seconds to the test times.

4.5.1 Class 1 Subtests

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

NOTE

There are restrictions on the execution sequence of these subtests:

- Subtest 100 must be the first subtest executed in this class.
- 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 sequential order.

Table 4–3, Class 1 Subtests

SUBTEST	DESCRIPTION	TIME (min:sec)
100	VIOP Reset	:02
101	VIOP Self-test	:31
102	VIOP Initialization Command	:02
103	VIOP Boot Command	:02

4.5.1.1 Subtest 100, VIOP Reset

Subtest 100 resets the VIOP via the scan ring. This causes the VIOP to execute the initialization code in its EPROM. The initialization code saves the state of the VIOP as found on reset and verifies that the VIOP's 68020 can calculate a checksum. When verified, the VIOP calculates a checksum of the EPROM. If these tests pass, the VIOP enters a loop and waits for an EPROM command to be placed on the LED register. If this test fails, the VIOP informs the SPU by asserting a hard error.

4.5.1.2 Subtest 101, VIOP Self-test

Subtest 101 is the VIOP EPROM based self-test. It consists of 11 steps that verify the VIOP's ability to load a program from main memory. The SPU commands the VIOP to do this test by placing a code in the VIOP's LED registers via the scan ring. When this code is detected, the VIOP executes the steps listed below. If a failure is detected, the VIOP returns the step number and other relevant data via the LED register and auxiliary test result register in the scan ring.

Step 1: Pattern Test the Auxiliary Test Results Register

Step 1 verifies that the Auxiliary Test Results Register (AUXTRR) is usable for reporting failure information back to the SPU in the event of a failure of one of the later steps.

Step 2: Verify Instructions Needed for Memory Test

Step 2 performs the following:

- Test VIOP 68020 registers
- Check addressing modes
- Verify jump and branch instructions
- The following operations are executed:
  - Integer arithmetic tests
  - Data movement operations
  - Shift and rotate instructions
  - Bit manipulation

Step 3: Initial Memory Operations Test

Step 3 determines the size of memory by writing to the first word in each bank until a bus error occurs. It performs a walking '1's and '0's test on the first word after the EPROM, and on the first word of the following banks. Then, on the last 4 Kbytes of installed memory, it does 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.

Step 4: Verify Remaining 68020 Instructions

Step 4 uses the last 4 Kbytes of installed memory as a stack, and checks the 68020 instructions not tested in Subtest 100, VIOP Reset Test or in step 2 of this subtest. The *stop*, *trace*, and *reset* instructions are not checked. It also checks privilege violations and address errors.

Step 5: Test Remaining Memory

Step 5 is similar to step 3, but it checks memory from immediately after the EPROM to the word preceding the last 4 Kbytes of installed memory. No parity testing is done.

Step 6: Verify VIOP Local Memory Mapper

Step 6 starts by doing a walking '1's and '0's test on the memory protection register at `0xff8000`. For all 256 registers, it does a uniqueness test and a true/complement pattern test. The test copies the VIOP's EPROM to its local RAM, modifies the protection for installed memory to valid, read, write, and execute. It clears the registers for uninstalled memory. Memory protection is turned on, and the valid, read, write, and execute bits are verified from the supervisor state and the user state. At the end of the test, memory protection is disabled.

Step 7: PMAP Register Test

The RAMs that store the PMAP register data are checked by performing column functionality, uniqueness, and bit functionality tests. Bad parity detection is then checked. This is followed by a functionality test of the valid bit in the PMAP register.

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 step with a VIOP that never started executing. The value of 8 is used by the SPU to command the VIOP to perform its self-test. As the VIOP progresses in its test, the value is incremented. By skipping step 8, the SPU can determine if the VIOP started execution of the self test.

Step 9: VIOP Cache Test

Step 9 clears the cache after disabling cache parity checking and clearing the longword/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/byte dirty flags and marks all tag registers as loaded. Finally, the pattern written to each cache location is verified.

Step 10: VME Cable Interface Loopback Test

Step 10 resets the VMEbus cable drivers and places them in loopback mode. A local memory copy of the EPROM is mapped in through each cable interface and a checksum test is performed. Then the pattern left by the cache test is checked through each of the VIOP cache windows.

Step 11: PBUS Access Test

Step 11 performs the PBUS access test which reads, verifies, and echos a pattern placed in main memory by the SPU. This test verifies that the SPU and the VIOP agree on the location of main memory and of the proper byte ordering in that memory. The first access to the PBUS made by the VIOP is to read the Population Configuration Map (PCM). This is needed to locate the test pattern in main memory.

4.5.1.3 Subtest 102, VIOP 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. Subtest 102 performs the following for each of the map registers:

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

Next, for C100 Series machines, the test enables cache parity checking, and points the first map register to the Population Configuration Map (PCM) on the MCU. The PCM is searched 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. For C200 Series machines, the test reads the Memory Base Pointer (MBP) to determine the address of the first block of memory.

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 VIOP. The pattern is the current time, in seconds, logically 'anded' with a mask of `0xffffff00`. When the VIOP echoes this pattern, the subtest ends.

4.5.1.4 Subtest 103, VIOP Boot Command

> **NOTE**
>
> Subtest 102 must have completed successfully before executing Subtest 103. 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 32 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 VIOP jumps to the local memory copy of the EPROM. Memory protection is enabled. The VIOP 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 VIOP enters a loop waiting for the SPU to place commands in main memory by the SPU. 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 VIOP 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.

NOTE

This subtest checks only the *load* command.

4.5.2 Class 2 Subtests

Class 2 subtests verify various parts of the VIOP that do not logically fall under any of the other classes. These subtests verify the next level of functionality past the EPROM based Class 1 subtests.

Table 4–5, Class 2 Subtests

SUBTEST	DESCRIPTION	TIME (min:sec)
200	PBUS Communication	:16
220	PBUS Test-and-set	:02
221[1]	PBUS Test-and-clear	:02
230	Line Clock Interrupt	:10
250	VIOP Microprocessor Clock Margin	:18
251	VIOP Cache Buffer Tag	:52
261	Parity Checker	:01

[1] This subtest is only available on C200 Series machines.

4.5.2.1 Subtest 200, PBUS Communication

After the VIOP has been booted, Subtest 200 tests the ability of the SPU to communicate with the program running on the VIOP via main memory by sending 1,000 no-op commands to the VIOP. The SPU puts a command number in the communication structure in main memory and waits for the VIOP to write into the structure indicating completion of the command. In this case the VIOP interprets the command as a no-op and immediately signals that the command has been completed.

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 221, PBUS Test-and-clear

> **NOTE**
>
> This test is only available for C200 Series machines and will not execute on a C1 or C120 machine.

Subtest 221 verifies that all PMAP registers operate in test-and-clear mode, by placing each PMAP register in test-and-clear mode in sequence. The PMAP register that logically follows the register under test is prepared for nontest-and-clear operation and is used to verify the operation.

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

4.5.2.4 Subtest 230, Line Clock Interrupt

Subtest 230 enables the line clock interrupt on the VIOP and waits for 10 seconds worth of line clock interrupts to occur. The SPU UNIX time of day is checked to ensure that the interrupts took 10 seconds to occur.

4.5.2.5 Subtest 250, VIOP Microprocessor Clock Margin

Subtest 250 tests the ability of the SPU to select the clock frequency for the VIOP microprocessor. The VIOP has two clock frequencies that can be selected to run the VIOP 68020 microprocessor.

The frequencies are generated by two separate clock oscillators. The nominal clock frequency is 20 MHz, and the alternate, or margin, clock frequency is usually 22 MHz. The alternate clock frequency is derived from a socketed oscillator which is divided by two to produce the alternate clock frequency. The socketed oscillator may be replaced with any value between 25 MHz and 44 MHz.

The test first sets all selected VIOPs to run from its standard or nominal clock frequency. The VIOPs are rebooted because selecting the clock frequency requires resetting the VIOP. The VIOP is then commanded in a timed loop to calculate the frequency of the VIOP 68020 processor clock. The test uses the VIOP line clock interrupt as a time base to calculate the microprocessor clock speed. The VIOP reports to the SPU the timed count value as well as the state of the clock select bit in the VIOP Miscellaneous Diagnostic Register.

The SPU then calculates the MHz value of the VIOP microprocessor clock and prints the value on the screen. The SPU also checks the state of the reported VIOP clock state select bit against what it should be. After nominal clocks are verified, each selected VIOP is set up to run from its margin clock oscillator, and the test sequence is repeated. On exiting from this subtest, the clock selection state for each VIOP is restored to its original state.

4.5.2.6 Subtest 251, VIOP Cache Buffer Tag

Subtest 251 verifies the VIOP cache buffer tags. Various cache accesses are performed to cause all the buffer tag RAMs to be tested for address uniqueness. As each tag is set, it is checked to verify that the correct tag is set and that no other tag is affected.

4.5.2.7 Subtest 261, Parity Checker

Subtest 261 verifies the ability of the VIOP to force various parity errors and recover from those errors. The Miscellaneous Diagnostic Register contains the following three special control bits that allow specific parity errors to be generated:

- Force Tag Parity Error Bit
- Force Address Parity Error Bit
- Force Cache Parity Error Bit

The subtest sets the Force Tag Parity Error Bit and verifies that accessing a cache window causes a 68020 bus error and a cache error interrupt. The Cache Error Log is also checked to see that the Force Tag Parity Error Bit is set. The error is cleared and another window access is performed to verify that all the cache errors are cleared.

The same technique is used to test the Force Address Parity Error Bit. The subtest verifies that cache error is generated and the Cache Error Log register reports an address parity error.

The Force Cache Parity Error bit causes a bad PBUS header to be generated and causes a PBUS bus error. The subtest sets this Force Cache Parity Error bit, verifies that a PBUS error interrupt is generated, and checks that the PBUS Error Log reports a PBUS bus error.

4.5.3 Class 3 Subtests

Class 3 subtests verify the functionality of the VIOP cache memory. The current subtests verify the cache in accelerated mode, normal mode, and in bypass mode. Also included is a test of the VIOP cache protection features with respect to VME controller accesses.

Table 4–6, Class 3 Subtests

SUBTEST	DESCRIPTION	TIME (min:sec)
300	VIOP Cache Accelerate Read	1:48
301	VIOP Cache Accelerate Write	2:46
302	VIOP Cache Bypass Read	4:04
303	VIOP Cache Bypass Write	3:50
304	VIOP Cache Protection	:30
310	VIOP Cache Test of Dirty Bytes in Longwords	:37
311	VIOP Cache Test of Dirty Longwords in a Buffer	1:15
312	VIOP Cache Test of Dirty Buffers in a Page	:36

4.5.3.1 Subtest 300, VIOP 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 the nonaccelerate, nonbypass mode.

The VIOP 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 VIOP(s). The table is accessed through the nonaccelerated PMAP register as described in the previous paragraph. The table has checksums and a VIOP identification 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 VIOP 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 4,096 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:

- Accelerate-on reference bit is set
- Loaded bit is set
- Buffer dirty bit is clear
- 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:

- Accelerate-on reference bit is clear
- Loaded bit is set for both buffers
- 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, VIOP 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 VIOP 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 4,096 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 4,096 byte block for integrity through the nonaccelerated window.

4.5.3.3 Subtest 302, VIOP Cache Bypass Read

Subtest 302 checks each of the windows in the VIOP cache. As in the accelerate read test, this test places the window that logically follows the window under test in the nonaccelerate, non-bypass 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, VIOP Cache Accelerate Read.

For each 4,096 bytes in the window, the test first uses the nonaccelerate window to write a 1 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, VIOP Cache Bypass Write

Subtest 303, the logical complement of Subtest 302, tests each window in the VIOP 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 the cache accelerate tests.

For each of the 4,096 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.

4.5.3.5 Subtest 304, VIOP Cache Protection

This subtest tests access protection of the VIOP cache. The 68020 on the VIOP simulates VME controller access into the VIOP cache by placing the cable interfaces into loopback mode and making accesses into VME address space. The VIOP cache protection bits in the PMAP registers determine which VME controllers can access a particular cache window. The subtest tests all of the available protection modes of each cache window for all possible VME controller accesses.

4.5.3.6 Subtest 310, VIOP Cache Test of Dirty Bytes in Longwords

Subtest 310 verifies that the VIOP cache can correctly update main memory from its cache windows for all possible combinations of dirty bytes in a cache longword.

4.5.3.7 Subtest 311, VIOP Cache Test of Dirty Longwords in a Buffer

Subtest 311 verifies that the VIOP cache can correctly update main memory from its cache windows for all possible combinations of dirty longwords in a cache buffer.

4.5.3.8 Subtest 312, VIOP Cache Test of Dirty Buffers in a Page

Subtest 312 verifies that the VIOP cache can correctly update main memory from its cache windows for all possible combinations of dirty buffers in a cache page.

4.5.4 Class 4 Subtests

Class 4 includes only the PBUS interrupt test. Other interrupts are tested in various other tests, such as the Subtest 101, VIOP Self-test.

Table 4–7, Class 4 Subtests

SUBTEST	DESCRIPTION	TIME (min:sec)
400	PBUS Interrupt	:02

4.5.4.1 Subtest 400, PBUS Interrupt

Subtest 400 verifies the ability of the VIOP 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 VIOP, one at a time. After each interrupt is sent, the VIOP 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 VIOP sends the SPU interrupt numbers 8, 9, 10, and 11, the four interrupts that the SPU is capable of receiving. The VIOP 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 VIOP in each of the four possible groups. Like the receive test, as each interrupt is sent, the VIOP verifies that only one interrupt is received and that the acknowledge is also received.

NOTE

The terms receive and transmit are from the VIOP's point of view.

4.5.5 Class 5 Subtests

Class 5 subtests verify the ability of the 68020 to do accesses to the VMEbus Control Unit (VBCU). VMEbus cable interfaces, interconnecting cables, and the VBCU itself are tested for address and data line functionality. Also the VBCU-forced interrupt capability is exercised.

Table 4–8, Class 5 Subtests

SUBTEST	DESCRIPTION	TIME (min:sec)
500	VBCU Cable Pattern	:01
501	VBCU Forced Interrupt	:01

4.5.5.1 Subtest 500, VBCU Cable Pattern

Subtest 500 verifies continuity of the VBCU cable address and data lines. The VBCU is set into pattern test mode and accesses are performed to test the cable with a walking '1's and '0's pattern. The VBCU address save register is used to read back the last VME address accessed to verify the address lines.

4.5.5.2 Subtest 501, VBCU Forced Interrupt

Subtest 501 uses the VBCU Forced Interrupt Register to test the ability of the VBCU to interrupt the 68020. The subtest forces all eight available interrupts and checks that the correct interrupt vector is received for each one for interrupt levels 1 and 3.

4.5.6 Class 6 Subtests

Class 6 subtests uses the VBCU analog to digital converters to measure the VME chassis power supplies. The supplies are tested in normal mode and the +5 supply is also tested under margin conditions.

Table 4–9, Class 6 Subtests

SUBTEST	DESCRIPTION	TIME (min:sec)
600	VMEbus Voltage	:01
601	VME Chassis Power Supply Margin	:01

4.5.6.1 Subtest 600, VMEbus Voltage

Subtest 600 reads and checks the four voltages in each of the VMEbuses for out-of-tolerance conditions. The voltages and tolerances for a VMEbus are listed in the following table:

Table 4–10, VMEbus Voltages and Tolerances

VOLTAGE	MINIMUM TOLERANCE	MAXIMUM TOLERANCE
+12.10	+11.40	+12.60
+05.10	+04.75	+05.25
-12.10	-12.60	-11.40

4.5.6.2 Subtest 601, VME Chassis Power Supply Margin

Subtest 601 performs the same voltage test as Subtest 600 on the VME chassis +5 volt supply for upper and lower voltage margins. On some VME chassis, the margin capability is disabled by a jumper on the VME backplane. In this case, the subtest prints the message:

```
Subtest 600 VME 0 not tested - margining disabled
```

THIS PAGE INTENTIONALLY LEFT BLANK

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

68020 subsystem tests 4-5

A

Accelerate write 4-13
Alaska, reporting problems from, telephone number for xii
Associated documents, how to order xii
Associated documents, listed xi

B

Bypass read 4-13
Bypass write 4-14

C

C Programming Language xi
Cache accelerate Read 4-12
Cache, buffer tag 4-11
Cache functionality tests 4-12
Canada, reporting problems from, telephone number for xii
cattypedevnn.suffix 1-1
Cautions, described xi
Classes, of subtests, described 4-5
Command scripts, user-created 3-1
contact, aborting the report A-3, A-6
contact, editing the report A-6
contact, ending a response A-3
contact, ending the report A-6
.contact file, skipping first prompt by using A-3
contact, including files in your report A-5
contact, invoking A-1, A-4
contact, prerequisites A-1
contact, prompts A-4
contact, prompts, step-by-step discussion of A-4
contact, report, suspending A-3
contact, reporting problems A-1
contact, restrictions, on tilde-escape sequences A-5
contact, reviewing the report A-6
contact, skipping first prompt by using a *.contact* file A-3
contact, submitting *dead.report* file A-3
contact, submitting the report A-6
contact, tilde-escape sequences A-4
contact, tips on using A-2
CONVEX, address, for ordering documents xii
CONVEX Diagnostic Utilities Manual, C120 xi
CONVEX Diagnostic Utilities Manual, (C200 Series) xi
CONVEX Processor Operation Guide xi
CONVEX UNIX Tutorial Papers xi
CPU 1-1
CPU, *cpu*, test program for 1-2
cpu, test category 1-2

D

dead.report file, submitting A-3
dead.report file, using *-r* option to submit A-3
dev, test category 1-2
Devices, *dev* for 1-1
Devices, test programs for, table 1-3
Devices, types, listed 1-2
Diagnostic environment, overview 1-1
Diagnostic shell. *See dshell*
Diagnostics, selecting 3-1
Disks 1-2
Disks, device, test program for 1-3
dshell, introduction 3-1
dshell, overview 3-1

E

Error messages, selecting 3-1
error reporting A-1

F

Files, test outputs to 3-1

H

Hawaii, reporting problems from, telephone number for xii

I

Interrupt tests 4-15
I/O, subsystem test, *io* for 1-2
I/O system, test program categories for 1-1
io, test category 1-2
io5000, overview 4-1

K

Kernel, hardware tests 1-2
Kernel, hardware tests, program for 1-3

L

Line clock interrupt, Subtest 230 4-10

M

MAU. *See* Memory Array Unit
MBCU. *See* Multibus Control Unit
mem, test category 1-2
Memory Array Unit 4-1
Memory, subsystem test, *mem* for 1-2
Memory system, test program name for 1-1
Multibus Control Unit 4-1

N

Networks 1-2
Networks, device, test program for 1-3
Notational conventions, discussed xi
Notes, described xi

O

Offline tests 1-2
Offline tests, functional, program for 1-3
Online tests 1-2
Online tests, functional, program for 1-3
Overview, diagnostic environment 1-1
Overview, *dshell* 3-1
Overview, VMEbus I/O Processor test 4-1

P

Parity checker test 4-11
PBUS, communication test 4-10
PBUS interrupt, Subtest 400 4-15
PBUS Test-and-clear 4-10
PBUS, Test-and-set 4-10
Peripheral devices, test program name for 1-1

Peripherals, *dev*, test program for 1-2
Printers 1-2
Printers, device, test program for 1-3
problems, reporting, overview A-1

R

Reader's Forum xii
Reporting problems xii
Revision sheet 3

S

Screens, test outputs to 3-1
Scripts, predefined 3-1
Self-tests 1-2
Self-tests, test program for 1-3
Service Processor Unit. *See* SPU
SP2, subsystem test, *spu* for 1-2
SP2, *.t* programs and 1-1
SP2, test program name for 1-1
SPU 4-1
SPU, *dshell* and, introduction 3-1
spu, test category 1-2
Standalone tests 1-2
Subsystems, *cat* for 1-1
Subtest 100 4-6
Subtest 230, Line clock interrupt 4-10
Subtest 250, VIOP microprocessor clock margin 4-11
Subtest 251, VIOP cache buffer tag 4-11
Subtest 261, parity checker 4-11
Subtest 310 4-14
Subtest 311 4-14
Subtest 312 4-14
Subtest 400, PBUS interrupt 4-15
Subtest 500 VBCU cable pattern 4-16
Subtest 501 4-16
Subtest 601 4-17
Subtests, descriptions 4-5

T

.t 1-1
TAC, reporting problems to xii
TAC (Technical Assistance Center), problems, reporting to A-1
Tape units 1-2
Tape units, test program for 1-3
Technical Assistance Center (TAC), problems, reporting to A-1
Technical assistance, discussed xii
Terminals 1-2
Terminals, test program for 1-3
Test programs, categories 1-1
Test programs, categories, table 1-2
Test programs, device types 1-2
Test programs, naming conventions 1-1
Test programs, types 1-2
Test programs, types, table 1-2, 1-3
Tests, options, selecting 3-1
Tests, output, selecting 3-1
tilde-escape sequences A-4
tilde-escape sequences, restrictions on use A-5
Trouble reports xii
trouble reports A-1

U

UNIX-to-UNIX Communication Protocol A-1
UNIX-to-UNIX copy command, *uucp* A-1
UUCP, connection to TAC A-1
uucp, UNIX-to-UNIX copy command A-1

V

VBCU interface tests 4-15
vers, program version number found by using A-2
VIOP Boot Command 4-8
VIOP, initialization command 4-8
VIOP, miscellaneous tests 4-9
VIOP, overview 4-1
VIOP, requirements for running 4-1
VIOP Reset 4-6
VIOP self-test 4-6
VIOP, subtests, classes of, described 4-5
VIOP test program invocation 4-2
VME I/O Processor. *See* VIOP
VMEbus Control Unit. *See* VBCU
VMEbus voltage 4-16
VMEbus voltage tests 4-16
Voltage tests, VMEbus 4-16

W

Warnings, described xi
whence, program path name found by using A-2
which, program path name found by using A-2

CONVEX VMEbus I/O Processor (*io5000*) Diagnostics Manual
Document No. 760-002930-000
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.____

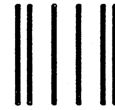
FOR ADDITIONAL INFORMATION OR DOCUMENTATION:

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

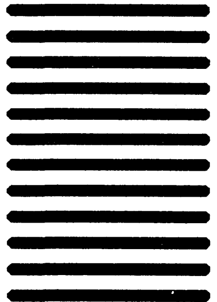
Direct mail orders to:

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)