CONVEX VMEbus STC Tape Controller (*dev5210*) Diagnostics Manual

Document No. 760-003130-000

First Edition
May 1991

CONVEX Computer Corporation
Richardson, Texas USA

CONVEX VMEbus STC Tape Controller
(dev5210) Diagnostics Manual
Order No. DHW-243
First Edition

Printed in the United States of America

Revision Sheet

CONVEX VMEbus STC Tape Controller (dev5210) Diagnostics Manual

Edition	Document No.	Date	Description
First	760-003130-000	May 1991	First release. Contains the *dev5210* diagnostic test information from the *CONVEX PBUS I/O Systems Diagnostics Manual.*

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

Appendixes

A Reporting Problems

List of Tables

List of Figures

Preface

Purpose and Intended Audience

This manual explains how to run the *dev5210* diagnostic, which checks the CONVEX VMEbus Tape Controller (VBTC) and its connected tape unit(s). This document is not a tutorial, but rather a reference for the users of the *dev5210* 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 *dev5210* 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 STC Tape Controller Test (*dev5210*)**—Describes how to operate the diagnostic, including prerequisites, test invocation, hardware initialization sequence, and class descriptions. It also describes the interactive debugger commands and gives examples of error messages.
- **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-243.
The document number for this manual is 760-003130-000.

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

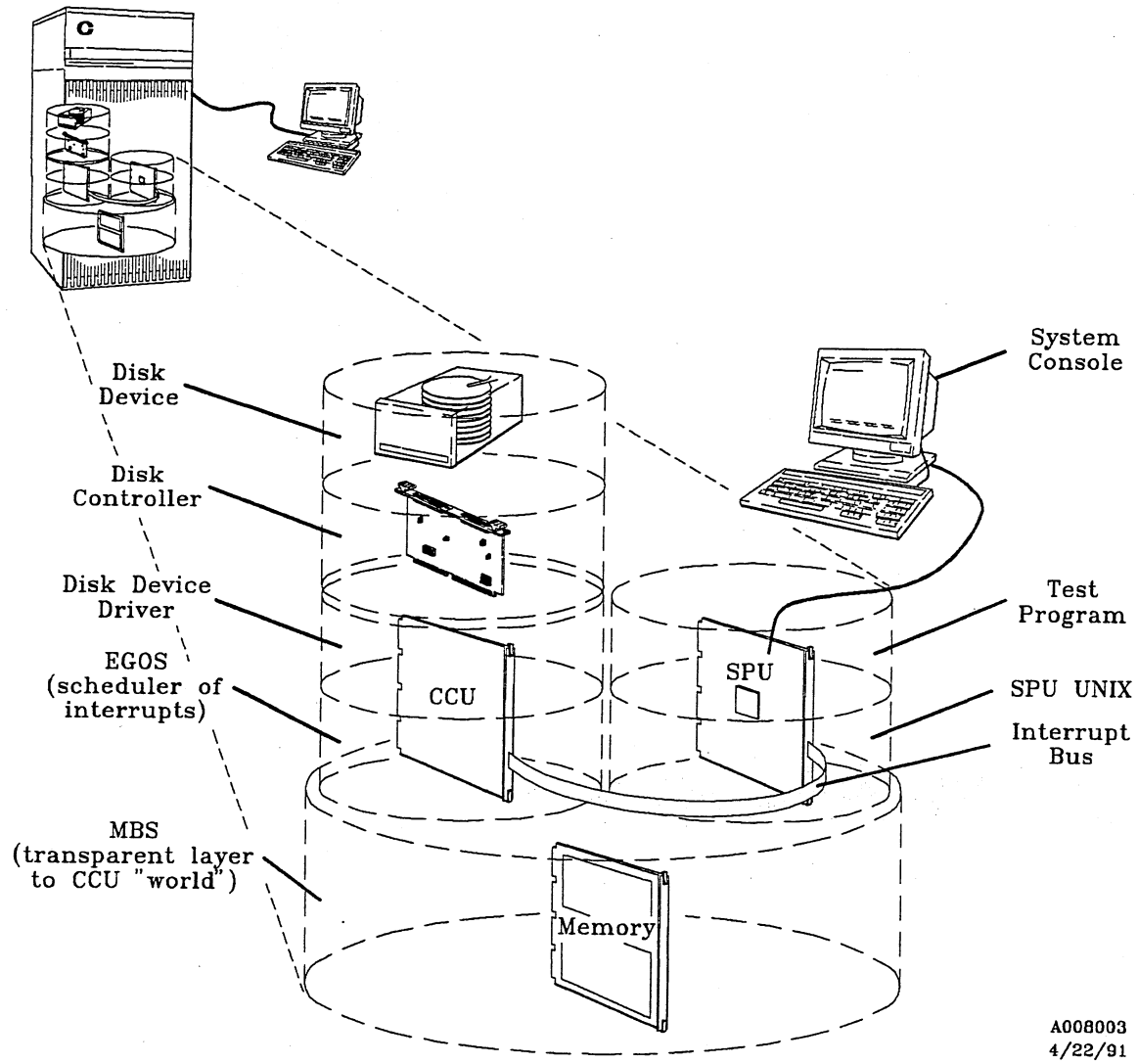
The VME interface version of EGOS is designed with a scheduler for the VIOP and is called VIOP EGOS. VIOP EGOS supports event driven device drivers as well as process type device drivers. VIOP EGOS utilizes a *sleep/wakeup* type of process control that improves efficiency of the device driver and makes 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 STC Tape Unit Controller Test (*dev5210*)

4.1 Overview

The *dev5210* test is a functional test for the CONVEX VMEbus Tape Controller (VBTC) and its connected tape unit(s). The VBTC provides a connection to a magnetic tape drive unit that is based on the STC interface. In general, this test attempts to force most software-forcible formatter and controller errors and verify that either the formatter or controller can detect them.

This functional test is modeled to test the STC 2921, STC 2922, and STC 1968 drives. In addition, the test supports testing of the Fujitsu 243XL. Read and write type testing is conducted at recording densities of:

- **800 bpi**—Nonreturn to Zero (NRZI)
- **1,600 bpi**—Phase Encoded (PE)
- **6,250 bpi**—Group Coded Recording (GCR)

Specifically, *dev5210* is designed to accomplish the following:

- Verifies the functional ability of the VBTC to operate in the CONVEX VMEbus I/O environment. This includes main memory access and interrupt generation. In addition, the FIFO is pattern tested along with verification of its ability to detect parity errors.
- Verifies 14 of the 15 supported STC interface commands. The controller's ability to use data chaining on transfers and pipeline commands is tested under normal and error conditions.
- Verifies the ability of the VBTC to detect most software-forcible formatter and controller errors. It insures that the VBTC can recover from bus errors.
- Verifies that the cable interface between the controller and the drive is operational.
- Provides an interactive debugger that can execute commands from a script file. This allows more flexibility in debugging.

Channel Control Unit (CCU) communications utilize the Event Governed Operating System (EGOS) and the Message Based System (MBS) used by ConvexOS. The intent is to test the communication paths used in a normal operating environment.

Most subtests implement an error recovery scheme to minimize the impact of media errors. Although media errors are minimized at most points the user may select a pattern that will stress data recovery. As a result, the test requires that tape media be in adequate condition. In most write subtests, identification data is written in each block. This data is used to verify positional coordinates on the tape.

The following table lists all of the STC tape commands and indicates which ones are tested:

Table 4–1, STC Interface Commands Tested

COMMAND DESCRIPTIONS					
Number	**Mnemonic**	**STC code**	**Function**	**Type**	**Tested**
1	*NOP*	0x00	No Operation	Diagnostic	Yes
2	*LWR*	0x07	Write to Read Loopback	Diagnostic	Yes
3	*DMS*	0x02	Diagnostic Mode Set	Diag/Norm	Yes
4	*CLR*	0x01	Drive Clear	Normal	Yes
5	*REW*	0x0e	Rewind to BOT	Normal	Yes
6	*RUN*	0x0f	Rewind to BOT and Unload	Normal	No
7	*ERG*	0x0d	Erase Gap	Normal	Yes
8	*WRT*	0x06	Write Block	Normal	Yes
9	*RDF*	0x04	Read Forward Block	Normal	Yes
10	*RDB*	0x05	Read Backward Block	Normal	Yes
11	*FSB*	0x0b	Forward Space Block	Normal	Yes
12	*BSB*	0x09	Backward Space Block	Normal	Yes
13	*WTM*	0x0c	Write Tape/File Mark	Normal	Yes
14	*FSF*	0x0a	Forward Space File	Normal	Yes
15	*BSF*	0x08	Backward Space File	Normal	Yes

4.2 Related Documents

This test description is intended as a reference for users who have a good understanding of the VBTC and VIOP and for those who have a basic understanding of tape drives and magnetic tape recording principles. Specifically, this test description assumes familiarity with Storage Technology Corporation's *StorageTek Standard Interface* (STC interface). The following table lists several documents that may provide additional information not contained within this test description:

Table 4–2, Related Documents

Document Title	Document Origin
STC Compatible Multibus Controller Specification	CONVEX
VME Tape Controller Difference Document	CONVEX
VME IOP Difference Document	CONVEX
The STC 2920 Maintenance Manual	STC
The STC 1960 Maintenance Manual	STC
The Fujitsu 243XL Maintenance Manual	Fujitsu

4.3 Prerequisites and Required Equipment

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

Table 4–3, Hardware Requirements

C1, C120	C200 Series
MCU	Memory System[1]
MAU	CPX
SPU	SP2
VIOP	VIOP
VBCU	PIA
VBTC[2]	VBCU
Tape Drive	VBTC[2]
	Tape Drive

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

[2] VBTC represents VMEbus Tape Controller

4.4 Test Invocation

The *dev5210* 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 *dev5210* test, use the procedure shown in Figure 4–1, "Initial Test Invocation Sequence." 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.

> **NOTE**
>
> Use the following test invocation sequence for the initial invocation of *dev5210* or when the state of the machine is unknown. Also, the following invocation sequence should be used if any hard errors have occurred since the last system initialization.

Figure 4–1, Initial Test Invocation Sequence

```
(spu)> cd /mnt/test RETURN
(spu)> sysreset RETURN
(spu)> mminit -s RETURN
(spu)> dshell RETURN
: test dev5210 [-c [class number(s)]] [-s [subtest number(s)]] [-dnqV] [-f FILE] [+>filename] RETURN
```

NOTE

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

The format for the *test* command is as follows:

test dev5210 [*option ...*] [[+>*filename*]]

where *option* is one of the following:

-c *class-number*	Execute one or more specific classes of subtest(s)
-d	Enter Debugger (no subtests are executed)
-n	Do not actually execute any subtest (used to create a parameter file without actually executing any subtest)
-q	Quick startup (use previous parameters from last test invocation—same as entering **dev5210x**; refer to the "Alternate Test Invocation Sequence")
-s *subtest-number*	Execute one or more individual subtests
-V	Print version string (compilation date of test–last modification date)
-f *FILE*	Use *FILE* as the parameter save file. If this option is omitted, */tmp/dev5210.tmp* is used as the parameter file.

Entering only **test dev5210** executes all *dev5210* subtests sequentially. The [+>*filename*] option allows the test results to be appended to *filename.*

NOTE

The following alternate test invocation procedure is optimal when invoking *dev5210* multiple times. Using this invocation sequence insures that the test is invoked and executed with all of the setup parameters supplied when the test was last executed with the initial invocation sequence.

Figure 4–2, Alternate Test Invocation Sequence

```
(spu)> cd /mnt/test RETURN
(spu)> dshell RETURN
:test dev5210x [-c [class number(s)]] [-s [subtest number(s)]] [-dnqV] [-f FILE] [+>filename] RETURN
```

The only difference in this alternate invocation sequence is the **x** after **dev5210**. When invoking *dev5210* in this manner, no prompts are displayed. The diagnostic obtains all prompt information from the parameter file created when the initial invocation sequence was performed.

4.4.1 Test Parameter Menu

Once the test is invoked, a test menu prompt is presented allowing selection of default switches. Figure 4–3, "Test Parameter Menu," shows all prompts, their possible answers (in brackets [ ]), and their default answers (in parentheses ( )). The prompts and responses in the figure appear sequentially on the screen, one line at a time. The figure 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. The numbers displayed on the screen during testing may not correspond to those shown in the example, as the questions illustrated are examples only.

For help or information during test parameter entry, enter one of the following characters followed by a RETURN:

Table 4–4, Getting Help During Test Parameter Entry

Character	Description
:?	Reviews previous entries
?	Provides specific help where available

After the desired help information displays, the system redisplays the last prompt.

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
        ?       Provides specific help where available

 1: Select Configuration File [<filepath>,?]      (/ioconfig) -> RETURN

                    PERIPHERAL CONFIGURATION DATA
             CCU    Chassis   Type    CSR    Int Unit   Type
            -------- ------- -------- ------ --- ---- --------
          1) viop  3     0    MTC-201  0x0600  1   1   MTD-204
          2) viop  3     0    MTC-201  0x3fc0  4   0   MTD-201

          *** Enter 0 for manual configuration ***

 2: Select Configuration File Entry to Test [0,1-2,?]     (1) -> 0
 3: Display VBTC Configuration Help [y,n,?]               (y) -> n
 4: VME IOP Number [3-7,?]                                (3) -> 3
 5: VME Chassis Number [0-1,?]                            (0) -> 0
 6: VMEbus Base Address of the Controller [0x0-0xffff,?]
                                                  (0x600) -> 0x600
 7: VMEbus Interrupt Level of the Controller [1-7,?]      (1) -> 1
 8: Tape Unit Number [0-7,?]                              (1) -> 1

          ** Tape Drive models **
    1:    Storage Technology Corp. 2921 ..................... MTD-201
    2:    Storage Technology Corp. 2922 ..................... MTD-204
    3:    Storage Technology Corp. 1963/1968 ................ MTD-202
    4:    Fujitsu 2436-Li ................................... MTD-203

 9: Tape Drive Type [1-4,?]                               (2) -> 2
10: Use Standard Defaults for Other Options [y,n,?]       (y) -> n

          ** Tape Spool Size in Feet **
    1:     600 ft.
    2:    1200 ft.
    3:    2400 ft.
    4:    3600 ft.

11: Select Mounted Tape Spool Size [1-4,?]                (1) -> 1
12: Run EOT Sensor Subtest #701 [y,n,?]                   (y) -> y
```

Figure 4–3, Test Parameter Menu (continued)

```
            ** Available Data Transfer VMEbus Address Modifiers **
    1:      Standard Supervisory Access (16 bit) (0x3d)
    2:      Standard Non-Supervisory Access (16 bit) (0x39)

13: Data Transfer VMEbus Address Modifier [1-2,?]          (1) -> 1

            ** Available Tape Velocities (IPS) **
    1:       50
    2:      100

14: Standard Tape Velocity [1-2,?]                         (2) -> 2
15: Standard Tape Block Size [16-16M,?]                   (8K) -> 8K
16: Large (Gapless) Tape Block Size [256K-16M,?]          (1M) -> 16M
*** WARNING:
    Due to insufficient tape length, the selected Large Block of 16777216
    will be reduced to 5760000 when using PE   recording format.
17: Run Only High Density Gapless Subtests (520,521) [y,n,?]
                                                           (y) -> y

  ** Printer On/Off Enable/Disable Options (bit mapped) **
    0x0001: Enable printer ON string before error
    0x0002: Enable printer OFF string after error

18: Select Printer On/Off Mode [0x0-0x3,?]               (0x0) -> 0x3
19: Printer On Character Sequence (before error)
    [<printer on string>,?]                        (\033[?5i) -> \033[?5i
20: Printer Off Character Sequence (after error)
    [<printer off string>,?]                       (\033[?4i) -> \033[?4i
21: Use Standard Debug Setup [y,n,?]                       (y) -> n

  ** USER Debug Options (bit mapped) **
    0x0001: Disable Asynchronous Command Mode
    0x0002: Disable Multi-Record Buffering
    0x0004: Disable Reset of Controller After ^C
    0x0008: Disable Hard Error Trapping (logging)
    0x0010: Enter Debugger After an Error Occurs
    0x0020: Disable copying of errors to error file
    0x0100: Pause AFTER each CCU driver command is returned
    0x0200: Show each CCU driver message AFTER it is returned
    0x0400: Pause BEFORE each CCU driver message is sent
    0x0800: Show each CCU driver message BEFORE it is sent
    0x1000: Automatically enter debugger during a pause

22: User Debug Option Mask [0x0-0x1fff,?]                (0x0) -> 0x1fff

            ** VBTC register dump modes **
    1:      After Error
    2:      After Each Command Completion
    3:      Never

23: Select Controller Register Dump Mode [1-3,?]           (3) -> 3
```

Figure 4–3, Test Parameter Menu (continued)

```
          ** Tape Media Error Logging Methods (bit mapped) **
    0x0001: After Test Completion
    0x0002: After Each Subtest
    0x0004: Upon Occurrence (Real Time) With Limited Register Dump
    0x0008: Upon Occurrence (Real Time) With Full Register Dump
    0x0010: Activate Printer During Media Errors (See Printer Options)

24: Select Media Error Logging Method [0x0-0x1f,?]      (0x3) -> 0x1f
25: Maximum Media Error Retry Count [0-999,?]            (10) -> 10
26: Maximum # of Non-Fatal Media Errors Allowed [0-9999,?]
                                                        (40) -> 40
27: Retry Formatter Correctable Media Errors On Writes [y,n,?]
                                                         (y) -> y
28: Pattern for Write/Read Subtests [<hexadecimal pattern>,?]
                                                  (0x6db6) -> 0x6db6

          ** Available Data Miscompare Dump Modes **
    1:    Separate Expected and Actual Values into Two Blocks.
    2:    Mix Expected and Actual Values (xx/xx).
    3:    Only Show Bytes that Miscompare.

29: Select Data Miscompare Display Mode [1-3,?]          (2) -> 2
30: Enter Data Miscompare Line Dump Count [1-1000,?]     (4) -> 4
31: Enter OK, or :NN to return to question NN [OK]      (OK) -> RETURN
```

NOTE

In the previous figure, a warning statement (see prompt #16) is printed because the selected tape length for the displayed density is not large enough to hold the large tape block selected. The test automatically compensates for this by changing the large tape block size to match the tape length. In all cases, the maximum large tape block size is limited by the amount of contiguous physical memory present in the target system and the maximum tape length. It is important to note that *dev5210* only compensates for insufficient tape length in the data chain mode read/write tests.

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

When all prompts are answered, the screen displays a test parameter summary which displays the prompts that were answered and their responses. Figure 4–4, "Sample Test Parameter Summary," shows a sample test parameter summary. The actual values and responses vary according to the input.

Figure 4–4, Sample Test Parameter Summary

```
                    TEST PARAMETER SUMMARY

Select Configuration File                                    : /ioconfig
Select Configuration File Entry to Test                      : 0
Display VBTC Configuration Help                              : y
VME IOP Number                                               : 3
VME Chassis Number                                           : 0
VMEbus Base Address of the Controller                        : 0x600
VMEbus Interrupt Level of the Controller                     : 1
Tape Unit Number                                             : 1
Tape Drive Type                                              : 2
          -> Storage Technology Corp. 2922 <-
Use Standard Defaults for Other Options                      : n
Select Mounted Tape Spool Size                               : 1
          ->  600 ft. <-
Run EOT Sensor Subtest #701                                  : y
Data Transfer VMEbus Address Modifier                        : 1
          -> Standard Supervisory Access (16 bit) (0x3d) <-
Standard Tape Velocity                                       : 2 (100 IPS)
Standard Tape Block Size                                     : 8K
Large (Gapless) Tape Block Size                              : 16M
Run Only High Density Gapless Subtests (520,521)             : y
Select Printer On/Off Mode                                   : 0x0003
          -> Enable printer ON string before error <-
          -> Enable printer OFF string after error <-
Printer On Character Sequence (before error)                 : \033[?5i
Printer Off Character Sequence (after error)                 : \033[?4i
Use Standard Debug Setup                                     : n
User Debug Option Mask                                       : 0x1fff
          -> Disable Asynchronous Command Mode <-
          -> Disable Multi-Record Buffering <-
          -> Disable Reset of Controller After ^C <-
          -> Disable Hard Error Trapping (logging) <-
          -> Enter Debugger After an Error Occurs <-
          -> Disable copying of errors to error file <-
          -> Pause AFTER each CCU driver command is returned <-
          -> Show each CCU driver message AFTER it is returned <-
          -> Pause BEFORE each CCU driver message is sent <-
          -> Show each CCU driver message BEFORE it is sent <-
          -> Automatically enter debugger during a pause <-
Select Controller Register Dump Mode                         : 3
          -> Never <-
Select Media Error Logging Method                            : 0x001f
          -> After Test Completion <-
          -> After Each Subtest <-
          -> Upon Occurrence (Real Time) With Limited Register Dump <-
          -> Upon Occurrence (Real Time) With Full Register Dump <-
          -> Activate Printer During Media Errors (See Printer Options) <-
Maximum Media Error Retry Count                              : 10
Maximum # of Non-Fatal Media Errors Allowed                  : 40
Retry Formatter Correctable Media Errors On Writes           : y
Pattern for Write/Read Subtests                              : 0x6db6
Select Data Miscompare Display Mode                          : 2
          -> Mix Expected and Actual Values (xx/xx). <-
Enter Data Miscompare Line Dump Count                        : 4
Enter OK, or :NN to return to question NN                    : OK
```

4.5 Initialization Sequence for *dev5210*

After the last prompt is entered, and before subtest code execution, the diagnostic performs the following:

- Determines if the test was invoked for quick startup (i.e., dev5210x or dev5210 -q). If so, the diagnostic reads the test parameters from the specified parameter file (default parameter file is */tmp/dev5210.tmp*). If not, the user is presented with the parameter menu. After parameter input, the responses are written to the parameter file (default or user-specified).
- Locates the largest contiguous main memory space after the first 2 Mbytes and reserves it for use by *dev5210*.
- Determines if the CCU is already loaded with the *dev5210* CCU driver. If the driver is not loaded, the CCU is reloaded. After the load completes, the driver is configured for EGOS and then the EGOS probe message starts the driver.
- Passes the current test parameters (i.e., retry information) to the CCU driver.

> **NOTE**
>
> The file */mnt/boot_db* is used to determine what memory is installed. If this file is nonexistent, it can be created by entering: **scn_util -b > /mnt/boot_db** from the **(spu)>** prompt.

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

4.5.1 Prompt Explanations

The test parameter prompts are repeated and explained in the following paragraphs.

```
1: Select Configuration File [<filepath>,?]        (/ioconfig) ->
```

Enter the name of an alternate */ioconfig* file if an alternate is desired. The file must still be in the same format as a conventional */ioconfig* file.

```
                PERIPHERAL CONFIGURATION DATA
    CCU     Chassis   Type     CSR    Int Unit   Type
 -------- ------- --------- ------ --- ---- ---------
1) viop  3     0    MTC-201  0x0600  1   1   MTD-204
2) viop  3     0    MTC-201  0x3fc0  4   0   MTD-201
    *** Enter 0 for manual configuration ***
2: Select Configuration File Entry to Test [0,1-2,?]      (1) ->
```

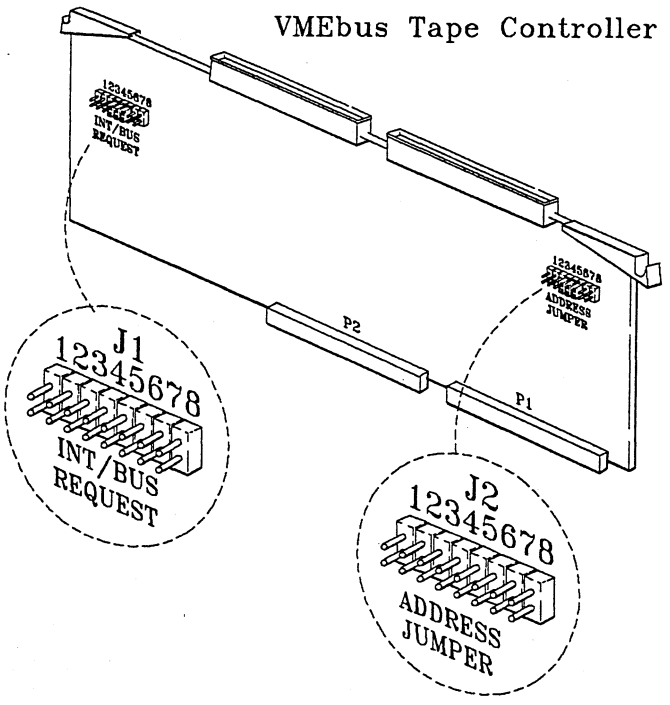
Enter the */ioconfig* file entry you want to test (if any). To select a predefined controller configuration entry from the */ioconfig* file, enter the number that is found to the left of the desired entry. Entering a **0** allows each item within the controller configuration entry to be independently specified. If most (but not all) of the items associated with a given entry in

/ioconfig are desired, select the number for that entry, backup to this prompt via the caret command (SHIFT **6**), and then specify **0** the second time. This action causes the default values for the independent items to be the same as the originally-selected entry.

```
3: Display VBTC Configuration Help [y,n,?]                    (y) ->
```

This option displays the VME tape controller information. This information is useful when installing the jumpers that determine the controller's base address and interrupt level. The VBTC has two jumper blocks which are used to set the board's base address, interrupt level, and VMEbus request level. All logic is inverted, i.e., a removed jumper sets that bit. The jumper blocks are shown in the following figure:

Figure 4–5, VMEbus Tape Controller Jumper Blocks



The jumper blocks are defined in the following tables:

> **NOTE**
>
> From left to right, the jumper block pin numbers on the VBTC are numbered 1 through 8. To make it easier to construct binary patterns, they are listed from 8 through 1 in the following tables, with 8 being the most significant bit and 1 being the least significant bit. Also, the information listed in the following three tables is displayed on the screen by entering **y** or **?** as response to the prompt.

Table 4–5, VBTC Jumper Block Pin Settings

Jumper Block 1—J1 (Upper left side of board)								
Jumper	8	7	6	5	4	3	2	1
Bit	INT2	INT1	INT0	XX	XX	XX	BR1	BR0

Table 4–6, VBTC Jumper Block Pin Settings

Jumper Block 2—J2 (Upper right side of board)								
Jumper	8	7	6	5	4	3	2	1
Address	A13	A12	A11	A10	A9	A8	A7	A6

Table 4–7, Standard Address and Interrupt Levels

VBTC #	Address	Interrupt Level	Bus Request
1	0x1000	7	3
2	0x1040	site dependent	3
3	0x1080	site dependent	3
4	0x10c0	site dependent	3

```
4: VME IOP Number [3-7,?]                                    (3) ->
```

Enter the CCU slot number for the VMEbus IOP containing the VMEbus subsystem connected to the tape controller/drive. The CONVEX VMEbus tape controller part number is 410-001152-600.

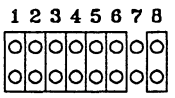
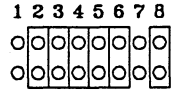
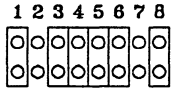
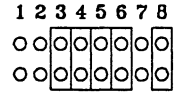
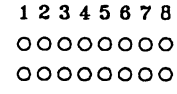
```
5: VME Chassis Number [0-1,?]                                    (0) ->
```

Enter the VMEbus chassis number for the chassis where the tape controller is installed. If there is only one chassis, the number is (in most cases) 0. The CONVEX VMEbus tape controller part number is 410-001152-600.

```
6: VMEbus Base Address of the Controller [0x0-0xffff,?]
                                                          (0x600) ->
```

Enter the low-order four hexadecimal digits of the VMEbus tape controller board's base address within the VMEbus standard address space. This address is controlled by jumper block 2 (**J2**) on the controller. The typical address is 0x1000. When all the straps are removed, the controller defaults to address 0x3fc0.

Table 4–8, VMEbus Address Level Jumper Settings (J2)

Address	Jumper Block 2 Settings
0x1000	
0x1040	
0x1080	
0x10c0	
0x3fc0	


```
7: VMEbus Interrupt Level of the Controller [1-7,?]        (1) ->
```

Enter the CONVEX VMEbus interrupt number assigned to the VMEbus tape controller board under test. This value is between 1 and 7, inclusive. The interrupt level is selected by jumper block 1 (**J1**) on the controller. The typical level is 7. When all the straps are removed, the controller defaults to level 7.

Table 4–9, VMEbus Interrupt and Bus Request Jumper Settings (J1)

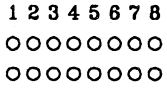
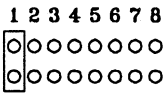
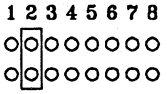
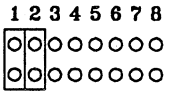
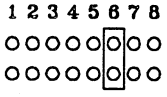
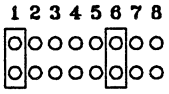
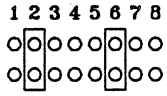
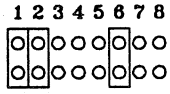
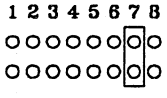
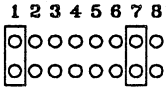
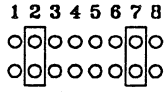
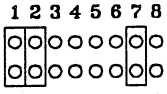
Interrupt Level	Bus Request	Jumper Block 1 Settings
7	3	
7	2	
7	1	
7	0	
6	3	
6	2	
6	1	
6	0	
5	3	
5	2	
5	1	
5	0	

Table 4–9, VMEbus Interrupt and Bus Request Jumper Settings (J1) (continued)

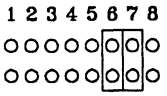
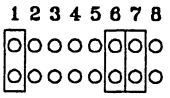
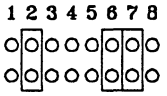
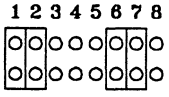
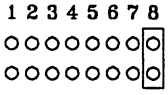
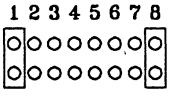
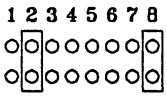
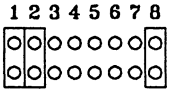
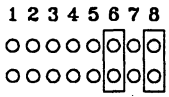
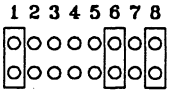
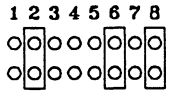
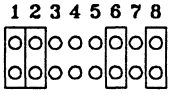
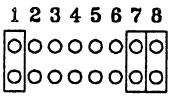
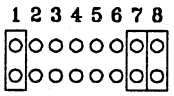
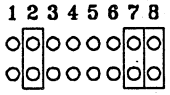
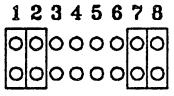
Interrupt Level	Bus Request	Jumper Block 1 Settings
4	3	
4	2	
4	1	
4	0	
3	3	
3	2	
3	1	
3	0	
2	3	
2	2	
2	1	
2	0	

Table 4–9, VMEbus Interrupt and Bus Request Jumper Settings (J1) (continued)

Interrupt Level	Bus Request	Jumper Block 1 Settings
1	3	
1	2	
1	1	
1	0	

```
8: Tape Unit Number [0-7,?]                                    (1) ->
```

The STC tape drive interface is a selector type interface (i.e., daisy chained). This entry specifies the tape unit select of the drive to test.

```
    ** Tape Drive models **
1:  Storage Technology Corp. 2921 ..................... MTD-201
2:  Storage Technology Corp. 2922 ..................... MTD-204
3:  Storage Technology Corp. 1963/1968 ................ MTD-202
4:  Fujitsu 2436-L1 ................................... MTD-203

9: Tape Drive Type [1-4,?]                                     (2) ->
```

Enter the menu number for the drive type to be tested. Menu entries are listed in the order they appear in the */mnt/bin/lib/DBtapefmt* file.

```
10: Use Standard Defaults for Other Options [y,n,?]            (y) ->
```

Enter **yes** to use the default values for all remaining prompts. This action eliminates the need to explicitly enter a RETURN in response to every remaining prompt.

```
   ** Tape Spool Size in Feet **
1:  600 ft.
2: 1200 ft.
3: 2400 ft.
4: 3600 ft.
11: Select Mounted Tape Spool Size [1-4,?]                (1) ->
```

Enter the menu number for the tape spool size loaded on the tape drive under test. This option allows limiting the amount of data written by any subtest, if applicable, to the amount that will fit on the tape. Four tape spool sizes are supported (600 ft., 1200 ft., 2400 ft., and 3600 ft.). Currently, this option only applies to the chain mode write/read subtests in the following list:

- 320—Chain mode write (NRZI)
- 321—Chain mode read (NRZI)
- 420—Chain mode write (PE)
- 421—Chain mode read (PE)
- 520—Chain mode write (GCR)
- 521—Chain mode read (GCR)

```
12: Run EOT Sensor Subtest #701 [y,n,?]                   (y) ->
```

Enter **y** if subtest 701, the EOT Sensor test, is to be executed. If a large tape spool is mounted, the EOT subtest can take more than five minutes to execute on some of the larger tape spool sizes.

```
   ** Available Data Transfer VMEbus Address Modifiers **
1:  Standard Supervisory Access (16 bit) (0x3d)
2:  Standard Non-Supervisory Access (16 bit) (0x39)
13: Data Transfer VMEbus Address Modifier [1-2,?]         (1) ->
```

Enter the menu number for the desired VMEbus address modifier to use with all VMEbus data transfers from or to the tape drive. The CONVEX VMEbus subsystem answers to all VMEbus address modifiers. Therefore, this option is likely to have no affect on transfers.

```
 ** Available Tape Velocities (IPS) **
1:   50
2:  100
14: Standard Tape Velocity [1-2,?]                        (2) ->
```

Enter the menu number for the default tape velocity (tape speed in inches per second) to use for all subtests that are not dependent on tape velocity.

```
15: Standard Tape Block Size [16-16M,?]                 (8K) ->
```

Enter the tape block size to use for all subtests that do not use predefined block sizes (90% of the tests in Classes 3, 4, and 5). A size greater than 128 Kbytes results in all standard transfers being performed in data chain mode with a chain interval size of 64 Kbytes. Increasing the size can greatly increase the amount of time required for test execution. An even binary-size multiple may be specified by appending the letter "K" for kilobytes or "M" for megabytes.

```
16: Large (Gapless) Tape Block Size [256K-16M,?]          (1M) ->
```

Enter the tape block size to use for the chain mode write and read subtests. The minimum size is 256 Kbytes. ConvexOS can write tape blocks as long as 16 Mbytes. The maximum size is limited by the amount of contiguous physical memory present in the target system. An even binary-size multiple may be specified by appending the letter "K" for kilobytes or "M" for megabytes.

NOTE

With the default retry limit of 10, the test will erase up to 17.5 feet of tape during an attempt to recover from a media error (1.75 feet per retry). The maximum STC legal gap size is 20 feet. Gaps longer than 20 feet may cause the formatter to timeout searching for a tape block. Therefore, if the specified block size results in a tape block longer that 17.5 feet, a simple media error beyond the first 17.5 feet results in a failure. The following table illustrates how tape block sizes in bytes relate to sizes in feet.

Table 4–10, Tape Block Size Relationships

Block Size		Block Length (feet)		
Kbytes	Mbytes	GCR	PE	NRZI
16384	16.0	223.6	873.8	1747.6
1024	1.0	13.9	54.6	109.2
512	0.5	7.0	27.3	54.6

```
17: Run Only High Density Gapless Subtests (520,521) [y,n,?]
                                                          (y) ->
```

Enter **y** to inhibit the execution of the NRZI and PE density Data Chain Mode subtests (i.e., NRZI: 320,321; PE: 420,421). This response is recommended. A "n" response requires that the tape drive and media be in superb condition because Data Chain Mode is used to write very large tape blocks, and larger blocks are more prone to media errors.

```
** Printer On/Off Enable/Disable Options (bit mapped) **
  0x0001: Enable printer ON string before error
  0x0002: Enable printer OFF string after error
```

```
18: Select Printer On/Off Mode [0x0-0x3,?]                (0x0) ->
```

This prompt allows a specified character string to be sent to the printer before and/or after an error (and its data) is displayed. Although any string up to 64 characters in length may be specified, the intended use is to selectively turn a printer on before an error is displayed and to turn a printer off after an error has been displayed. This option is a paper-saving feature when executing *dev5210* multiple times or for long periods of time. This assumes a printer is connected to the auxiliary port on the display terminal where *dev5210* is executing and that the auxiliary port can be turned on and off via an escape character sequence.

To select both options, enter the hexadecimal mask obtained by ORing the two options together (0x03).

```
19: Printer On Character Sequence (before error)
    [<printer on string>,?]                          (\033[?5i) ->
```

This option allows the specification of the enable printer port escape character sequence. The default value is the "Enter Auto Print Mode" sequence for terminals that support VT100 terminal protocol. Control characters may be specified by using standard escape sequences used in C programming, for example:

\033	Octal value of the ASCII escape character (0x1b)
\x1b	Same as above value but specified in hex
\r	Carriage return
\n	Line feed
\t	Tab character
\b	Backspace character
\f	Form feed character

```
20: Printer Off Character Sequence (after error)
    [<printer off string>,?]                         (\033[?4i) ->
```

This option allows the specification of the disable printer port escape character sequence. The default value is the "Exit Auto Print Mode" sequence for terminals that support VT100 terminal protocol. Control characters may be specified by using standard C programming style escape sequences. For example:

\033	Octal value of the ASCII escape character (0x1b)
\x1b	Same as above value but specified in hex
\r	Carriage return
\n	Line feed
\t	Tab character
\b	Backspace character
\f	Form feed character

```
21: Use Standard Debug Setup [y,n,?]                          (y) ->
```

Enter **y** to use the default values for all prompts related to debug mode. This action eliminates the need to explicitly enter a RETURN in response to every remaining prompt.

```
** USER Debug Options (bit mapped) **
  0x0001: Disable Asynchronous Command Mode
  0x0002: Disable Multi-Record Buffering
  0x0004: Disable Reset of Controller After ^C
  0x0008: Disable Hard Error Trapping (logging)
  0x0010: Enter Debugger After an Error Occurs
  0x0020: Disable copying of errors to error file
  0x0100: Pause AFTER each CCU driver command is returned
  0x0200: Show each CCU driver message AFTER it is returned
  0x0400: Pause BEFORE each CCU driver message is sent
  0x0800: Show each CCU driver message BEFORE it is sent
  0x1000: Automatically enter debugger during a pause

22: User Debug Option Mask [0x0-0xfff,?]                (0x0) ->
```

Enter the number that indicates which debug option you want to enable. These options allow the standard test behavior to be altered in ways that are often useful for troubleshooting. In addition, the test can be paused before or after each command (i.e., typically one controller operation) is sent to the CCU driver. If you want multiple options, enter the hexadecimal mask obtained by ORing the desired options together. For example, if the first two options are desired, enter **0x3**. If all options are desired, enter **0x1fff**. Each option is described below in more detail.

`0x0001: Disable Asynchronous Command Mode`

This option inhibits the asynchronous command mode used in most of the write/read subtests. Asynchronous command mode occurs when multiple CCU driver commands are transmitted without waiting for a response to the commands. This mode allows the test to meet the tape drive reinstruct time and keep the tape spinning at maximum velocity by eliminating the 2 ms to 4 ms overhead required to transmit a command and receive its response via MBS. Unfortunately, the asynchronous command mode can be confusing when using the CCU driver single-step mode; therefore, selecting this option disables the asynchronous command mode.

`0x0002: Disable Multi-Record Buffering`

This option inhibits the multirecord buffering that takes place in 75% of the write/read subtests. Multirecord buffering causes the test to write or read as much data as the main memory buffer can hold at once. This buffering allows the test to keep the tape spinning at maximum velocity as long as possible (i.e., the number of tape start/stop cycles is reduced). Unfortunately, this buffering can be confusing when using the register dump on error mode. For example, if 4 blocks are read into the buffer and a data compare error occurs on the first and if the register dump on error mode was set, the register dump would be for the fourth block read, not the first. Therefore, this option forces one read at a time each followed by a data compare.

> **NOTE**
>
> The previous option does not work for all subtests that do multirecord buffering.

`0x0004: Disable Reset of Controller After ^C`

Normally, the controller is reset whenever the test is aborted by entering CTRL C. This action may destroy valuable register states in some cases. This option prevents the reset from occurring.

`0x0008: Disable Hard Error Trapping (logging)`

By default, this subtest monitors for hard system errors and aborts if a hard error occurs. Because hard errors are processed as an interrupt to the normal test logic, the state of the test

cannot be reported when a hard error occurs. Since hard errors always result in the immediate halt of the main memory system, a subtest may eventually fail without hard error logging enabled. This is often useful since the subtest state is reported (i.e., more fail data is reported to allow more insight into the possible source of the problem).

```
0x0010: Enter Debugger After an Error Occurs
```

This option specifies automatic entry into the interactive debugger any time an error is detected within a subtest. The debugger may also be invoked by specifying the *-d* option at test invocation time (i.e., **dev5210**[x] **-d**).

```
0x0020: Disable copying of errors to error file
```

This option disables the automatic copying of error reports to the file */tmp/dev5210.err*. This option might be used if 1000 lines of miscompared data were to be displayed; in that case, the file system might fill up or it could require a large amount of time to write the data to the file.

```
0x0100: Pause AFTER each CCU driver command is returned
0x0200: Show each CCU driver message AFTER it is returned
0x0400: Pause BEFORE each CCU driver message is sent
0x0800: Show each CCU driver message BEFORE it is sent
0x1000: Automatically enter debugger during a pause
```

These bit options allow single stepping before and/or after each CCU driver command. In addition, the internal command can be displayed by selecting one of the "show command" option bits. This is a valuable debugging tool when used with the register dump after each command mode. Also, the main memory read/write buffer and CCU memory can be examined by entering the debugger. See the "Interactive Debugger" section at the end of this test description for more details on the debugger.

For most cases, the "after" modes are sufficient. If the bit for the option "Automatically enter debugger during a pause" is not set, the test will pause, at which time the user has the following options:

Space bar	Continue
Return	Continue
:	Enter debugger

```
 ** VBTC register dump modes **
1: After Error
2: After Each Command Completion
3: Never

23: Select Controller Register Dump Mode [1-3,?]          (3) ->
```

Enter the menu number that indicates when to display the last known register state of the VBTC. The "After Each Command Completion" mode only works if the "CCU driver single step mode" option (show each CCU driver message after it is returned) is selected in the next option.

```
  ** Tape Media Error Logging Methods (bit mapped) **
0x0001: After Test Completion
0x0002: After Each Subtest
0x0004: Upon Occurrence (Real time) With Limited Register Dump
0x0008: Upon Occurrence (Real Time) With Full Register Dump
0x0010: Activate Printer During Media Errors (See Printer Options)

24: Select Media Error Logging Method [0x0-0x1f,?]        (0x3) ->
```

Enter the number that indicates when tape media errors are logged to the display. If you want multiple options, enter the hexadecimal mask obtained by ORing the desired options together. For example, if the first two options are desired, enter **0x3**. If all options are desired, enter **0x1f**.

Media errors are isolated into two classes:

- **1**—Formatter-correctable errors
- **2**—Uncorrectable errors

Formatter-correctable errors are errors that are mathematically correctable by the tape formatter. These errors can occur on both reads or writes. In the case of a write, a formatter-correctable error is detected by the read after write logic in the formatter. Uncorrectable errors cannot be corrected by the formatter and must be retried. Both of the error types can be logged as they occur, after each subtest, after the entire test, or after both. The individual options are selected by ORing the individual option bits together to build a mask of the desired options. Each option is described below in more detail:

```
0x0001: After Test Completion
```

This option prints a summary of all media errors that occurred during the entire test. This summary is printed after the last subtest is executed or at test termination when a fatal error has occurred. The format of the report is as follows:

```
Media Error Class ---- Total ---- NOT Recovered ---- Recovered ---- Ignored
Formatter-Corrected       13                  0              0          13
Uncorrectable              0                  0              0           0
```

Formatter-corrected errors during a read operation are ignored.

```
0x0002: After Each Subtest
```

This option specifies that a report such as the one in the previous example is to be printed after each subtest completes. The report is only printed if one or more media errors occur. In addition, only the applicable classes of media errors will be printed (i.e., if one formatter-correctable error occurred, the uncorrectable line would be omitted).

```
0x0004: Upon Occurrence (Real Time) With Limited Register Dump
```

This option prints a detailed media error report at the time a media error is detected. The report includes:

- **1**—Media error class
- **2**—Action taken (Recovered/Not recovered/Ignored)
- **3**—The number of retries required to recover the error (if any)
- **4**—The tape position (block and or file number if known)
- **5**—The pertinent controller registers that reflect the error

```
0x0008: Upon Occurrence (Real Time) With Full Register Dump
```

This option is identical to the previous "Real Time" option except that, with this option, all registers are printed after a media error occurs.

```
0x0010: Activate Printer During Media Errors (See Printer Options)
```

This option causes the Printer On/Off character sequences to be sent to the display before and after any media error reports displayed. This option allows the printer to be turned on and off automatically to include only the media error reports.

> **NOTE**
>
> In order for this option to work, the printer on/off character sequences must be enabled via the "Select Printer On/Off Mode" option (see option 18).

```
25: Maximum Media Error Retry Count [0-999,?]          (9) ->
```

Enter the maximum number of media-type error retries to be made after the original attempt when attempting to recover from a media error. A retry count of 9 results in 10 total maximum tries of a tape operation. A 0 value causes all media errors to be considered fatal.

```
26: Maximum # of Non-Fatal Media Errors Allowed [0-9999,?]
                                                       (40) ->
```

Enter the maximum number of non-fatal (i.e., recovered plus ignored) media errors allowed during one invocation of the test. An ignored media error could be a formatter-correctable error during a read operation. Recovered media errors are errors that are corrected by a retry.

The default is 20 errors per available recording density (e.g., NRZI, PE, GCR). For strenuous testing, set this value to 5 or less, which requires tape media that is in superb condition.

```
27: Retry Formatter Correctable Media Errors On Writes [y,n,?]
                                                         (y) ->
```

Enter **y** to retry tape media errors that the tape formatter can correct. This option only applies to tape writes; a formatter-correctable error during a read is not retried. An example of a formatter-correctable error during a write would be a single dead track while in GCR mode. The formatter can mathematically reconstruct a single dead track in GCR.

```
28: Pattern for Write/Read Subtests [<hexadecimal pattern>,?]
                                                    (0x6db6) ->
```

Enter the hexadecimal data pattern to use for all tape write/read subtests. The pattern may be any length from 1 nibble to 256 bytes. The pattern is replicated as necessary when formatting buffers for tape writes.

The pattern can be specified via a file on the SPU disk. For this method enter a "+" followed by the optional file name of the pattern file. If the file name is omitted, the test defaults to the file *dev5210.pat*. The pattern may contain white space and may be spread across multiple lines.

NOTE

The buffer fill pattern preceding a read is "0x55aa55aaaa55aa55." Therefore, to ensure accurate test results, this pattern should be avoided as the write/read data pattern.

```
  ** Available Data Miscompare Dump Modes **
1: Separate Expected and Actual Values into Two Blocks.
2: Mix Expected and Actual Values (xx/xx).
3: Only Show Bytes that Miscompare.

29: Select Data Miscompare Display Mode [1-3,?]          (2) ->
```

Enter the menu number for the desired format of data miscompare dumps. The "separate block" mode requires the most display space while "mixed" mode uses the least amount of space. The "show miscompares only" gives greater depth into the data (assuming miscomparisons occur at sparse intervals). A sample of each format's output follows:

```
------------ Separate Expected and Actual Values into Two Blocks ------------
Expected:
    00100000:  6d* b6  6d  b6* 6d  b6  6d  b6* 6d  b6  6d  b6  6d  b6  6d* b6
    00100010:  6d  b6  6d  b6  6d  b6  6d  b6  6d  b6  6d  b6  6d  b6  6d  b6
Actual:
    00100000:  00* b6  6d  77* 6d  b6  6d  88* 6d  b6  6d  b6  6d  b6  99* b6
    00100010:  6d  b6  6d  b6  6d  b6  6d  b6  6d  b6  6d  b6  6d  b6  6d  b6


------------------- Mix Expected and Actual Values (xx/xx) ------------------
Expected/Actual:
    00100000:  6d/00* b6/b6  6d/6d  b6/77* 6d/6d  b6/b6  6d/6d  b6/88*
    00100008:  6d/6d  b6/b6  6d/6d  b6/b6  6d/6d  b6/b6  6d/99* b6/b6


--------------------- Only Show Bytes that Miscompare ----------------------
Expected/Actual (Miscompares only):
    00100000: 6d/00*  00100003: b6/77*  00100007: b6/88*  0010001e: b6/88*
```

```
30: Enter Data Miscompare Line Dump Count [1-1000,?]      (4) ->
```

Enter the maximum number of display lines to use when dumping data after a miscompare.

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

Enter **OK** or RETURN to terminate the test parameter menu; inputs are no longer changeable. Otherwise, enter **:NN** to return to the question indicated by *NN*.

4.6 Class Descriptions

The *dev5210* test contains the seven classes of subtests listed in the following table:

Table 4–11, *dev5210* Test Classes

Class	Description
1	VBTC loopback tests
2	Formatter/drive tests
3	Write/read tests—800 bpi (NRZI)
4	Write/read tests—1,600 bpi (PE)
5	Write/read tests—6,250 bpi (GCR)
6	Exception tests
7	Miscellaneous tests

NOTE

In the previous table, Class 3, 4, and 5 all perform the same subtests but at different recording densities and parameters.

All *dev5210* subtests are loopable under the *dshell*.

4.7 Class 1 Subtests

Class 1 subtests only test the VBTC via the VBTC Diagnostic Loopback Mode; no tape drive is required. Class 1 subtests verify the following functionalities:

- Ability to communicate with the VMEbus Input/Output Processor (VIOP), VMEbus Control Unit (VBCU), and the VMEbus Tape Controller (VBTC)
- Ability of the VBTC to interrupt and to mask interrupts
- Accessibility to main memory
- Detection of FIFO parity errors

Most Class 1 subtests use the VBTC Diagnostic Loopback Mode. Diagnostic Loopback Mode is enabled by setting bit <6> in the control register at address 0x*n*01. When the Diagnostic Loopback Mode is enabled, it is verified (upon command termination) by reading the status register at address 0x*n*10. The VBTC is reset at the start of each subtest in this class (except Subtest 101) by writing to the VBTC register at address 0x*n*0F.

The following table lists all Class 1 subtests, their descriptions, and the approximate times required to execute each subtest:

Table 4–12, Class 1 Subtests

Subtest	Description	Time (min:sec)
101	VBTC Pending Register RAM Pattern Test	00:07
102	VBTC Reset Test	00:00
103	VBTC Interrupt Loopback Test	00:02
104	VBTC FIFO Pattern Test	00:14
105	VBTC FIFO Variable Size Write/Read Test	01:46

The times listed are approximate. The time required to execute each subtest depends on the amount of memory, size of the tape, density, and the velocity of the tape drive.

4.7.1 Subtest 101, VBTC Pending Register RAM Pattern Test

Subtest 101 verifies the integrity of the VBTC pending level registers via pattern tests. Registers from 0x01 through 0x0E are pattern-tested using the patterns in the following table:

Table 4–13, Subtest 101 Test Patterns

Description	Hexadecimal Pattern	Rotations (iterations)
Eight zeros and eight ones	`0x00ff`	2
Walking ones	`0x0102040810204080`	8
Walking zeros	`0xfefdfbf7efdfbf7f`	8
Alternating zeros and ones	`0xaa55`	2
Two ones and a zero	`0xdb6db6`	3
Two zeros and a one	`0x249249`	3
Two zeros and two ones	`0xcc993366`	4

Each byte of each pattern is written to and read from each RAM location. For example, the eight zeros and eight ones pattern (0x00ff) is written to and read from each RAM location once. Then it is rotated (0xff00), rewritten, and read. Each rotation is written, read, and compared twice.

The subtest does not write the start flag or the reset flag. It requires that the VBTC decode addresses on the bus.

4.7.2 Subtest 102, VBTC Reset Test

Subtest 102 verifies that multiple VBTC resets result in the same controller state. This subtest is the first to actually execute a command. This subtest first resets the VBTC, then executes a *NOP*

command in the Diagnostic Loopback Mode. After the *NOP* command completes (i.e., the controller interrupts), the register state of the controller is saved. The *NOP* command is supervised by a 1-second timeout. The above sequence is then repeated, and the ending register state is partially compared with the original register state. Differences are reported if they occur.

4.7.3 Subtest 103, VBTC Interrupt Loopback Test

Subtest 103 verifies that the VBTC command completion interrupts work. This subtest first resets the VBTC, then verifies that the VBTC can generate a 68020 interrupt on the VIOP. The interrupt capability is tested at the selected VBCU interrupt level and at 68020 interrupt level 3. The subtest also verifies that the interrupt enable bit works by de-asserting it and expecting the driver to receive a timeout error while executing a command. A *NOP* command, in Diagnostic Loopback Mode, is used for all commands.

4.7.4 Subtest 104, VBTC FIFO Pattern Test

Subtest 104 verifies the VBTC FIFO via pattern tests in the Diagnostic Loopback Mode. Initially, the subtest resets the VBTC. Since the VBTC FIFO cannot be filled (i.e., capacity is 1,024 bytes) in the Diagnostic Loopback Mode, the subtest writes and reads half of the FIFO (512 bytes) with each pattern rotation twice. Due to the way the FIFO works, this tests all FIFO RAM locations. The patterns in the following table are used:

Table 4–14, Subtest 104 Test Patterns

Description	Hexadecimal Pattern	Rotations (iterations)
Eight zeros and eight ones	`0x00ff`	2
Walking ones	`0x0102040810204080`	8
Walking zeros	`0xfefdfbf7efdfbf7f`	8
Alternating zeros and ones	`0xaa55`	2
Two ones and a zero	`0xdb6db6`	3
Two zeros and a one	`0x249249`	3
Two zeros and two ones	`0xcc993366`	4

Each byte of each pattern is written to and read from each RAM location. For example, the eight zeros and eight ones pattern (0x00ff) is written to and read from each RAM location once. Then it is rotated (0xff00), rewritten, and read. Each rotation is written, read, and compared twice.

4.7.5 Subtest 105, VBTC FIFO Variable Size Write/Read Test

Subtest 105 verifies the VBTC FIFO via variable-length reads and writes. The subtest first resets the VBTC, then writes and reads varying block sizes in the Diagnostic Loopback Mode from both

even and odd addresses. During reads, the controller transfer count is set to 512 bytes more than the current size of the written data. At the end of the read, the VBTC is expected to have a residual transfer count of 512. In addition, data compares occur on the read data and the following 512 bytes. The 512-byte overflow buffer beyond the read data is expected to contain the initial buffer fill pattern. The subtest tries all transfer sizes from one to 957 bytes (even and odd addresses).

4.8 Class 2 Subtests

Class 2 contains subtests not specific to tape recording density. No data is actually written to the tape (i.e., the digital loopback write command *LWR* is used).

The following table lists all Class 2 subtests, their descriptions, and the times required to execute each subtest:

Table 4–15, Class 2 Subtests

Subtest	Description	Time (min:sec)
200	Formatter Accessibility Test	00:00
201	Formatter/VBTC Interface Test	00:02
203	Formatter Looped Write to Read Test	00:03
204	VBTC Piped Write Test	00:04

The times listed are approximate. The time required to execute each subtest depends on the amount of memory, size of the tape, density, and the velocity of the tape drive.

4.8.1 Subtest 200, Formatter Accessibility Test

Subtest 200 verifies that the formatter is accessible and can execute a command. This subtest uses a *NOP* command and attempts to execute the simplest formatter command available to verify interconnect to the tape drive/formatter. According to the *STC 2920 Series Maintenance Manual* (StorageTek Interface Specification), a *NOP* results only in the formatter asserted and then deasserting BUSY. No status lines should be changed.

4.8.2 Subtest 201, Formatter/VBTC Interface Test

Subtest 201 verifies the general interface to the formatter by using the *DMS, NOP* command sequence available on the STC 2920 Series drives. It verifies that many of the discrete status lines from the drive are physically connected and readable by the VBTC. Refer to the *STC 2920 Series Maintenance Manual* (Chapter 4) for more information.

> **NOTE**
>
> Subtest 201 is only supported on the STC 2920 Series drives.

The following table lists the status lines that are tested:

Table 4–16, Tested STC Status Lines

TESTED STC INTERFACE SIGNALS				
NOP #	**Description**	**Mnemonic**	**Connector Pin Number**	**Software Detectable**
1	Online status	ONLS	J6-49	Yes
2	Identification burst status	ID BRST	J6-25	No
3	File protect status	FPTS	J7-27	Yes
4	Rewinding status	REWS	J7-59	Yes
5	Expecting data status	RECV	J6-23	No
6	Operation incomplete status	OP INC	J6-27	Yes
7	Online status	ONLS	J6-49	Yes
8	Tape mark status	TMS	J6-31	Yes
9	Command reject	REJ	J6-33	Yes
10	Overrun status	OVRNS	J6-35	Yes
11	Data check status	DATA CHK	J6-37	Yes
12	EPROM error status	ROMPS	J6-39	Yes
13	Corrected error status	CRERR	J6-41	Yes
14	Online status	ONLS	J6-49	Yes
15	High density status	HDNS	J6-45	No
16	Ready status	RDYS	J6-53	Yes
17	Write status	WRTS	J6-55	Yes
18–53	MUX bytes (0–3) all 9 bits	ERRMX <0..7,P>	J6-<1,3,5,..17>	Yes

4.8.3 Subtest 203, Formatter Looped Write to Read Test

Subtest 203 verifies the data bus interface to the formatter. This subtest first rewinds the tape to BOT, then writes ten 16-Kbyte blocks via the *LWR* command. This subtest verifies the digital data path between the drive analog circuitry and VBTC.

4.8.4 Subtest 204, VBTC Piped Write Test

Subtest 204 verifies that the VBTC can operate in a pipelined mode. This subtest rewinds the tape to BOT, then uses the *LWR* command to verify that the VBTC can pipeline two commands. Twenty *LWR* commands are pipelined with a transfer size of 16 Kbytes.

4.9 Class 3, 4, and 5 Subtests

Class 3, 4, and 5 subtests verify that the drive can write and read data to its tape. All tape motion commands are verified. Where applicable, Class 3, 4, and 5 subtests verify the correct state of the drive prior to the actual test logic being executed. A *NOP* command is issued at the start of each test to select the correct unit. In the event the drive is in an incorrect state, the subtest pauses and the operator is requested to intervene and place the drive online.

Each subtest pauses for up to 30 minutes while waiting for the drive to come online and ready. Once the drive is online and ready, the *CLR* command is issued, followed by a check of the formatter status registers for errors. Most subtests rewind the tape at this point.

> **NOTE**
>
> This diagnostic allows a maximum of 5 minutes for the rewind to complete.

Read and write subtests take advantage of both current tape position and controller state. This implies the position of the tape and the controller status is known at most times. Initially, if the tape position is not known, the tape is rewound, and the VBTC is reset, if necessary. Written tape blocks contain address information for verification of positional coordinates.

The following table illustrates the layout of the tape header written at the start of each block:

Table 4–17, Tape Header Layout

Field Description	Number of Bytes
Current block number	2
Number of blocks in this file	2
Current file number	2
Total number of files present	2
Size of this block	4
Subtest that wrote this data	2
Checksum of header	2
Total	16

Class 3, 4, and 5 tests perform the same subtests but at different densities as follows:

- **3**—Write/read tests (800 bpi (NRZI))
- **4**—Write/read tests (1,600 bpi (PE))
- **5**—Write/read tests (6,250 bpi (GCR))

Table 4–18, "Class 3, 4, and 5 Subtests," lists all Class 3, 4, and 5 subtests, their descriptions, and the time required for each to execute. The times listed are approximate. The time required to execute each subtest depends on the amount of memory, size of the tape, density, and the velocity of the tape drive.

These subtests are numbered from 300–599 (n00–n99). Subtest numbers in these three classes contain a prefix of *n* (as the hundreds digit) that is used to select tape density. The 300 series subtests are for 800 bpi (NRZI) tapes, the 400 series for 1,600 bpi (PE) tapes, and the 500 series for 6,250 bpi (GCR) tapes.

Table 4–18, Class 3, 4, and 5 Subtests

Subtest	Description	Time (min:sec)
*n*00	ID Burst Write Test	00:02
*n*01	ID Burst Read Test	00:03
*n*02	Write Block Test	00:17
*n*03	Read Block Fwd/Bwd Test	01:06
*n*04	Skip Block Fwd/Bwd Test	01:20
*n*05	Erase Gap Test	00:17
*n*06	Write Tape Mark Test	00:12
*n*07	Skip Tape Mark Test	00:14
*n*08	Write Block/Tape Mark Test	00:25
*n*09	Read Block/Tape Mark Fwd/Bwd Test	01:22
*n*10	Skip Block/Tape Mark Fwd/Bwd Test	01:07
*n*11	Write File UNIX Style Test	00:38
*n*12	Read UNIX Style Written File Test	00:48
*n*13	Write Variable Size Blocks Test	00:22
*n*14	Read Variable Size Blocks Test	00:29
*n*20	Chain Mode Write Test	00:23
*n*21	Chain Mode Read Test	00:44
*n*50	Write/Read All Velocities Comb. Test	01:28

[1] *n* varies from 3 to 5: 3 is 800 bpi; 4 is 1,600 bpi; and 5 is 6,250 bpi. The diagnostic defaults to all densities specified in the */mnt/bin/lib/DBtapefmt* file.

4.9.1 Subtest *n*00, ID Burst Write Test

Subtest *n*00 verifies that the drive under test can write an ID burst. This subtest first rewinds the tape to BOT. It then writes a tape mark and verifies that no errors occur and that the correct recording density is selected. This subtest verifies that the ID burst was successfully written because any write from BOT causes the ID BURST, for the current recording density, to be written and verified (read after write).

4.9.2 Subtest *n*01, ID Burst Read Test

Subtest *n*01 verifies that the drive under test can read an ID burst. This subtest rewinds the tape to BOT and then issues a Read Forward (*RDF*) command, expecting to transfer zero bytes (i.e., a tape mark is detected). In addition, the test verifies that the density reflects the assumed ID burst that subtest *n*00 wrote.

4.9.3 Subtest *n*02, Write Block Test

Subtest *n*02 rewinds the tape, then writes 200 fixed-length tape blocks from BOT. The block size is the size specified in the Test Parameter Menu. No tape marks are written during the subtest.

4.9.4 Subtest *n*03, Read Block Fwd/Bwd Test

Subtest *n*03 verifies simple reads by reading the 200 blocks that Subtest *n*02 wrote. It rewinds the tape to BOT, issues 200 *RDF* commands, and issues 200 Read Backward (*RDB*) commands. Data compares occur at the end of each read pass.

4.9.5 Subtest *n*04, Skip Block Fwd/Bwd Test

Subtest *n*04 verifies that the Forward Skip Block (*FSB*) and the Backward Skip Block (*BSB*) commands can skip over tape blocks. This subtest skips over the data that Subtest *n*02 wrote. The subtest first rewinds the tape. After each seek from block to block (using the necessary count of either the *FSB* or *BSB* command), a Read Forward (*RDF*) command is issued, and the tape header is checked to verify that a seek error (tape positioning error) has not occurred. The following list contains the order in which the subtest skips blocks and the number of the blocks that it skips:

- Skip forward (further each time) to the following blocks:
  - 2, 4, 8, 14, 22, 32, 64, 128, 200
- Skip backward (further each time) to the following blocks:
  - 198, 196, 192, 186, 178, 168, 136, 72, 1
- Skip forward then backward (sawtooth-like pattern) to the following blocks:
  - 90, 102, 92, 104, 94, 106, 96, 108, 98, 110, 100
- Skip backward then forward (sawtooth-like pattern) to the following blocks:
  - 110, 98, 108, 96, 106, 94, 104, 92, 102, 90, 100
- Skip backward and forward (accordion-like pattern) to the following blocks:
  - 130, 70, 120, 80, 110, 90, 106, 94, 104, 96, 102, 98

4.9.6 Subtest *n*05, Erase Gap Test

Subtest *n*05 verifies that the Erase Gap (*ERG*) command can erase the tape. This subtest begins by rewinding the tape. It then writes 10 blocks and verifies them by reading backwards all 10 blocks. It then positions the tape just beyond the first block with a Backward Space File (*BSF*) command. Next, it erases the second tape block and then rewinds the tape. Since the *ERG* command is supposed to erase approximately 3.5 inches of tape, the length of the blocks written is

slightly more than this. The size is, therefore, based on the current recording density. Table 4–19, "Erase Gap Sizes," lists the different erase gap sizes.

Table 4–19, Erase Gap Sizes

Density	Block Sizes (inches)
800 bpi	5.12 inches (4 Kbytes)
1,600 bpi	5.12 inches (8 Kbytes)
6,250 bpi	5.24 inches (32 Kbytes)

Block 1 should not be affected by the erase. Block 2 should be unreadable after the erase. This subtest first reads and verifies Block 1. Next, it tries to read the erased block 2. It is possible to obtain two different outcomes. First, a tape read error may result (most likely). Second, the read may complete successfully if the third block is actually read instead of the second (rare possibility).

4.9.7 Subtest *n*06, Write Tape Mark Test

Subtest *n*06 verifies that multiple tape marks can be written. This subtest first rewinds the tape, then writes one block followed by 200 tape marks and a second block. Both blocks are the size of a tape block header since they are only used to verify tape position.

4.9.8 Subtest *n*07, Skip Tape Mark Test

Subtest *n*07 verifies that multiple tape marks (200 with no data between them) can be skipped over correctly and that the tape marks are written correctly. This subtest uses the data that *n*06 wrote. It first rewinds the tape to BOT, reads the first block, and executes 200 Forward Skip File (*FSF*) commands. Next, the subtest attempts to read the second block. Tape position is verified by checking the second block's header.

4.9.9 Subtest *n*08, Write Block/Tape Mark Test

Subtest *n*08 verifies that a combination of blocks and file marks can be written. This subtest writes 10 files of 25 blocks each, with a tape mark separating each file. An end-of-tape (EOT) mark is written after the last tape mark.

4.9.10 Subtest *n*09, Read Block/Tape Mark Fwd/Bwd Test

Subtest *n*09 verifies that the combination of blocks and file marks that subtest *n*08 wrote is readable. It uses the *RDF* and *RDB* commands to read all blocks and tape marks and perform a data compare. The data is first read forward and a data compare performed. Next, the data is read backward and another data compare is performed.

4.9.11 Subtest *n*10, Skip Block/Tape Mark Fwd/Bwd Test

Subtest *n*10 verifies that the Forward Skip Block (*FSB*), Backward Skip Block (*BSB*), Forward Skip File (*FSF*), and the Backward Skip File (*BSF*) commands execute properly. This subtest skips over the data that subtest *n*08 wrote. It first rewinds the tape. After each seek from point to point (using the appropriate combination of the Skip commands), a Read Forward (*RDF*) command is issued and the tape header checked to verify that a seek error (tape positioning error) has not occurred. The following list contains the file and block coordinates reached after each skip operation:

- Skip forward (further each time) to {file, block}
  - {2, 1}
  - {5, 1}
  - {10, 1}
- Skip backward (further each time) to {file, block}
  - {8, 24}
  - {4, 24}
  - {1, 24}
- Skip backward and forward (accordion-like pattern) to {file, block}
  - {2, 10}
  - {1, 2}
  - {9, 23}
  - {7, 24}
  - {4, 1}
  - {10, 25}

4.9.12 Subtest *n*11, Write File UNIX Style Test

Subtest *n*11 verifies that a file can be written as the ConvexOS driver writes files. This subtest writes several files using the same command sequence as the ConvexOS driver. The sequence for each file is:

1. *N* write-blocks
2. Write-tape mark
3. Write-tape mark
4. Backspace-file

Asynchronous I/O is used for the entire command stream to the CCU driver. The number of blocks per file and the file count varies with the recording density. Table 4–20 lists the block count and file count:

Table 4–20, Subtest *n*11 Write File/Block Count

Density	Block Count (*N*)	File Count
800 BPI	8	8
1,600 BPI	16	16
6,250 BPI	50	20

4.9.13 Subtest *n*12, Read UNIX Style Written File Test

Subtest *n*12 verifies that the file written by subtest *n*11 can be read. The subtest initially rewinds the tape, then reads all tape blocks and marks that subtest *n*11 wrote.

4.9.14 Subtest *n*13, Write Variable Size Blocks Test

Subtest *n*13 verifies that variable-length tape blocks can be written. This subtest writes varying block sizes from one byte to 258 bytes, in increments of one, and then -2 through +2 around every base 2 boundary up to 2^{16} (64 Kbytes). Tape headers are only written if the current block size is large enough to contain the entire header.

4.9.15 Subtest *n*14, Read Variable Size Blocks Test

Subtest *n*14 verifies that the variable length tape blocks can be read. This subtest reads varying block sizes of data that subtest *n*13 wrote from one byte to 258 bytes, in increments of one, and then -2 through +2 around every base 2 boundary up to 2^{16} (64 Kbytes). Tape headers are only read and verified if the current block size is large enough to contain the entire header.

4.9.16 Subtest *n*20, Chain Mode Write Test

Subtest *n*20 verifies that chain mode write operations execute properly. It initially rewinds the tape, then writes one long gapless tape block using the VBTC data chaining mode. The chain interval size is 32 Kbytes. Normally the CCU driver would default to 64-Kbyte intervals (128-Kbyte windows). The actual size of the block written by the test is controlled by the size of the "Large Tape Block Size" specified in the "Test Parameter Menu". The size is reduced, if necessary, to the maximum size that can be written in the current density on the currently mounted tape spool.

4.9.17 Subtest *n*21, Chain Mode Read Test

Subtest *n*21 verifies that chain mode reads execute properly. This subtest reads the data that subtest *n*20 wrote and performs a data compare. Refer to subtest *n*20 for a description of the subtest algorithm.

4.9.18 Subtest *n*50, Write/Read All Velocities Comb. Test

Subtest *n*50 writes 50 blocks delimited by two tape marks in each velocity and reads back each written group in every other available velocity.

4.10 Class 6 Subtests

Class 6 subtests verify that the VBTC and tape formatter can detect and recover from a variety of error conditions. The initial setup at the start of each test is the same as that described for the Class 3, 4, and 5 subtests (i.e., check for online, *CLR*, and rewind). The following table lists all Class 6 Subtests, their descriptions, and the times required to execute each subtest.

Table 4–21, Class 6 Subtests

Subtest	Description	Time (min:sec)
600	VBTC Read Underrun Test	00:08
601	VBTC/Drive Forced Parity Error Test	00:03
602	VBTC Chain Mode Missed Interrupt Test	00:10
603	VBTC Chain Mode Read Underrun Test	00:05
604	VBTC Pipe Mode Error Test	00:06
606	VBTC Bus Error Recovery Test	00:23
608	Chain Mode Interval Boundary Write Test	02:10
609	Chain Mode Interval Boundary Read Test	02:20
610	Chain Mode Error Test	00:10

The times listed are approximate. The time required to execute each subtest depends on the amount of memory, size of the tape, density, and the velocity of the tape drive.

4.10.1 Subtest 600, VBTC Read Underrun Test

Subtest 600 verifies that the VBTC detects a read underrun condition and correctly handshakes for and drops the remaining data. This subtest begins by writing three 16-Kbyte tape blocks from BOT. It then writes two delimiting tape marks and rewinds the tape. Each of the three blocks is read back with transfer counts less than their size, which should cause read underrun. At the end of each read, the subtest verifies that only a read underrun is reported by the VBTC (i.e., no formatter-reported errors). Although no data compare occurs, the subtest verifies the read tape block headers. After the read of the third block, the subtest attempts to read across the two trailing tape marks.

4.10.2 Subtest 601, VBTC/Drive Forced Parity Error Test

Subtest 601 verifies that parity errors are detectable. This subtest initially rewinds the tape to BOT, then writes four 4-Kbyte blocks using the *LWR* command with good parity. The subtest then repeats the write command sequence with bad parity forced by the VBTC. After each *LWR* command, the test checks for a drive-reported error (parity or data check). The expected error type is controlled by one of the personality bits in the drive database file

(*/mnt/bin/lib/DBtapefmt*). Next, an optional drive reset may occur (based on another one of the personality bits in the drive database file) since some drives may not clear a parity error until after a reset. Finally, the subtest uses the *CLR* command to insure that the previous error state can be cleared.

4.10.3 Subtest 602, VBTC Chain Mode Missed Interrupt Test

Subtest 602 verifies that the VBTC halts when an interrupt receives late service in the data chain mode. This subtest first rewinds the tape to BOT. It then uses the *LWR* command to verify that the VBTC halts when command chaining is active and that a chain interval interrupt is not serviced before the next interval interrupt arrives. The test begins by starting a 128-Kbyte write in data chain mode with an interval size of 16 Kbytes. It then pauses on the third interval for two seconds and expects the VBTC to halt with the appropriate error status. The process is repeated on the fourth, fifth, and sixth chain intervals.

4.10.4 Subtest 603, VBTC Chain Mode Read Underrun Test

Subtest 603 verifies that a read underrun is detected in the data chain mode under a variety of boundary conditions. This subtest rewinds the tape to BOT and then writes two 40,960-byte tape blocks (8,192 * 5) followed by two tape marks. It then rewinds the tape and issues a read with a size of 28,672, followed by a size of 36,864 and a chain interval size of 8192 or 8 Kbytes. Both reads should cause a read underrun to occur, and the VBTC should continue to read the data until the real end-of-block is detected. As a result, the only error that should be indicated in the terminating status is a read underrun. The above read sizes cause the read underrun to occur once on Buffer A and once on Buffer B. Although the data chain mode bit is already reset when read underrun occurs, this subtest still tries to assure it can be detected at the end of a data chain mode transfer (on both Buffer A and Buffer B).

4.10.5 Subtest 604, VBTC Pipe Mode Error Test

Subtest 604 verifies that the VBTC halts command pipelining when an error occurs during piped commands. This subtest rewinds the tape, then writes a 32-Kbyte block, backspaces the tape, and writes a 16-Kbyte block. This action guarantees that a read after the 16-Kbyte block is written generates an error since there is no evident division before the second block begins. Next, a *RDF, REW* command sequence is pipelined. The rewind should not execute because the *RDF* fails, and the VBTC should halt pipelining. The fact that the following *REW* command did not execute should be indicated by the Command Abort status bit being asserted by the VBTC. In addition, BOT should not be present after the failure (i.e., make sure the *REW* command was not executed).

4.10.6 Subtest 606, VBTC Bus Error Recover Test

Subtest 606 verifies that VMEbus errors are recoverable. This subtest first rewinds the tape to BOT. It then writes four 4,096-byte tape blocks with a VMEbus standard address of 0xc00000 (bits <22> and <23> set). This address is not mapped by the VBCU, and the read should result in a VMEbus bus error. This subtest verifies that the VBTC reports a bus error after each write.

4.10.7 Subtest 608, Chain Mode Interval Boundary Write Test

Subtest 608 tests VBTC data chain mode under various boundary conditions. This subtest initially rewinds the tape to BOT, then writes varying block sizes starting with 96,296 bytes (96 Kbytes - 8) up to 98312 bytes (96 Kbytes + 8). Two blocks of each size are written (one from an even address alignment and one from an odd address alignment). Tape headers are written in each block. All tape blocks are written in data chain mode with a chain interval size of 32,768 bytes (32 Kbytes). Next, the test verifies the previously written records by rewinding the tape to BOT, reading the blocks in normal mode (i.e., no data chain mode), and comparing the data after the read.

4.10.8 Subtest 609, Chain Mode Interval Boundary Read Test

Subtest 609 uses the tape block data that subtest 608 wrote. This subtest initially rewinds the tape to BOT, then attempts to to read each of the 34 records written by subtest 608. Two blocks of each size are read (one to an initial even address alignment and one to an odd address alignment). All tape blocks are read in data chain mode with a chain interval size of 32,768 bytes (32 Kbytes). An overflow area of 512 bytes is skipped after each read to insure the VBTC does not transfer more than the correct amount of data. After each read, the data, tape header, and the overflow buffers are compared against expected results. Next, the subtest rewinds the tape to BOT and reads all the record sizes again with the transfer count set to 192 Kbytes. This read terminates the controller while in chain mode. It expects the appropriate residual transfer count at the end of each. As with the previous group of reads, all data is again compared.

4.10.9 Subtest 610, Chain Mode Error Test

Subtest 610 verifies that latter versions of the VBTC do not require an STC *CLR* command when the previous command terminated with an error. Older versions of the VBTC require the drive status to be error-free before initiating a command that utilizes data chain mode. This subtest initially rewinds the tape, then writes a 32-Kbyte block, backspaces the tape, and writes a 16-Kbyte block. This action guarantees a read (after the 16-Kbyte block is written) generates an error since there is no evident division before the second block begins. Next, the subtest starts a 256-Kbyte data chain mode write, thus guaranteeing that a chain mode command can be executed when an error was present at the end of the previous command.

> **NOTE**
>
> This subtest will not execute on VBTCs that are earlier than Fab revision B, assembly revision K.

4.11 Class 7 Subtests

Class 7 consists of the tape velocity measurement subtest and the end of tape sensor subtest. Both of these subtests have initialization sequences similar to Classes 3-6. The following table lists all Class 7 Subtests, their descriptions, and the times required to execute each subtest.

Table 4–22, Class 7 Subtests

Subtest	Description	Time (min:sec)
700	Drive Tape Velocity Verification	00:39
701	Drive EOT Sensor Verification Test	01:20

The times listed are approximate. The time required to execute each subtest depends on the amount of memory, size of the tape, density, and the velocity of the tape drive.

4.11.1 Subtest 700, Drive Tape Velocity Verification

Subtest 700 verifies all available tape velocities on the current drive in all supported densities. This subtest verifies that the achieved tape velocity is within the specified tolerance (in the database file */mnt/bin/lib/DBtapefmt*). Current suggested tolerance is ± 5%. This subtest also verifies that each velocity can be selected via software control (whatever the method). This subtest starts with the lowest tape density and ends with the most dense recording format while verifying every available velocity. For each density and velocity, this subtest rewinds the tape, then writes enough data to keep the tape moving for a minimum of 4 seconds. This subtest determines the number of blocks needed to meet the 4-second time requirement. The following table lists the block sizes used for each density:

Table 4–23, Block Sizes Written

Density	Block Sizes
800 bpi	64 Kbytes
1,600 bpi	128 Kbytes
6,250 bpi	512 Kbytes

The time required to write the first block is ignored to get the tape up to speed and to write the ID burst. The time required to write the remaining blocks is measured, and the resulting tape velocity in Inches Per Second (IPS) is calculated and displayed in the following format:

```
Density ---- Velocity Nom/Meas ---- Tolerance ---- Delta
PE              50.0 /  49.8         +/-5.0%       -0.4%
PE             100.0 /  99.4         +/-5.0%       -0.6%
GCR             50.0 /  50.0         +/-5.0%        0.0%
GCR            100.0 / 100.1         +/-5.0%       +0.1%
```

4.11.2 Subtest 701, Drive EOT Sensor Verification Test

Subtest 701 verifies that EOT can be sensed by writing until EOT is detected. This subtest rewinds the tape to BOT and continues to write 8-Kbyte blocks until EOT is sensed. It is probable that the tape will go off the spool if EOT is not sensed.

4.12 Interactive Debugger

The diagnostic provides an interactive debugger that supports the ability to execute commands from a script file, which allows more flexibility in debugging. Invoke the debugger with one of the following methods:

- Use the **-d** option when invoking the diagnostic (enter **dev5210**[x] **-d**). No subtests are executed.
- The prompt "`User Debug Option Mask [0x0-0xfff,?]`" provides a bit option to force the diagnostic to enter the debugger after an error is reported. To ensure that this option is set, **0x10** must be ORed into the hexadecimal bit-pattern response for the prompt.
- Enter a ":" when in single-step mode.

Once the interactive debugger is entered, online help commands are available. By entering **help**, the following screen is displayed:

Figure 4–6, Interactive Debugger On-line Help

```
Input base specification:
    0dNN - decimal,  0xNN or NN - hexadecimal,  the default is hexadecimal

Meta-command sequences:
   ![UNIX_CMD]       - execute UNIX_CMD
   !![UNIX_CMD]      - fork a shell and execute UNIX_CMD (allows redirection)
   <FILE             - redirect input from FILE (recursive)
   <<FILE            - end input from current file and change input to FILE

Commands:
    Commands may be abbreviated as long as the abbreviation is unique.

    boot                                        - reboot the CCU driver
    buffer                                      - display main memory buffer data
    cd      [DIRECTORY]                         - change to DIRECTORY
    continue [-p]                               - continue from single step
    echo    [-n] [word1 [word2 ...]]            - write message to display
    exit                                        - exit the diagnostic
    fb      begin [end] value [incr [step]]     - fill bytes (VIOP)
    fw      begin [end] value [incr [step]]     - fill words (VIOP)
    fl      begin [end] value [incr [step]]     - fill longs (VIOP)
    help    [COMMAND ...]                       - display general or specific help
    mb      begin [end]                         - modify/[dump] bytes (VIOP)
    mw      begin [end]                         - modify/[dump] words (VIOP)
    ml      begin [end]                         - modify/[dump] longs (VIOP)
    mfb     begin [end] value [incr [step]]     - fill bytes (Main mem)
    mfw     begin [end] value [incr [step]]     - fill words (Main mem)
    mfl     begin [end] value [incr [step]]     - fill longs (Main mem)
    mmb     begin [end]                         - modify/[dump] bytes (Main mem)
    mmw     begin [end] [count]                 - modify/[dump] words (Main mem)
    mml     begin [end] [count]                 - modify/[dump] longs (Main mem)
    pause   [-n] [seconds]                      - pause for <C/R> or seconds
    quit                                        - exit debug mode
    reg                                         - display current VBTC registers
    reset                                       - reset VBTC/tape drive
    skip    [-d dbg_mask] [-emq] [count]        - set single step skip mode/count
```

In addition to the help screen in the previous figure, you can display help for a specific command by entering:

help *command*

where *command* is the desired debugger command. Abbreviations of desired commands may be used as long as they are unique. For example, to display help for all commands starting with the letter "r," enter **help r**.

4.13 Interactive Debugger Command Descriptions

4.13.1 boot

Usage: **boot**

Reboots the CCU driver.

> **WARNING**
>
> This command can cause a subtest to fail if used from within the middle of the subtest.

4.13.2 buffer

Usage: **buffer**

Displays main memory buffer address and length.

4.13.3 cd

Usage: **cd** [DIRECTORY]

Changes to another directory, where DIRECTORY is any valid directory path. If DIRECTORY is omitted, the default path is $HOME or / if $HOME not set.

4.13.4 continue

Usage: **continue** [-p]

Continues from single step after restoring previous skip mode and counter, where *-p* prints the current skip option setting and does not continue.

This command restores the previous skip count and exits the debugger. The **-p** option prints the current skip option setting and does not continue. This command is useful for continuing from a single step pause point with the same skip options that last caused the test to pause in single step mode.

4.13.5 echo

Usage: **echo** [-n] [word1 [word2 ...]]

Writes words separated by blanks and terminated by a newline to the display, where *-n* means do not echo the terminating newline character.

4.13.6 exit

Usage: **exit**

Exits the diagnostic. This command exits the entire test. If the command is executed interactively, you will be asked to confirm that you really want to exit the test.

4.13.7 fb, fl, fw

Usage: **fb begin value**
fb begin end value [incr [step]]
fb begin,count value [incr [step]]

fl begin value
fl begin end value [incr [step]]
fl begin,count value [incr [step]]

fw begin value
fw begin end value [incr [step]]
fw begin,count value [incr [step]]

Fills memory with specified pattern in byte-at-a-time mode (fb), longword-at-a-time move (fl), or word-at-a-time mode (fw), where:

- **begin** is the starting address
- **value** is the initial fill value
- **end** is the ending address
- **incr** is the fill value increment
- **step** is the address increment
- **count** is the count of elements to fill

The first format (e.g., **fb begin value**) stores *value* at address *begin*.

The second format (e.g., **fb begin end value** [incr [step]]) fills from the address *begin* up to and including address *end* with the value *value*. If the optional *incr* parameter is specified, *value* is incremented by *incr* after each fill. If *incr* is followed by *step*, the fill address is incremented by *step* elements instead of the normal step of one.

The third format (e.g., **fb begin,count value** [incr [step]]) is identical to the second with one exception: *end* is not specified. Instead, the *end* parameter is calculated from the *count* parameter.

The following examples illustrate use of these commands.

1. *ffl 2000000 12345678*

   The above command stores one longword (32 bits) of value 0x12345678 at main memory address 0x200000.

2. *ffl 200000,0d1000 0*

   The above command zeroes 1000 longword (32 bit) locations starting a main memory address 0x200000.

3. *fl ffa048,0d20 78000200 1*

   The above command maps 20 CCU main memory windows starting at window 18 to contiguous physical main memory addresses starting at address 0x200000. The map registers are set up for modes ACCEL, PBUS, 68K, and ALL-VME-SLOTS.

4. *fb 30000,0d40 10 4 2*

   The above command fills 40 even bytes starting at CCU address 0x30000 with a value that begins at ten and increments by four each time.

4.13.8 help

Usage: **help** [COMMAND ...]

Displays general or specific help information, where COMMAND is the desired debugger command. Abbreviations of desired commands may be used as long as they are unique. For example, the following command displays help for all commands starting with the letter "r":

help r

4.13.9 mb, mw, ml

Usage: **mb begin,count**
mb begin end
mb begin

mw begin,count
mw begin end
mw begin

ml begin,count
ml begin end
ml begin

Displays and/or modifies CCU address space in byte-at-a-time mode (mb), word-at-a-time mode (mw), or long-word-at-a-time mode (ml), where:

- **begin** is the starting CCU microprocessor address
- **end** is the ending CCU microprocessor address
- **count** is the number of elements to display

The first format (e.g., **mb begin,count**) displays *count* number of elements from the starting address. The second format (e.g., **mb begin end**) displays all elements from the address *begin* up to and including address *end*.

The third format (e.g., **mb begin**) enters an interactive mode that allows modification of memory. The following list gives the valid responses while in interactive mode:

[<value>]	write optional <value> to current address, advance to next address
[<value>]=	write optional <value> to current address, and stay at the present address (re-read)
[<value>]^[N]	write optional <value> to current address, move to address N (address 0 if N is omitted)
[<value>]+[N]	write optional <value> to current address, advance to the next address (N addresses if N is specified)
[<value>]-[N]	write optional <value> to current address, back up to the previous address (N addresses if N is specified)
[<value>]q	write optional <value> to current address, exit interactive mode

Multiple commands may be specified on the same line. A comma or space may be used to separate the commands or value as shown in the following example:

```
Debug mode ->     mb c03fc1
<CCU:c03fc1> = 1c 00=ff,1q
```

where **1c 00=ff,1q** is an example of executing multiple commands on the same line. This sequence modifies the byte at address 0xc03fc1 to 0, re-reads and displays the new value, modifies the byte to 0xff, skips to address 0xc03fc2 and modifies it to a 0x1, and then quits interactive mode.

4.13.10 mfb, mfw, mfl

Usage: **mfb begin value**
mfb begin end value [incr [step]]
mfb begin,count value [incr [step]]

mfl begin value
mfl begin end value [incr [step]]
mfl begin,count value [incr [step]]

mfw begin value
mfw begin end value [incr [step]]
mfw begin,count value [incr [step]]

Fills main memory with specified pattern in byte-at-a-time mode (mfb), word-at-a-time mode (mfw), or longword-at-a-time move (mfl), where:

- **begin** is the starting main memory address
- **value** is the initial fill value
- **end** is the ending main memory address
- **incr** is the fill value increment
- **step** is the address increment

The first format (e.g., **mfb begin value**) stores *value* at address *begin*. The second format (e.g., **mfb begin end value** [incr [step]]) fills from the address *begin* up to and including address *end* with the value *value*. If the optional *incr* parameter is specified, *value* is incremented by *incr* after each fill. If *incr* is followed by *step*, the fill address is incremented by *step* elements instead of the normal step of one.

The third format (e.g., **mfb begin,count value** [incr [step]]) is identical to the second with one exception: *end* is not specified. Instead, the *end* parameter is calculated from the *count* parameter.

4.13.11 mmb, mmw, mml

Usage: **mmb begin,count**
mmb begin end
mmb begin

mmw begin,count
mmw begin end
mmw begin

mml begin,count
mml begin end
mml begin

Displays and/or modifies main memory address space in byte-at-a-time mode (mmb), word-at-a-time mode (mmw), or long-word-at-a-time mode (mml), where:

- **begin** is the starting main memory address
- **end** is the ending main memory address
- **count** is the count of elements to fill

The first format (e.g., **mmb begin,count**) displays *count* number of elements from the starting address. The second format (e.g., **mmb begin end**) displays all elements from the address *begin* up to and including address *end*.

The third format (e.g., **mmb begin**) enters an interactive mode that allows modification of memory. The following list gives the valid responses while in interactive mode:

[<value>]	write optional <value> to current address, advance to next address
[<value>]=	write optional <value> to current address, and stay at the present address (re-read)
[<value>]^[N]	write optional <value> to current address, move to address N (address 0 if N is omitted)
[<value>]+[N]	write optional <value> to current address, advance to the next address (N addresses if N is specified)
[<value>]-[N]	write optional <value> to current address, back up to the previous address (N addresses if N is specified)
[<value>]q	write optional <value> to current address, exit interactive mode

Multiple commands may be specified on the same line. A comma or space may be used to separate the commands or values as shown in the following example:

```
Debug mode -> mmb c03fc1
<Main-Mem:c03fc1> = 1c 00=ff,1q
```

where **1c 00=ff,1q** is an example of executing multiple commands on the same line. This sequence modifies the byte at main memory address c03fc1 to 0, re-reads and displays the new value, modifies the byte to 0xff, skips to address 0xc03fc2 and modifies it to a 0x1, and then quits interactive mode.

4.13.12 pause

Usage: **pause** [-n] [seconds]

Wait for specified amount of time or for a RETURN if the time is omitted, where *-n* means do not echo the pause message, and *seconds* specifies the number of seconds to pause.

4.13.13 quit

Usage: **quit**

Exits the interactive debugger and continues to the next single step point, if applicable.

4.13.14 reg

Usage: **reg**

Displays current VBTC register state. This command reads the VBTC registers and displays the entire register image in an easily-readable format.

4.13.15 reset

Usage: **reset**

Resets the VBTC controller and its attached drive(s) by writing (ORing) a one into the Reset Register at address offset 0xf. It then resets the one by ANDing with 0xfe.

4.13.16 skip

Usage: **skip** [-emqrx] [-d [MASK]] [COUNT]

Displays or sets the single step modifier and skip counter, where:

-e	Skips until CCU driver returns an error (non-zero in the status field)
-m	Same as -e, except non-fatal media errors are not stopped on
-q	Disables printing while in skip mode. This will cause the test to "skip pause points" at normal execution speed since nothing will be printed (i.e., no pause messages)
-r	Resets current skip options (e.g., skip count = 0)
-x	Exits debugger after skip command
COUNT	Skips the next COUNT (or 1 if COUNT omitted) single step pause points. If a COUNT is specified with the *-e* or *-m* option, the test will skip the next COUNT errors before pausing.
-d [MASK]	Sets debug mask to MASK. If MASK is omitted, the current mask is printed. The debug bit definitions follow:

```
** USER Debug Options (bit mapped) **
0x0001: Disable Asynchronous Command Mode
0x0002: Disable Multi-Record Buffering
0x0004: Disable Reset of Controller After ^C
0x0008: Disable Hard Error Trapping (logging)
0x0010: Enter Debugger After an Error Occurs
0x0020: Disable copying of errors to error file
0x0100: Pause AFTER each CCU driver command is returned
0x0200: Show each CCU driver message AFTER it is returned
0x0400: Pause BEFORE each CCU driver message is sent
0x0800: Show each CCU driver message BEFORE it is sent
0x1000: Automatically enter debugger during a pause
```

If no options are specified, only the current skip modifier and counter are printed.

The **skip** command is useful when you want to avoid stepping through several hundred single step points (pressing RETURN each time). By stopping only on errors (*-e* or *-m*), the test does not stop at every single step point. With the *-q* option, the test does not print single step displays until it stops at one.

4.14 Error Messages

The diagnostic reports four categories of errors:

- Tape positioning errors
- Register mismatch errors
- Data compare errors
- Media errors (not fatal in most cases)

> **NOTE**
>
> There are two type of media errors. Formatter-corrected errors are media errors that were mathematically corrected by the tape formatter. These can be ignored or retried based on selected options from the test parameters. Uncorrectable media errors are errors that cannot be corrected by the formatter and must be retried or ignored.

In all cases, the diagnostic returns as much information as possible. The format of the first three types of error messages (tape positioning, register mismatch, and data compare) are basically consistent.

In the following examples, `exp` refers to the expected value, `act` refers to the actual value, `dc` refers to the insignificant bits (don't care bit mask). Also, `diff` is the difference between the actual value and the expected value excluding the insignificant bits. The `diff` value is obtained by ANDing the actual value with the complement of the insignificant bits (don't care bit mask) and exclusive ORing the result with the expected value. All bits that are in discrepancy are set. For example:

```
exp/act/dc = 0x0000/0x0100/0x0000, diff = 0x0100
```

The following figures show examples of all four types of messages:

Figure 4–7, Tape Positioning Error Example

```
*****  Thu May 31 14:42:21 1990  *****
Test:    dev5210.t  1.1    Class: 5    Subtest: 504 1.1   Count: 1    Error: 7
Failed:  GCR: Skip Block Fwd/Bwd Test

        ------------------- trace point: 504.115.20 -------------------
Tape Position:      File 1 of 1, Block 14 of 200
Current Action:     Verifying the positional coordinates of the tape are as
                    expected.
Other Info:         Error occurred while skipping in the forward direction from
                    block 9 to block 14.  There were 3 successful skips
                    completed prior to the error.
Error Description: The read tape block header indicates a tape positioning
                    error has occurred (i.e., a seek error).
Tape Block Header Mismatch:
          Block-num Block-cnt File-num  File-cnt  Block-size    Wrt-Subtest
exp/act = 000e/000f 00c8/00c8 0001/0001 0001/0001 002000/002000 502
```

NOTE

In the following example, the bytes that miscompare are marked by an asterisk (*).

Figure 4–8, Data Compare Error Example

```
*****  Thu May 31 14:52:43 1990  *****
Test:    dev5210.t  1.1    Class: 1    Subtest: 105 1.1   Count: 1    Error: 7
Failed:  VBTC FIFO Variable Size Write/Read Test

       -------------------- trace point: 105.125 --------------------
Iteration:          Loop 1 of 1914
Current Action:     Comparing the buffer fill pattern with the data beyond the
                    expected read area.
Other Info:         The failure occurred with a transfer size of 1 and began on
                    an EVEN address.  Transfer sizes from 1 through 957 were
                    being tested on both EVEN and ODD addresses.
Error Description: Data compare error occurred.
                    1 data compare error(s) occurred.  The first was at byte
                    offset 1 (0x1) in the data block under compare.  The data
                    block begins at main memory address: 0x02002000 and is 1
                    bytes in length.  It is in NORMAL order.  The error was in
                    the 512 byte overflow space (at byte offset 0 (0x0) beyond
                    the end of the read data).
Expected/Actual:
    02002001: aa/33* 55/55  aa/aa  aa/aa  55/55  aa/aa  55/55  55/55
    02002009: aa/aa  55/55  aa/aa  aa/aa  55/55  aa/aa  55/55  55/55
    02002011: aa/aa  55/55  aa/aa  aa/aa  55/55  aa/aa  55/55  55/55
    02002019: aa/aa  55/55  aa/aa  aa/aa  55/55  aa/aa  55/55  55/55
```

Figure 4–9, Register Mismatch Error Example

```
*****  Thu May 31 14:56:43 1990  *****
Test:    dev5210.t  1.1    Class: 2    Subtest: 201 1.1   Count: 1    Error: 7
Failed:  Formatter/VBTC Interface Test

       -------------------- trace point: 201.125 --------------------
Current Action:     Attempting to execute a tape formatter NOP command.
Other Info:         The STC Mnemonic of the interface signal that was under
                    test was "FPTS".  This signal is located on connector J7,
                    pin 57.
                    This failure occurred at NOP number 3 of the DMS/NOP
                    command sequence.  See the STC 2920 series maintenance
                    manual (Ch 4) for more info.
Error Description: The VBTC register state at completion of the last command
                    did not match the expected register state.
VBTC Register mismatches:
       3fda(rtsb):  exp/act = 0x02/0x81, dc = 0x9c
Extra bits(rtsb):   0x1<Rew>
Missing bits(rtsb): 0x2<Wrt_Prt>
```

Figure 4–10, Media Error Example

```
                    ---- TEST MEDIA ERROR TOTALS ----

Media Error Class ---- Total ---- NOT Recovered ---- Recovered ---- Ignored
Formatter-Corrected       13                 0              0           13
Uncorrectable              0                 0              0            0
```

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

T

U

V

W

CONVEX VMEbus STC Tape Controller
(*dev5210*) Diagnostics Manual
Document No. 760-003130-000
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._____

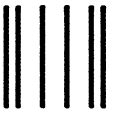
FOR ADDITIONAL INFORMATION OR DOCUMENTATION:

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

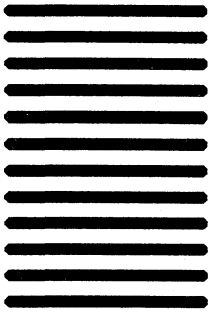
Direct mail orders to:

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)