CONVEX Tape Library Interface Subsystem (*tli4480*) Diagnostics Manual

Document No. 760-003730-000

First Edition
May 1991

CONVEX Computer Corporation
Richardson, Texas USA

CONVEX Tape Library Interface Subsystem
(tli4480) Diagnostics Manual
Order No. DHW-249
First Edition

Printed in the United States of America

Revision Sheet

CONVEX Tape Library Interface Subsystem (tli4480) Diagnostics Manual

Edition	Document No.	Date	Description
First	760-003730-000	May 1991	First release. Contains the *tli4480* diagnostic test information from the *CONVEX PBUS I/O Systems Diagnostics Manual.*

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

1 Diagnostics Environment

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

2 EGOS Overview

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

3 Dshell Overview

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

4 Tape Library Interface Subsystem Test (*tli4480*)

4.1 Overview 4-1
4.2 Required Equipment 4-1
4.3 Test Invocation 4-2
4.4 Test Parameter Menu 4-5
4.4.1 Prompt Explanations 4-7
4.5 Internal Initialization Sequence 4-9
4.6 Class Descriptions 4-10
4.7 Class 0 Subtests 4-11
4.7.1 Subtest 1, Board Reset Test 4-12
4.7.2 Subtest 2, Slot Verification Test 4-13
4.7.3 Subtest 3, EEPROM Write Protection Test 4-13
4.7.4 Subtest 100, Data RAM Bit Functionality Test 4-13
4.7.5 Subtest 110, Data RAM Column Functionality Test 4-13
4.7.6 Subtest 120, Data RAM Uniqueness Test 4-13
4.7.7 Subtest 130, Data RAM Parity Test 4-13
4.7.8 Subtest 200, Instruction RAM Bit Functionality Test 4-13
4.7.9 Subtest 210, Instruction RAM Column Functionality Test 4-14
4.7.10 Subtest 220, Instruction RAM Uniqueness Test 4-14
4.7.11 Subtest 230, Instruction RAM Parity Test 4-14
4.7.12 Subtest 240, Instruction RAM Execution Test 4-14
4.7.13 Subtest 300, PMAP RAM Bit Functionality Test 4-14
4.7.14 Subtest 310, PMAP RAM Column Functionality Test 4-14
4.7.15 Subtest 320, PMAP RAM Uniqueness Test 4-14
4.7.16 Subtest 330, PMAP RAM Parity Test 4-15
4.7.17 Subtest 400, PBUS Header Generation Test 4-15
4.7.18 Subtest 410, PBUS Access Test 4-15
4.7.19 Subtest 500, Data Buffer Port 0 Bit Functionality Test 4-15

4.7.20 Subtest 501, Data Buffer Port 0 Column Functionality Test 4-15
4.7.21 Subtest 502, Data Buffer Port 0 Uniqueness Test 4-16
4.7.22 Subtest 503, Data Buffer Port 0 Parity Test 4-16
4.7.23 Subtest 510, Data Buffer Port 1 Bit Functionality Test 4-16
4.7.24 Subtest 511, Data Buffer Port 1 Column Functionality Test 4-16
4.7.25 Subtest 512, Data Buffer Port 1 Uniqueness Test 4-16
4.7.26 Subtest 513, Data Buffer Port 1 Parity Test 4-17
4.7.27 Subtest 604, DPED 0 Column Functionality Test 4-17
4.7.28 Subtest 605, DPED 0 Parity Test 4-17
4.7.29 Subtest 614, DPED 1 Column Functionality Test 4-17
4.7.30 Subtest 615, DPED 1 Parity Test 4-17
4.7.31 Subtest 700, PIGA Bit Functionality Test 4-17
4.7.32 Subtest 701, PIGA Column Functionality Test 4-17
4.7.33 Subtest 702, PIGA Uniqueness Test 4-18
4.7.34 Subtest 800, Data RAM Protection Test 4-18
4.8 Class 1 Subtests 4-18
4.8.1 Subtest 1000, Channel Reset Test 4-19
4.8.2 Subtest 1010, Channel Bus-Out Test 4-19
4.8.3 Subtest 1011, Channel Out-Tag Test 4-19
4.8.4 Subtest 1012, Channel Bus-In Test 4-19
4.8.5 Subtest 1013, Channel In-Tag Test 4-19
4.9 Class 2 Subtests, Manufacturing Support Board Tests 4-19
4.9.1 Subtest 2000, Channel 0 Bus-On-Bus Loopback Test 4-20
4.9.2 Subtest 2010, Channel 0 Tag-On-Bus Loopback Test 4-20
4.9.3 Subtest 2020, Channel 1 Bus-On-Bus Loopback Test 4-20
4.9.4 Subtest 2030, Channel 1 Tag-On-Bus Loopback Test 4-20
4.10 Class 4 Subtests, Tape Motion Read/Write Device Tests 4-20
4.10.1 Subtest 4000, Tape Mark Test 4-21
4.10.2 Subtest 4010, Space Record Test 4-21
4.10.3 Subtest 4020, Erase Tape Test 4-22
4.10.4 Subtest 4030, Long Block Read Test 4-22
4.10.5 Subtest 4040, Fixed Record Size Read/Write Test 4-22
4.10.6 Subtest 4050, Write File UNIX-Style Test 4-23
4.10.7 Subtest 4060, Variable Size Records Test 4-23
4.10.8 Subtest 4100, Extended Status Test 4-25
4.11 Interactive Commands 4-25
4.11.1 ! 4-26
4.11.2 banner 4-27
4.11.3 base 4-28
4.11.4 -c 4-29
4.11.5 debug 4-30
4.11.6 flags 4-30
4.11.7 help 4-30
4.11.8 init 4-31
4.11.9 log 4-32
4.11.10 man 4-32
4.11.11 Prt_err 4-32
4.11.12 quit 4-33
4.11.13 -s 4-33
4.11.14 save 4-34
4.11.15 set 4-34
4.11.16 state 4-36
4.11.17 trace 4-36

4.11.18 trout 4-37
4.12 Interactive Debugger 4-37
4.13 Interactive Debugger Command Descriptions 4-39
4.13.1 ? 4-39
4.13.2 block 4-39
4.13.3 bsf 4-39
4.13.4 bsr 4-39
4.13.5 cd 4-39
4.13.6 connect 4-40
4.13.7 display 4-40
4.13.8 echo 4-40
4.13.9 erase 4-40
4.13.10 exit 4-40
4.13.11 fb, fl, fw 4-41
4.13.12 ffb, ffl, ffw 4-42
4.13.13 fsf 4-42
4.13.14 fsr 4-42
4.13.15 help 4-42
4.13.16 identify 4-43
4.13.17 iu 4-43
4.13.18 mb, mw, ml 4-43
4.13.19 mmb, mmw, mml 4-44
4.13.20 pause 4-44
4.13.21 quit 4-44
4.13.22 rewind 4-45
4.13.23 rphys 4-45
4.13.24 status 4-45
4.13.25 unitclr 4-45
4.13.26 unload 4-45
4.13.27 weof 4-45
4.13.28 wphys 4-45

Appendixes

A Reporting Problems

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

List of Tables

1–1 Test Program Categories 1-2
1–2 Test Program Types 1-2
1–3 Test Program Device Types 1-3
1–4 Example Test Program Names 1-3
3–1 *dshell* Commands 3-2
4–1 Hardware Requirements 4-2
4–2 *tli4480* Symbolic Links 4-3
4–3 Getting Help During Test Parameter Entry 4-6
4–4 *tli4480* Test Classes 4-10
4–5 Class 0 Data Pattern 4-11
4–6 Class 0 Subtests 4-12
4–7 Class 1 Subtests 4-18
4–8 Class 2 Subtests 4-20
4–9 Class 4 Subtests 4-21
4–10 Subtest 4040 Data Pattern 4-23
4–11 Subtest 4060 Record Sizes 4-24
4–12 Interactive Commands 4-26
4–13 Settable Constant Values 4-35

List of Figures

2–1 EGOS' Position in the Environment 2-3
3–1 Syntax Help for the *loop* Command 3-3
4–1 Initial Test Invocation Sequence 4-4
4–2 Test Parameter Menu 4-5
4–3 Sample Test Parameter Summary 4-6
4–4 Interactive Debugger Online Help 4-38

Preface

Purpose and Intended Audience

This manual explains how to run the *tli4480* diagnostic, which verifies the operation of a Tape Library Interface (TLI). This document is not a tutorial, but rather a reference for the users of the *tli4480* 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 *tli4480* 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. Tape Library Interface Subsystem Test (*tli4480*)**—Describes how to operate the diagnostic, including prerequisites, test invocation, internal initialization sequence, and class descriptions. It also describes interactive commands and the interactive debugger.
- **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-249.
The document number for this manual is 760-003730-000.

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

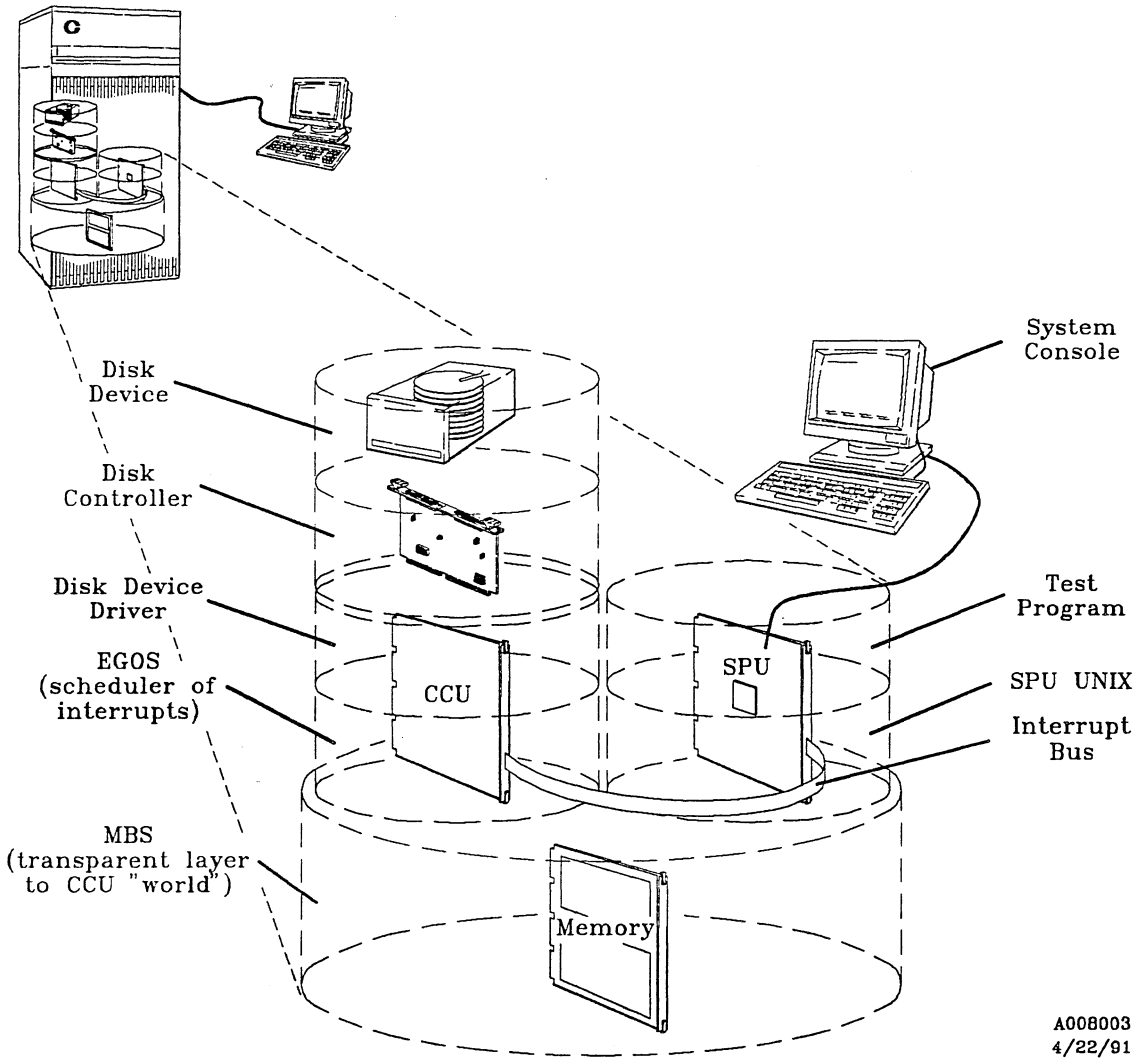
The VME interface version of EGOS is designed with a scheduler for the VIOP and is called VIOP EGOS. VIOP EGOS supports event driven device drivers as well as process type device drivers. VIOP EGOS utilizes a *sleep/wakeup* type of process control that improves efficiency of the device driver and makes 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
Tape Library Interface Subsystem Test (*tli4480*)

4.1 Overview

tli4480 is a diagnostic test suite for the CONVEX Tape Library Interface (TLI) subsystem. This diagnostic supports the StorageTek 4480 (3 Mbyte/sec and 4.5 Mbyte/sec) and the IBM 3480 (3 Mbyte/sec) tape drives. The TLI subsystem consists of:

- TLI block-multiplexer interface Channel Control Unit (CCU)
- Internal cabling from the CCU to the CONVEX external block-multiplexer connectors
- Downloaded CCU driver software

Specifically, *tli4480* accomplishes the following:

- Verifies that the TLI CCU is electrically sound in that its microprocessor can execute instructions from its ROM, access its data RAM, and access the PBUS
- Verifies that the normal path CCU driver firmware can be downloaded, probe all ports, attach to all control units, and connect to all tape drives
- Verifies that the normal path software can communicate with the StorageTek 4480 and IBM 3480 control units and perform basic read/write/verify tape operations
- Verifies that the TLI subsystem is functional, reliable, and capable of supporting ConvexOS
- Provide an interactive debugger that can execute commands from a script file.

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

4.2 Required Equipment

Table 4–1 lists the required hardware for the TLI subsystem:

Table 4–1, Hardware Requirements

C200 Series
Memory System[1]
CPX
SP2
VIOP
VBCU (Rev D or later)
Tape Library Interface subsystem

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

4.3 Test Invocation

The *tli4480.t* diagnostic is unique as a CONVEX diagnostic in two ways:

- First, it is a dedicated CCU diagnostic; it does not access a chassis controller such as a VMEbus SCSI controller.
- Second, it attempts to address the diverse needs of its three primary users (manufacturing, development/integration, and field engineering).

The net result is a single diagnostic that encompasses board, device, system checkout, and interactive control tests in one object body. This combined diagnostic can be invoked from the Diagnostic Shell (*dshell*), as a standalone test, or in an interactive mode.

The three methods of invocation serve three purposes:

- For those users familiar with the *dshell* environment, operation of this diagnostic will be no different from any other *dshell*-compatible diagnostics. All *dshell* features, such as looping, test flags, etc., are supported.
- For integration purposes the diagnostic can be executed in a standalone mode to get a go/no-go determination of the subsystem operation.
- Manufacturing personnel will find the control and logging features necessary to assist them in the production and debugging of the CCU.

Invocation control is provided by creating symbolic links to the same object file, *tli4480.t*. Each link also has an associated script file, *tli4480*.scr*. Upon invocation, the diagnostic determines the name it was called under and then looks for a file of the same name with a *.scr* extension. If such a file exists, the diagnostic submits that file to the debugger's script file processor for execution. Upon completion, the diagnostic is terminated. This technique provides for updating

the diagnostic without recompiling the source, as well as allowing for a more user-configurable test. For example, the equivalent of a board-test diagnostic can be created with any of the following procedures:

1. Enter the command `ln tli4480.t tli4480b.t`

   This command links the diagnostic to a *b*, or board, test description.

2. Create a file *tli4480b.scr*, containing the following:

```
echo "tli4480b.scr - BOARD TEST"
echo "execute all class 0 test"
* -c 0
echo "all done"
```

3. Call *dshell*, then enter : `test tli4480b`.

   This command executes the diagnostic, submitting the script file to the script processor. The output would be similar to the following:

```
tli4480b.scr - BOARD TEST
execute all class 0 test
st0001 etc 0 Scan reset of tli & eprom checksum
  .
  .
  .
st0100 etc 6 Data ram bit functionality
  .
  .
  .
st0702 etc 2 piga uniqueness all done
```

Table 4–2 lists the symbolic links and associated script files included in the *tli4480* diagnostic:

Table 4–2, *tli4480* Symbolic Links

Symbolic Link	Associated Script File	Type of Test
tli4480b.t	*tli4480b.scr*	Board tests
tli4480d.t	*tli4480d.scr*	Device tests
tli4480i.t	----------	Interactive mode
tli4480f.t	*tli4480f.scr*	Fast, or integration checkout, tests

Refer to the section entitled "Interactive Debugger" for more information on the creation of acceptable script files. Any valid debugger, interactive, or UNIX command can be executed from a script file, which means that the device under test can be switched, logging can be turned on or off, tracing can be enabled, etc.

The format for the *test* command under *dshell* is:

test tli4480 [*option* ...]

where *option* is one of the following:

-c *class-number*	Execute one or more specific classes of subtest(s)
-d	Go to debug mode only; no tests are executed
-f *filename*	Use *filename* as the parameter save file. If this option is omitted, the parameter file used is */tmp/tli4480.tmp*.
-i	Go to interactive mode only; no tests are executed
-s *subtest-number*	Execute one or more individual subtests
-V	Print the version (build date) of this test

To invoke the *tli4480* test, use the procedure shown in Figure 4–1. All responses in **boldface** are entered by the user.

> **NOTE**
>
> Use the following test invocation sequence for the initial invocation of *tli4480* 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)> initall RETURN
(spu)> dshell RETURN
:test tli4480 [-c [class number(s)]] [-s [subtest number(s)]] [-option] [-f FILE] [+>filename] RETURN
```

In actual usage, all the prompts and responses appear sequentially on the screen, one line at a time. All prompts and responses are shown here in one figure for convenience.

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

Entering only **test tli4480** executes all *tli4480* subtests sequentially.

4.4 Test Parameter Menu

Once the test is invoked, a test menu prompt is presented allowing selection of default switches. Figure 4–2 shows all prompts, their possible responses (in brackets [ ]), and their default responses (in parentheses ( )). The prompts and responses 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.

Figure 4–2, 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: CCU number (slot) of TLI to test [0-15,?]                (0) -> RETURN
 2: TLI Port to test [0-1,?]                                 (0) -> 0
 3: Tape Control Unit (CU) to test [0-15,?]                  (0) ->
 4: Tape drive to test [0-15,?]                              (0) -> RETURN
 5: xfr rate 0[4.5mb/x] 1[3mb/s] [0-1,?]                     (0) -> RETURN
 6: device: [0]StorageTek] 1[IBM] [0-1,?]                    (0) -> RETURN
 7: Use Defaults for Remaining Parameters [y,n,?]            (y) -> n
 8: Trace messages [0-1,?]                                   (0) -> RETURN
 9: Trace output control (screen/file) [0-1,?]               (0) -> RETURN
10: Driver: 0[scan], 1[diag], 2[normal path] [0-2,?]         (2) -> RETURN
11: Auto menu:  0[off], 1[on] [0-1,?]                        (1) -> RETURN
12: Auto banner:  0[off], 1[on] [0-1,?]                      (1) -> RETURN
13: Error Reporting Verbosity [0-5,?]                        (0) -> RETURN
14: <cr> req'd for continue. 1[yes] 0[no] [0-1,?]            (1) -> RETURN
15: auto-logging 1[on] 0[off] [0-1,?]                        (1) -> RETURN
16: max control unit to auto-attach [0-15,?]                 (4) -> RETURN
17: max tape number to auto-connect [0-15,?]                 (4) -> RETURN
18: Subtests results file: on[1], off[0] [0-1,?]             (0) -> RETURN
19: Ticker tock: on[1], off[0] [0-1000,?]                    (0) -> RETURN
20: Auto probe: on[1], off[0] [0-1,?]                        (1) -> RETURN
21: Enable Debug Monitor [y,n,?]                             (n) -> RETURN
22: Variable record subtest pattern  [<hexadecimal pattern>,?]
                                                        (0x6db6)   -> RETURN
23: Variable record miscompare line dump count [1-100,?]     (8) -> RETURN

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

24: Select Printer On/Off Mode [0x0-0x3,?]                  0x0) -> 0x3
25: Enter OK, or :NN to return to question NN [OK]          (OK) -> RETURN
```

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

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

Table 4–3, 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.

After all parameters have been entered, the diagnostic displays a TEST PARAMETER SUMMARY listing the prompts that were answered and their responses. Figure 4–3 illustrates an example of a TEST PARAMETER SUMMARY screen. The actual values and responses vary according to the input.

Figure 4–3, Sample Test Parameter Summary

```
                    TEST PARAMETER SUMMARY

CCU number (slot) of TLI to test                          : 0
TLI Port to test                                          : 0
Tape Control Unit (CU) to test                            : 0
Tape drive to test                                        : 0
xfr rate 0[4.5mb/s] 1[3mb/s]                              : 0
device: 0[StorageTek] 1[IBM}                              : 0
Use Defaults for Remaining Parameters                     : n
Trace messages                                            : 0
Trace output control (screen/file)                        : 0
Driver: 0[scan], 1[diag], 2[normal path]                  : 2
Auto menu:  0[off], 1[on]                                 : 1
Auto banner:  0[off], 1[on]                               : 1
Error Reporting Verbosity                                 : 0
<cr> req'd for continue. 1[yes] 0[no]                     : 1
Auto-logging 1[on] 0[off]                                 : 1
max control unit to auto-attach                           : 4
max tape number to auto-connect                           : 4
Subtests results file: on[1], off[0]                      : 0
Ticker tock: on[1], off[0]                                : 0
Auto probe: on[1], off[0]                                 : 1
Enable Debug Monitor                                      : n
Variable record subtest pattern                           : 0x6db6
Variable record miscompare line dump count                : 8
Select Printer On/Off Mode                                : 0x0000
Enter OK, or :NN to return to question NN                 : OK
```

4.4.1 Prompt Explanations

The following paragraphs repeat and explain the test parameter prompts.

```
1: CCU number (slot) of TLI to test [0-15,?]          (3) -> RETURN
```

Enter the slot number of the TLI board that you wish to test.

```
2: TLI Port to test [0-1,?]                           (1) -> 0
```

Enter the number of the port that you want to test.

```
3: Tape Control Unit (CU) to test [0-15,?]            (0) -> 2
```

Enter the number of the tape control unit that you want to test.

```
4: Tape drive to test [0-15,?]                        (0) -> RETURN
```

Enter the number of the tape drive that you want to test.

```
5: xfr rate 0[4.5mb/s] 1[3mb/s] [0-1,?]               (0) -> RETURN
```

Enter the transfer rate for data being moved.

```
6: Device: [0]StorageTek] 1[IBM] [0-1,?]              (0) -> RETURN
```

Enter the type of tape control unit that you want to test.

```
7: Use Defaults for Remaining Parameters [y,n,?]      (y) -> n
```

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.

```
8: Trace messages [0-1,?]                             (0) -> RETURN
```

Enter **1** if you want MBS message tracing to be enabled. Otherwise, the messages will not be saved.

```
9: Trace output control (screen/file) [0-1,?]         (0) -> RETURN
```

Enter **0** if you want the trace messages displayed on the screen; otherwise, enter **1** if you want to save them to the log file.

```
10: Driver: 0[scan], 1[diag], 2[normal path] [0-2,?]  (2) -> RETURN
```

Enter the option that indicates which driver tests you want to execute, where

- 0 supports EPROM or scan-based level 0 board diagnostics
- 1 supports level 1 RAM-based diagnostics
- 2 supports normal path MBS-based diagnostics

```
11: Auto menu:  0[off], 1[on]  [0-1,?]                    (0) -> RETURN
```

Enter **1** to automatically display the menu after a <CR> is entered.

```
12: Auto banner:  0[off], 1[on] [0-1,?]                   (0) -> RETURN
```

Enter **1** to automatically update the banner after a <CR> is entered.

```
13: Error Reporting Verbosity [0-5,?]                     (0) -> RETURN
```

Enter the level of error reporting you want, where *0* is no error reporting and *5* is everything. This option controls the amount of interactive responses you have with the diagnostics test.

```
14: <cr> req'd for continue. 1[yes] 0[no] [0-1,?]         (1) -> n
```

Enter **1** to indicate that a <CR> is required to continue the subtest after an error occurs. Otherwise, the subtest will end when an error occurs.

```
15: auto-logging 1[on] 0[off] [0-1,?]                     (1) -> RETURN
```

Enter **1** to indicate that logging is to happen automatically. Otherwise, logging will be disabled.

```
16: max control unit to auto-attach [0-15,?]              (4) -> RETURN
```

Enter the total number of control units to automatically attach to the TLI.

```
17: max tape number to auto-connect [0-15,?]              (4) -> RETURN
```

Enter the total number of tape drives to automatically connect to the TLI.

```
18: Subtests results file: on[1], off[0] [0-1,?]          (0) -> RETURN
```

Enter **1** to store manufacturing test results to a file for future reference.

```
19: Ticker tock: on[1], off[0] [0-1000,?]                 (0) -> RETURN
```

Enter **1** to update time at a user prompt.

```
20: Auto probe: on[1], off[0] [0-1,?]                     (0) -> RETURN
```

Enter **1** to automatically probe all ports, units, and tapes.

```
21: Enable Debug Monitor [y,n,?]                          (n) -> RETURN
```

Enter **y** to enable the debug monitor. Otherwise, it will be disabled. Enabling the debug monitor automatically takes you to the debugger in the event of an error.

```
22: Variable record subtest pattern  [<hexadecimal pattern>,?]
                                                    (0x6db6)  -> RETURN
```

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

```
23: Variable record miscompare line dump count [1-100,?]  (8) -> RETURN
```

Enter the maximum number of display lines to print when a miscompare occurs.

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

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

This prompt allows a specified character string to be sent to the printer before 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 subtests multiple times or for long periods of time. This assumes a printer is connected to the auxiliary port on the display terminal where *tli4480* 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).

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

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

4.5 Internal Initialization Sequence

The file *tli4480.save* is created upon the successful exit from the diagnostic. After the last prompt is entered, and before subtest code execution, the diagnostic performs the following events:

- Checks if the file *tli4480.save* was created. If so, the diagnostic reads the test parameters from that. If not, the input parameters are written to the parameter file (default or user-specified).
- Checks to see if the CCU is already loaded with the *tli4480* 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.

> **NOTE**
>
> The file */mnt/boot_db* is used to determine what CCU and memory boards are installed. If this file is non-existent, it can be created with the following command at the `(spu)>` prompt:
>
> **scn_util -b > /mnt/boot_db**

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

4.6 Class Descriptions

Table 4–4 lists the classes of subtests contained in the *tli4480* diagnostic:

Table 4–4, *tli4480* Test Classes

CLASS	DESCRIPTION
0	Scan-ring/EPROM-based board tests
1	Downloaded RAM-based Subsystem Board Tests
2	Manufacturing Support Board Tests
4	Tape Motion Read/Write Device Tests

Subtest looping may be achieved either under *dshell* or, independent of *dshell*, by passing parameter flags on the command line to the subtest. These flags may be used in either the *dshell* or interactive mode. The *-L* flag controls the class test loop count, while *-l* controls the subtest loop count. The loop count is equivalent to an inner loop, controlled by *l*, and an outer loop, controlled by *L*. For example,

<ROM> **-s 1000**

executes subtest 1000 one time.

<ROM> **-s -l2 1000**

executes subtest 1000 two times.

<ROM> **-s -l2 -L2 1000, 1100**

would be the equivalent of

```
-s -l2 1000,1100
-s -l2 1000,1100
```

and results in eight subtest invocations.

Similarly,

<ROM> **-c -L3 -l1000 0**

executes each subtest in class 0 1000 times, three times.

4.7 Class 0 Subtests

Class 0 subtests test data and program memory, along with PBUS interface logic. They verify that the TLI can be downloaded with firmware and execute out of its own memory.

These subtests are contained in an electrically-erasable, programmable read-only memory, or EEPROM.

Table 4–5 lists the data pattern used:

Table 4–5, Class 0 Data Pattern

Hexadecimal Test Pattern	
0x00000000	0xAAAAAAAA
0xffffffff	0x11111111
0x55555555	0xeeeeeeee

Table 4–6 lists all Class 0 subtests, their descriptions, and the approximate time in seconds required to execute each subtest:

Table 4–6, Class 0 Subtests

Subtest	Description	Time (sec)
1	Board Reset Test	1
2	Slot Verification Test	1
3	EEPROM Write Protection Test	1
100	Data RAM Bit Functionality Test	1
110	Data RAM Column Functionality Test	1
120	Data RAM Uniqueness Test	1
130	Data RAM Parity Test	1
200	Instruction RAM Bit Functionality Test	6
210	Instruction RAM Column Functionality Test	1
220	Instruction RAM Uniqueness Test	1
230	Instruction RAM Parity Test	1
240	Instruction RAM Execution Test	1
300	PMAP RAM Bit Functionality Test	2
310	PMAP RAM Column Functionality Test	1
320	PMAP RAM Uniqueness Test	1
330	PMAP RAM Parity Test	1
400	PBUS Header Generation Test	1
410	PBUS Access Test	1
500	Data Buffer Port 0 Bit Functionality Test	3
501	Data Buffer Port 0 Column Functionality Test	1
502	Data Buffer Port 0 Uniqueness Test	1
503	Data Buffer Port 0 Parity Test	1
510	Data Buffer Port 1 Bit Functionality Test	3
511	Data Buffer Port 1 Column Functionality Test	1
512	Data Buffer Port 1 Uniqueness Test	1
513	Data Buffer Port 1 Parity Test	1
604	DPED 0 Column Functionality Test	1
605	DPED 0 Parity Test	1
614	DPED 1 Column Functionality Test	1
615	DPED 1 Parity Test	2
700	PIGA Bit Functionality Test	2
701	PIGA Column Functionality Test	1
702	PIGA Uniqueness Test	2
800	Data RAM Protection Test	170

4.7.1 Subtest 1, Board Reset Test

Subtest 1 resets the TLI board and checks the results of the self-test. The self-test on the TLI board includes an EPROM checksum test. It also reports the EPROM code version.

This subtest uses the scan-ring to set, then release, the reset signal of the TLI microprocessor. After the reset signal is released, the processor performs a checksum on its ROM and reports its status back to its LED register. This value is scanned in and compared to determine if the operation was successful and the board is functional.

4.7.2 Subtest 2, Slot Verification Test

Subtest 2 uses the scan-ring communication capability to verify that the CCU slot matches the slot of the diagnostic. Specifically, the CCU reads its slot ID from a location on the backplane and uses a multiple of it as a main memory address for CMI (Common Message Interface) data structures. If the slot ID yields the wrong location, it will not be possible to download and communicate with the CCU via CMI.

4.7.3 Subtest 3, EEPROM Write Protection Test

Subtest 3 verifies that the local processor on the CCU cannot write its EEPROM unless it is write enabled.

4.7.4 Subtest 100, Data RAM Bit Functionality Test

Subtest 100 tests Data RAM bit functionality. It performs a true/complement pattern test for all locations in data RAM, using the patterns in Table tli4480-5. Byte, word, and longword accesses are checked.

4.7.5 Subtest 110, Data RAM Column Functionality Test

Subtest 110 tests Data RAM column functionality. It performs a walking 1s and 0s test on the first word in each bank of data RAM (walking from least significant bit to most significant bit).

4.7.6 Subtest 120, Data RAM Uniqueness Test

Subtest 120 tests data RAM location uniqueness. It writes an incrementing value to each location in the data RAM, then verifies that all locations contain the expected value. Byte, word, and longword accesses are checked for uniqueness.

4.7.7 Subtest 130, Data RAM Parity Test

Subtest 130 tests data RAM parity checking. It performs a walking 1s and 0's test, writing a location in data RAM with inverted parity and then reading it to generate a parity error. The location is rewritten with correct parity when the test is complete. All four parity bits are checked individually.

4.7.8 Subtest 200, Instruction RAM Bit Functionality Test

Subtest 200 tests instruction RAM bit functionality. It performs a true/complement pattern test for all locations in instruction RAM, using the patterns in Table tli4480-5. Only longword accesses are checked.

4.7.9 Subtest 210, Instruction RAM Column Functionality Test

Subtest 210 tests instruction RAM column functionality. It performs a walking 1s and 0s test on the first word in each bank of instruction RAM (walking from least significant bit to most significant bit).

4.7.10 Subtest 220, Instruction RAM Uniqueness Test

Subtest 220 tests instruction RAM location uniqueness. It writes an incrementing value to each location in the instruction RAM, then verifies that all locations contain the expected value. Only longword accesses are (or can be) checked for uniqueness.

4.7.11 Subtest 230, Instruction RAM Parity Test

Subtest 230 tests instruction RAM parity checking. It performs a walking 1' and 0's test, writing a location in instruction RAM with inverted parity and then reading it to generate a parity error. The location is rewritten with correct parity when the test is complete. All four parity bits are checked simultaneously (longword access).

4.7.12 Subtest 240, Instruction RAM Execution Test

Subtest 240 tests that the code in the instruction RAM can be correctly executed. First a pattern of oxfefefefe is written to IRAM, followed by a jump to it. If no instruction fault occurs, an error is reported back to the SPU. Afterwards, a relocatable subroutine is copied out of IROM into IRAM. A jump is then performed, returning back to IROM. If an instruction fault occurs, an error is reported back to the SPU.

4.7.13 Subtest 300, PMAP RAM Bit Functionality Test

Subtest 300 tests PMAP RAM bit functionality. It performs a true/complement pattern test for all locations in PMAP RAM, using the patterns in Table tli4480-5. Only longword access is checked.

4.7.14 Subtest 310, PMAP RAM Column Functionality Test

Subtest 310 tests PMAP RAM column functionality. It performs a walking 1s and 0s test on the first word in each bank of PMAP RAM (walking from least significant bit to most significant bit).

4.7.15 Subtest 320, PMAP RAM Uniqueness Test

Subtest 320 tests PMAP RAM location uniqueness. It writes an incrementing value to each location in PMAP RAM, then verifies that all locations contain the expected value. Only longword accesses are checked for uniqueness.

4.7.16 Subtest 330, PMAP RAM Parity Test

Subtest 330 tests that the parity checking is working for the PMAP registers. The first PMAP register is written with a zero and bad parity. The 88100 then reads the PMAP register and verifies that it generated a data fault and that the Fault Store Register (FSR) logs an IBUS fault. The process is repeated with a pattern of 0xfefefefe (odd number of bits in each byte).

> **NOTE**
>
> A window access on a PMAP register with bad parity is not detected by the TLI. The TLI depends on the memory system (PBI) to detect the error. Verification that this condition is properly processed is outside the scope of the level 0 test. Also, the PMAP registers are word accessible only.

4.7.17 Subtest 400, PBUS Header Generation Test

Subtest 400 verifies that the PMAP translation and PBI header generation are working properly. The PMAP is first initialized with a translation map, and the PBI is then set to test mode via the Diagnostic Control Register (DCR). The 88100 processor then attempts a write to main memory (the transaction is terminated via test mode). The 88100 processor then reads the upper then lower 32 bits of the PB2LBREG, simulating the header. This value is then verified by the value written to the PMAP, and the size of the transfer. Miscompare will result in failure, reporting back to the host both the expected and the found values.

4.7.18 Subtest 410, PBUS Access Test

Subtest 410 reads, verifies, and echos a pattern placed in main memory by the SPU. This test verifies that the SPU and the TLI agree on the location of main memory and of the proper byte ordering in that memory.

4.7.19 Subtest 500, Data Buffer Port 0 Bit Functionality Test

Subtest 500 tests the bit functionality of the data buffer for Port 0. It performs a true/complement pattern test for all locations in the data buffer, using the patterns in Table tli4480-5. Only longword access is checked.

4.7.20 Subtest 501, Data Buffer Port 0 Column Functionality Test

Subtest 501 tests the column functionality of the data buffer for Port 0. It performs a walking 1s and 0s test on the first word in each bank of the data buffer (walking from least significant bit to most significant bit).

4.7.21 Subtest 502, Data Buffer Port 0 Uniqueness Test

Subtest 502 tests the location uniqueness of the data buffer for Port 0. It writes an incrementing value to each location in the data buffer, then verifies that all locations contain the expected value. Only longword accesses are checked for uniqueness.

4.7.22 Subtest 503, Data Buffer Port 0 Parity Test

Subtest 503 tests that the parity checking is working for the data buffer for Port 0. The first data buffer register is written with a zero and bad parity. The 88100 processor then reads the data buffer register and verifies that it generated a data fault, and that the FSR logs an IBUS fault. The process is repeated with a pattern of 0xfefefefe (odd number of bits in each byte).

> NOTE
>
> A window access on a PMAP register with bad parity is not detected by the TLI. The TLI depends on the memory system (PBI) to detect the error. Verification that this condition is properly processed is outside the scope of the level 0 test. Also, the PMAP registers are word accessible only.

4.7.23 Subtest 510, Data Buffer Port 1 Bit Functionality Test

Subtest 510 tests the bit functionality of the data buffer for Port 1. It performs a true/complement pattern test for all locations in the data buffer, using the patterns in Table tli4480-5. Only longword access is checked.

4.7.24 Subtest 511, Data Buffer Port 1 Column Functionality Test

Subtest 511 tests the column functionality of the data buffer for Port 1. It performs a walking 1s and 0s test on the first word in each bank of the data buffer (walking from least significant bit to most significant bit).

4.7.25 Subtest 512, Data Buffer Port 1 Uniqueness Test

Subtest 512 tests the location uniqueness of the data buffer for Port 1. It writes an incrementing value to each location in the data buffer, then verifies that all locations contain the expected value. Only longword accesses are checked for uniqueness.

4.7.26 Subtest 513, Data Buffer Port 1 Parity Test

Subtest 513 tests that the parity checking is working for the data buffer for Port 1. The first data buffer register is written with a zero and bad parity. The 88100 processor then reads the data buffer register and verifies that it generated a data fault, and that the FSR logs an IBUS fault. The process is repeated with a pattern of 0xfefefefe (odd number of bits in each byte).

4.7.27 Subtest 604, DPED 0 Column Functionality Test

Subtest 604 tests the column functionality of the DPED Basic Control Register at address ff0090 for Port 0. It performs a walking 1s and 0s test on the lower 30 bits of the four-byte word (walking from least significant bit to most significant bit).

4.7.28 Subtest 605, DPED 0 Parity Test

Subtest 605 tests that the parity checking is working for the DPED 0. The first data buffer register is written with a zero and bad parity. The 88100 processor then reads the data buffer register and verifies that it generated a data fault, and that the FSR logs an IBUS fault. the process is repeated with a pattern of 0xfefefefe (odd number of bits in each byte).

4.7.29 Subtest 614, DPED 1 Column Functionality Test

Subtest 614 tests the column functionality of the DPED Basic Control Register at address ff1090 for Port 1. It performs a walking 1s and 0s test on the lower 30 bites of the four-byte word (walking from least significant bit to most significant bit).

4.7.30 Subtest 615, DPED 1 Parity Test

Subtest 615 tests that the parity checking is working for the DPED 1. The first data buffer register is written with a zero and bad parity. The 88100 processor then reads the data buffer register and verifies that it generated a data fault, and that the FSR logs an IBUS fault. The process is repeated with a pattern of 0xfefefefe (odd number of bits in each byte).

4.7.31 Subtest 700, PIGA Bit Functionality Test

Subtest 700 tests PIGA bit functionality. It performs a true/complement pattern test for all locations in the PIGA, using the patterns in Table tli4480-5. Byte, word, and longword accesses are checked.

4.7.32 Subtest 701, PIGA Column Functionality Test

Subtest 701 tests PIGA column functionality. It performs a walking '1s and '0's test on the first word in each bank of PIGA (walking from least significant bit to most significant bit).

4.7.33 Subtest 702, PIGA Uniqueness Test

Subtest 702 tests PIGA location uniqueness. It writes a incrementing value to each location in the PIGA, then verifies that all locations contain the expected value. Byte, word, and longword accesses are checked for uniqueness.

4.7.34 Subtest 800, Data RAM Protection Test

Subtest 800 tests Data RAM read/write protection. It writes values to every page in the data RAM, then verifies that all locations contain the expected value. This procedure is then repeated twice, first with the data RAM read protection enabled, then with the data RAM write protection enabled. In both of these cases, proper error detection is verified.

4.8 Class 1 Subtests

Class 1 subtests test all internal CCU diagnostic hardware logic and performs additional CCU-specific board tests.

> **CAUTION**
>
> Run these tests with no tape devices connected to the block-multiplexer interface. Otherwise, unexpected results may occur.

Table 4–7 lists all Class 1 subtests, their descriptions, and the approximate time in seconds required to execute each subtest:

Table 4–7, Class 1 Subtests

Subtest	Description	Time (sec)
1000	Channel Reset Test	2
1010	Channel Bus-Out Test	1
1011	Channel Out-Tag Test	1
1012	Channel Bus-In Test	1
1013	Channel In-Tag Test	1

4.8.1 Subtest 1000, Channel Reset Test

> **NOTE**
>
> This subtest should be performed before any Class 1 subtest.

Subtest 1000 reads both block mux channel microsequencers. It reloads the control stores and verifies the correct ending channel status.

4.8.2 Subtest 1010, Channel Bus-Out Test

Subtest 1010 writes a pattern of walking 1s across both channels' Bus-out register and verifies the pattern by reading the Loop In Register (LIR).

4.8.3 Subtest 1011, Channel Out-Tag Test

Subtest 1011 writes a pattern of walking 1s across both channels' Bus-In Register (BIR) and verifies the pattern via loopback logic.

4.8.4 Subtest 1012, Channel Bus-In Test

Subtest 1012 writes a pattern of walking 1s to the Bus In Register via loopback logic and verifies the pattern by reading the BIR.

4.8.5 Subtest 1013, Channel In-Tag Test

Subtest 1013 writes a pattern of walking 1s to the loopback logic and verifies the pattern by reading the In-Tag.

4.9 Class 2 Subtests, Manufacturing Support Board Tests

Class 2 subtests are designed specifically for manufacturing and field support. They require the connection of a special external loopback connector (CONVEX part numbers 601-180001-200 [bus] and 601-160002-200 [tag]). They are highly interactive and produce test results data that can be stored in the log file.

For each subtest, the diagnostic prompts the user to attach the loopback connecter.

Table 4–8 lists all Class 2 subtests, their descriptions, and the approximate time in seconds required to execute each subtest:

Table 4–8, Class 2 Subtests

Subtest	Description	Time (sec)
2000	Channel 0 Bus-On-Bus Loopback Test	2
2010	Channel 0 Tag-On-Bus Loopback Test	2
2020	Channel 1 Bus-On-Bus Loopback Test	2
2030	Channel 1 Tag-On-Bus Loopback Test	8

4.9.1 Subtest 2000, Channel 0 Bus-On-Bus Loopback Test

Subtest 2000 writes a pattern of walking 1s to the Bus-Out signal on Channel 0 and verifies the pattern by reading the Bus-In signal.

4.9.2 Subtest 2010, Channel 0 Tag-On-Bus Loopback Test

Subtest 2010 writes a pattern of walking 1s to the Tag-Out signal on Channel 0 and verifies the pattern by reading the Bus-In signal.

4.9.3 Subtest 2020, Channel 1 Bus-On-Bus Loopback Test

Subtest 2020 writes a pattern of walking 1s to the Bus-Out signal on Channel 1 and verifies the pattern by reading the Bus-In signal.

4.9.4 Subtest 2030, Channel 1 Tag-On-Bus Loopback Test

Subtest 2030 writes a pattern of walking 1s to the Tag-Out signal on Channel 1 and verifies the pattern by reading the Bus-In signal.

4.10 Class 4 Subtests, Tape Motion Read/Write Device Tests

Class 4 subtests verify tape motion and the ability to read and write data to the tape. Class 4 subtests use the standard Common Message Interface (CMI) command set.

The first four subtests do not perform any data verification checks. A status check is made prior to all commands to ensure that the tape is online and ready. Before any command using forward motion is issued, a check is made to ensure that the physical end of the tape has not been reached. For all write commands, a check is made to ensure that the write protect has not been set on the tape media. A Beginning-of-Tape (BOT) status bit set check is done for every *REWIND* command to check completion status. The *REWIND* command will not be sent to the tape driver if the tape media is already at BOT.

Each subtest has no dependency on prior execution of other subtests. The execution of the subtests in sequential order will check the tape subsystem in order of increasing complexity.

Table 4–9 lists all Class 4 subtests, their descriptions, and the approximate time in seconds required to execute each subtest:

Table 4–9, Class 4 Subtests

Subtest	Description	Time (sec)
4000	Tape Mark Test	044
4010	Space Record Test	217
4020	Erase Tape Test	054
4030	Long Block Read Test	044
4040	Fixed Record Size Read/Write Test	076
4050	Write File UNIX-Style Test	049
4060	Variable Size Records Test	024
4100	Extended Status Test	005

4.10.1 Subtest 4000, Tape Mark Test

Subtest 4000 verifies that a tape mark can be written to the tape and detected. The subtest first verifies that a written tape mark can be detected in the forward direction. Next, 100 tape records are written, each followed by a tape mark, . *SPACE FILE* commands are then issued in both the forward and reverse directions. Tape position is then verified by testing for the presence or absence of a tape mark. The following is an example of a failed Tape Mark Test:

```
Error: Expected tape mark status did not occur
     : while fwd space file
```

4.10.2 Subtest 4010, Space Record Test

Subtest 4010 verifies the *FORWARD SPACE RECORD* and the *BACK SPACE RECORD* commands. The subtest begins by rewinding the tape and writing two records followed by a tape mark. The subtest then writes 20 records of decreasing size to the tape, the biggest of these records being 1,000 bytes and the smallest being 50 bytes. A tape mark is then written to the tape. A space record test is then performed between the end points denoted by the tape marks. The subtest then issues a varying series of *FORWARD SPACE RECORD* commands. The subtest verifies the success of the space record test by checking for the presence of a filemark. The testing sequence is repeated using the *BACK SPACE RECORD* command. The subtest then repeats the testing sequence with a combination of *FORWARD SPACE* and *BACK SPACE* commands. The following is an example of a failed Space Record Test:

```
Error: Unexpected tape mark status sensed
     : while fwd space rec
```

4.10.3 Subtest 4020, Erase Tape Test

Subtest 4020 verifies that an *ERASE* command used in a write recovery, due to write errors, does not leave glitches on the tape. The subtest writes 10 records of 4,096 bytes each to the tape. Each record, in turn, is erased and the remaining records are reconstituted to make a total of 10 records in the file again. The integrity of the file is checked by spacing over the records. The following is an example of a failed Erase Test:

```
Error: Expected tape mark status did not occur
     : while backspace rec
```

4.10.4 Subtest 4030, Long Block Read Test

Subtest 4030 verifies that the long-block status bit indicates correctly whether a long block is present. This test begins by writing a series of records correctly on the tape, starting with a record of four bytes with each subsequent record being twice the size of the previous record. The sixteenth and last record written to the tape is 128 Kbytes long. Next, four read passes are used to read the records, test the long-block status bit, and verify that the number of bytes transferred was correct.

The first read pass has the number of bytes to be transferred equal to the record size on the tape. A check is then made to ensure that the long-block status bit is not set.

The second pass reads each record with the requested transfer size as two bytes less than the record size on the tape. A check is made to ensure that the long-block status bit is set and the number of bytes transferred is equal to the number requested.

The third pass is the same as the second pass except that the requested transfer size is one byte less than the record size on the tape. The results of this pass are odd byte counts in the data transfer size requests.

The fourth and final pass reads each record, except the 128-Kbyte record, with the requested transfer size equal to 0x20000, the maximum number of bytes allowed by the tape drive. A check is made to ensure that the long-block status bit is checked for a reset state and that the number of bytes transferred is not equal to the number of bytes requested. The following is an example of a failed Long Block Read Test:

```
Error: Expected long block read status did not occur
     :from long_xblk_test
```

4.10.5 Subtest 4040, Fixed Record Size Read/Write Test

Subtest 4040 writes and reads 160 records of a fixed size. The first write pass writes twenty 16-Kbyte records to the tape. Each of these records contains a different data pattern. On subsequent write passes, the record size is increased by 16 Kbytes until the record size reaches 128 Kbytes. The tape is then rewound and each record is read back and verified. Table 4–10 lists the data pattern used:

Table 4–10, Subtest 4040 Data Pattern

Hexadecimal Test Pattern			
00000000	ffffffff	a5a5a5a5	5a5a5a5a
f0f0f0f0	0f0f0f0f	cc33c3c3	99669696
01010101	02020202	04040400	8080808
10101010	20202020	40404040	80808080
fefdfbf7	efdfbf7f	0fa5c396	12487edb

The following is an example of a failed Fixed Record Size Read/Write Test:

```
Error: data miscompare
     : buf adr 00000000 exp 00000000 act 12487edb
     : while reading from tape
```

4.10.6 Subtest 4050, Write File UNIX-Style Test

Subtest 4050 simulates the sequence of commands that are given to the tape driver by the ConvexOS driver. A file of 8 records, each of 32 Kbytes, is written to tape. Two filemarks are written to the tape, and a *BACKSPACE FILEMARK* command is issued. Seven more files are written in the same manner. The data in each record is set equal to the record number. The end result is 64 numbered records on the tape with a filemark after every 8 records. The last record ends with two filemarks. The tape is rewound. All records are read and verified and all filemarks are verified to be in the right locations on the tape. The following is an example of a failed Write File UNIX-Style Test:

```
Error: Expected tape mark status did not occur
     : while verifying last tapemark on tape
```

4.10.7 Subtest 4060, Variable Size Records Test

Subtest 4060 verifies that variable length records on a tape can be written, read, and verified. First, records of 1 byte to 258 bytes (incremented by 1 byte) are written to the tape. Next, sets of 5 records each are written to the tape. The sizes of these records are designed to cross base 2 boundaries. With the exception of the 1-byte record, the first 2 bytes of each record contain the record number. Byte 3 and greater all contain a data pattern that is either defaulted or is user-specified from the keyboard or from a file. The default pattern is 0x6db6. The default number of error printouts is 8, but can be set up to 100 error printouts by the user. The total number of miscompares in a record are totaled for the printout. There are a total of 298 records written and verified with the size of the records ranging from 1 byte to 65,538 bytes in this subtest. Table 4–11 lists the record sizes for Subtest 4060:

Table 4–11, Subtest 4060 Record Sizes

Starting Size	Number of Records	Record Sizes	Record Number
1	258	1 to 258	1 to 258
510	5	510 to 514	259 to 263
1,022	5	1,022 to 1,026	264 to 268
2,046	5	2,046 to 2,050	269 to 273
4,094	5	4,094 to 4,098	274 to 278
8,190	5	8,190 to 8,194	279 to 283
16,382	5	16,382 to 16,386	284 to 288
32,766	5	32,766 to 32,770	289 to 293
65,534	5	65,534 to 65,538	294 to 298

The following is an example of a failed Variable Records test:

```
Error in record 292, record length=32769 bytes
Error: data byte miscompared
     : buf adr = 00000008  exp = 6d  act = dd
     : buf adr = 00000009  exp = b6  act = bb
     : buf adr = 00000f01  exp = b6  act = bb
     : buf adr = 00000f02  exp = 6d  act = 6f
     : buf adr = 00002012  exp = 6d  act = 66
     : buf adr = 00002013  exp = b6  act = b7
     : buf adr = 00002014  exp = 6d  act = 7d
     : buf adr = 00004022  exp = 6d  act = 6f
     : Total miscompares = 10
     : while comparing data read from tape
  0:0030     failed
*****  Sat May 12 15:23:30 1990  *****
Test:    tli4480.t  1.1  Class: 3     Subtest: 306 1.1  Count: 1   Error: 0
Failed:  Variable size records read/write test
```

4.10.8 Subtest 4100, Extended Status Test

Subtest 4100 verifies that the tape drive can properly report extended status information. After requesting and receiving extended status from the driver, the test verifies certain bytes within the extended status. These bytes are:

Byte 7 expected values: 0x19 or 0x29

Byte 32 expected value: 0xFF

Byte 33 expected value: 0x34

Byte 34 expected value: 0x80

Byte 35 expected values: 0x11 or 0x22

Byte 36 expected value: 0x34

Byte 37 expected value: 0x80

Byte 38 expected values: 0x11 or 0x22

Byte 47 expected values: 0x21 or 0x30

4.11 Interactive Commands

tli4480 provides commands that can be invoked interactively from the <ROM> prompt. Table 4–12 lists each command followed by a synopsis of its meaning:

Table 4–12, Interactive Commands

Command	Meaning
!	Executes UNIX command
banner	Displays multiline status information
base	Sets numeric base for input data
-c	Executes class test(s)
debug	Enters interactive debugger
flags	Displays or sets the current test flags
help	Displays help menu or file
init	Downloads and initializes CCU
log	Writes comments to the log file
man	Displays online extended diagnostic help
Prt_err	Displays error code information
quit	Exits interactive diagnostic
-s	Executes subtest(s)
save	Saves current configuration in *tli4480.save*
set	Changes settable control constant
state	Displays active state, status, and link count
trace	Enables message tracing
trout	Controls trace output

The following sections describe each command in detail and give examples of its meaning.

4.11.1 !

Provides a shell hook to execute UNIX operating system commands without exiting the diagnostic.

The format for this command is

! *command*

where *command* is the UNIX command to execute.

The following example illustrates the use of this command.

!vi file Edits the file *file* with the *vi* editor. Upon exiting the editor, control would be returned to the diagnostic.

4.11.2 banner

Clears the screen and displays a multiline information banner across the top of the screen.

The format for this command is

banner

The banner is in the form of:

```
name version    ctrl_desc     devname        ctrl_mfg                date
    ccu[no.][status] port[no.][status] unit[no.][status] tape[no.][status]
state: CURSTATE trace: STATE to: WHERE verb: LEVEL print: STATE dmode: STATE
```

where

name	Is the name of the diagnostic
version	Is the build/release version
ctrl_desc	Is the controller description
devname	Is the tape formatter/controller name
ctrl_mfg	Is the tape formatter/controller manufacturer
date	Is the current date
ccu *no*	Is the selected CCU under test
ccu *status*	Indicates whether the CCU has been loaded; options are *loaded* or blank
port *no*	Is the selected CCU port or channel under test
port *status*	Indicates whether the port has been probed; options are *probed* or blank
unit *no*	Is the selected control unit under test
unit *status*	Indicates whether the unit is attached; options are *attached* or blank
tape *no.*	Is the selected tape drive under test
tape *status*	Indicates whether the tape is connected; options are *connected* or blank
state *curstate*	Is the initialization level of CCU/diagnostic, where *curstate* can be

ROM	currently communicating via scan-ring.
RAM	currently communicating via MBS/CMI, but cannot communicate with the selected control unit. Have downloaded the driver and probed the ports, but could not attach to the control unit.
NORMAL PATH	currently communicating with the CCU via MBS/CMI, have probed the ports, attached to the selected control unit, and connected to the desired tape drive
DEBUG	currently using the interactive debugger, or have downloaded the diagnostic driver and have probed, attached, and connected.

trace *state* Indicates whether tracing is enabled; options are *on* or *off*. When tracing ie enabled, all CMI/MBS messages between the SPU and the CCU will be collected, interpreted, and directed to the location specified by *to*.

to *where* Indicates the destination of the output from the trace command, where *where* can be

FILE all trace output will be stored in the file *tli.trace*. *tli.trace* is closed and reopened upon every invocation of a subtest in order to limit its size.

SCREEN all trace output will be directed to the screen.

verb Is the current verbosity level as set in the initial parameter selection or the *set -s verb* command. *verb* ranges from 0–5, where the higher the level, the more verbose the error and logic flow reporting is. Verbosity level 5 is reserved for software debug, and subtests will not be executed.

print Controls the attached printer. When enabled via the initial parameter selection or the *set -s printer 1* command, all error messages will be printed.

dmode Controls automatically jumping to the interactive debugger in the event of an error. It is enabled with the *initall* parameter selection or the *set -s dmode 1* command.

4.11.3 base

Sets the numeric base for all input used as test parameters.

The format for this command is

base [*option*]

where *option* is one of the following:

b binary

d ten, or decimal

o octal

t ten, or decimal

x hexadecimal

4.11.4 -c

Executes an entire class of tests, with looping provided.

The format for this command is

-c -[*option*] [*clno*]

where *option* is one of the following:

c Controls subtest continued execution

h Displays help on the indicated class in the format

stno	subtest number
clno	the class to which the subtest belongs
et	estimated or approximate run length in seconds
description	text description of the subtest or class
ticker	number of seconds expired since invocation
L	Controls the number of times that each test in the class is executed
l	Controls the number of times that each subtest is executed before proceeding on to the next subtest

and *clno* is the class number(s) to execute. *clno* can be specified in any of the following formats:

N	a specific class number
N,N,...N	a list of specific class numbers
N-N	a range of class numbers, with the specific beginning and ending classes numbers included

The following examples illustrate the use of this command.

-c	No test would be executed. A list of all class tests and their descriptions would be presented.
-c 0-4	Executes all subtests in classes 0,1,2,3, and 4.
-c -L10 -l10 0,1,2 -c	Executes each subtest in class 0 ten consecutive times, then repeat the process ten times, then repeat the whole sequence again for all subtests in class 1 and 2. If each class had 10 tests, then the above command would result in 3,000 tests being performed. Each subtest would have been executed 100 times. If there was an error encountered along the way, execution of the next subtest would continue without user input.

4.11.5 debug

Enters the interactive debugger.

The format for this command is

debug [*-option*] [*command*]

where

option is one of the following:

f	Forces a reinitialization (download driver, probe, attach, connect)
h	Displays help information for this command
n	Suppresses initialization, even if it is required

command is a specific debugger command. The debugger will execute that command and return back to the main command processor. A script file may be executed from the main level via the debugger. Once in the debugger, online help is available.

The following examples illustrate the use of this command.

debug	Initializes the selected CCU, port, unit, and tape by downloading the current driver and doing an init, probe, attach, and connect. It then prompts for further input with the <debug> prompt.
debug -n	Enters the debugger without downloading the driver and initializing the subsystem. Useful when the intent of entering the debugger is to utilize its powerful script processor to do scan-level subtests.
debug -n <testall	Enters the debugger without initialization, and submits the script file *testall*. Once the *testall* script expires, control is returned to the main command processor.

4.11.6 flags

Displays the *dshell* test flags.

4.11.7 help

Displays a menu of the most frequently used commands. Commands are classified as internal, common, and frequent. Internal commands are not normally executed from the command level. Common commands are the list of all commands. Frequent commands are the most used common commands.

The format for this command is

help [*-option*] [*command*]
help -h *command*

where *option* is one of the following:

a	Displays all commands
h	Displays exhaustive help for all commands
i	Displays all internal commands
m	Prints extended help files for all main and debugger commands to the file *help.manual*
n	Displays all common but normally not displayed commands
?	Displays help usage and commands

and *command* indicates the specific command for which help is needed.

If *option* is omitted, only the most used commands are displayed.

The following examples illustrate the use of this command:

help	Displays the most frequently used commands
help -a	Displays the list of all commands
help -h	Displays the entire help file
help -h help	Displays only the help file for the command *help*

4.11.8 init

Initializes the system to the current state of the settable control constants. It then takes any necessary action to ensure that the current settings of all control constants are true.

The format for this command is

init [*-option*]

where *option* is one of the following:

d	Sets the debug flag to one
D	Resets the debug flag to zero
f	Forces a reinitialization even if logic indicates a satisfactory level of initialization. It is good practice to perform a *init -f* after a complicated normal path error, or when doubts exist as to the state of the system.
h	Displays help information for this command
n	Suppresses initialization

4.11.9 log

Adds comments to the log file. This command can also be used to reset the log file (close the file, then reopen it).

The format for this command is

log [*-option*] [*comment*]

where *option* is one of the following:

c	Closes the logfile and turns logging off
r	Resets the logfile; closes the file, then reopens it

and *comment* is the comment(s) to be added to the log file.

4.11.10 man

Provides online information about diagnostic commands.

The format for this command is

man [*-option*]

where *option* is one of the following:

c *count*	Sets the number of columns on screen (default is 80)
f *file*	Outputs help data to file *file*
h	Displays this extended help file
i *file*	Uses *file* as data file (default file is *Tli.help*)
l *count*	Sets the number of lines on screen (default is 24)
m *token*	Prints all help files that contain the string *token*
t	Prints all known tokens and their indexes
x [*0,1*]	*0* uses the file *help.db* to store the offset/token data base. *1* stores the data base in memory for faster access.
z *count*	Sleeps for *count* seconds after each screen is displayed
?	Displays this usage

4.11.11 Prt_err

Executes the error path for the given err_code. Will display the error details to the screen as well as to the output files *tli.trace* and *tli.err* as appropriate.

The format for this command is

Prt_err *err_code*

where *err_code* is the error code whose details are to be displayed.

4.11.12 quit

Exits the interactive diagnostic.

The format for this command is

q or **quit**

The last thing done before returning to the calling process (*dshell*, etc) is to create a file called *tli4480.save*. Subsequent invocations of the diagnostic will check for the existence of this file, and if so, will take its parameter configuration information from here instead of the interactive parameter query. To force parameter query, such as when a new TLI CCU has been added, simply remove the file before invoking the diagnostic.

4.11.13 -s

Executes a specific subtest, with looping provided.

The format for this command is

-s -[*option*] [*stno*]

where *option* is one of the following:

c
Controls subtest continued execution

h
Displays help on the indicated subtest in the format

stno	subtest number
clno	the class to which the subtest belongs
et	estimated or approximate run length in seconds
description	text description of the subtest or class
ticker	number of seconds expired since invocation

l
Controls the number of times that each subtest is executed before proceeding to the next subtest

L
Controls the number of times that each test in the class is executed

and *stno* is the subtest number(s) to execute. *stno* can be specified in any of the following formats:

N	a specific subtest number
N,N,...N	a list of specific subtest numbers
N-N	a range of subtests, with the specific beginning and ending test numbers included

The following examples illustrate the use of this command.

-s	No subtests would be executed. A list of all subtests and their descriptions would be presented.
-s 1	Executes subtest 1 one time.
-s -l100 1	Executes subtest 1 100 times.
-s -L10 -l10 1,1000	Subtest 1 is a class 0 test, and subtest 1000 is a class 1 test. Subtest 1 would be executed 10 times for 10 times, followed by the same for subtest 1000.

4.11.14 save

Saves the current configuration to the file *tli4480.save*.

4.11.15 set

Updates settable control constants within the interactive diagnostic without having to exit and re-enter the diagnostic. Each time a set command is used to update a constant, a routine is run, unless suppressed by the *-n* option, to perform whatever initialization is required to support the new level.

The format for this command is

set -*flag constant val*

where

flag is one of the following:

- h Displays the help information for the *set* command
- n Suppresses auto-initialization after setting value.
- r Resets *constant* to its default value, or default to reset all constants to their default values.
- s Sets *constant* to *val*, then do any initialization required. The *s* flag must precede each constant that is to be adjusted. More than one constant can be set on one invocation of the command. Each value set is checked against the upper, lower, and default value for safety. In the event that the set value is out of range, the user is notified, and the default is used.
- v Views the value of constant or default to all constants. Data displayed includes the name, lower value, upper value, default value, and current value.
- ? Displays the syntax of the *set* command

constant is the specific constant to be reset. Table 4–13 lists the name of each constant whose value can be reset, its values, and definition.

val is the new value assigned to the constant.

Table 4–13, Settable Constant Values

Name	Lval	Uval	Default	Meaning
banner	0	1	1	auto banner update after <cr>
ccu	0	15	0	selected CCU
continue	0	1	1	auto continue after error: 1[yes] 0[no]
device	0	1	0	device 0[StorageTek] 1[IBM]
dmode	0	1	0	jump to debugger on error: 1[on] 0[off]
driver	0	2	2	selected driver: 0[scan] 1[diag] 2[normal]
log	0	1	1	turn logging 1[on] 0[off]
maxtape	0	15	15	max tape drives to auto-attach
maxunit	0	15	15	max control units to auto-connect
menu	0	1	1	auto menu display after <cr>
port	0	1	0	selected port or channel
print	0	1	0	print error output: 1[on] 0[off]
probe	0	1	1	auto connect all tape drives: 1[yes] 0[no]
rate	0	1	0	xfr rate 0[4.5mb/s] 1[3mb/s]
state	0	5	0	current initialization level
tape	0	3	0	selected tape drive
tock	0	1000	0	post-cursor time update
trace	0	1	0	message tracing: 1[on] 0[off]
trout	0	1	0	trace output: 1[file] 0[screen]
unit	0	15	0	selected control unit (formatter)
verb	0	5	0	error reporting verbosity/sw debug

NOTE

Some of the most frequently used settable control constants can be set directly from the main menu. See *trace*, *trout*, or *results* for more information.

The following example illustrates the use of this command.

```
set -n -s port 1 -s tape 0 -s menu 0
init -f
```

This command would suppress auto initialization, then adjust the current active port or channel to "1", the current tape drive to "0", then turn off the auto menu after every command completion. The following *init -f* is for good measure to force initialization in order to ensure that future commands can be executed without error (the purpose of auto-init). This is a means of delayed initialization.

4.11.16 state

Displays the active state of the selected devices. Displays a table of all of the CCUs, ports, units, and tape drives, and the link counts of each. The link count is incremented upon each selection of that element via *init* or *set -s*. This table is used to verify that the driver can dynamically probe, attach, and connect to any physically attached device.

The format for this command is

state

4.11.17 trace

Enables message tracing to the file *tli.log* or to the screen. A message trace consist of a CMI/MBS message, with certain interpreted fields, the time of receipt/transmission, and the trace count. It is useful in determining the point of fault in a transaction, especially during field integration.

The format for this command is

trace [*option*] [*value*] [*state*]

where

option Indicates whether to send the message trace to a file (*f*) or to the screen (*s*).

value Indicates whether message tracing is enabled (*1*) or disabled (*0*).

state Indicates whether message tracing is enabled (*on*) or disabled (*off*).

The following example illustrates the use of this command.

```
-s 1
trace -f on
init -f
trace off
```

This command sequence first resets the board by performing Subtest 1. It would open up a file called *tli.trace*. The *init -f* would force an initialization of the CCU, saving all message traffic associated with the probe, attach, and connect operations. Then, the file would be closed, making it safe for viewing. If the banner upon return from the *init -f* did not indicate the following:

ccu[*0*][*LOADED*] port[*1*][*PROBED*] unit[*15*][*ATTACHED*] tape[*CONNECTED*]

then the contents of the trace file could be examined to determine exactly what operation failed.

4.11.18 trout

Controls the destination of the output of the *trace* command.

The format for this command is

trout [*-h*] [*option*]

where *-h* displays help information for this command and *option* sets the trace output destination. *option* can be one of the following:

on, 1 Sends output to the file *tli.log*

off, 0 Sends output to the screen

4.12 Interactive Debugger

The *tli4480* diagnostic provides an interactive debugger with 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 **tli4480**[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. Figure 4–4 illustrates the information displayed when you enter **help**:

Figure 4–4, Interactive Debugger Online 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.

    ?                                          - display currently available
                                                 debugger commands
    help    [COMMAND ...]                      - display general or specific help
    cd      [DIRECTORY]                        - change to DIRECTORY
    quit                                       - exit debug mode
    echo    [-n] [arg ...]                     - echo statements to display
    pause   [-n] [seconds]                     - pause for <C/R> or seconds
    exit                                       - exit the diagnostic
    mb      begin [end]                        - modify/[dump] bytes on CCU
    mw      begin [end]                        - modify/[dump] words on CCU
    ml      begin [end]                        - modify/[dump] longs on CCU
    mmb     begin [end]                        - modify/[dump] bytes in MM
    mmw     begin [end]                        - modify/[dump] words in MM
    mml     begin [end]                        - modify/[dump] longs in MM
    fb      begin [end] value [incr [step]]    - fill bytes on CCU
    fw      begin [end] value [incr [step]]    - fill words on CCU
    fl      begin [end] value [incr [step]]    - fill longs on CCU
    ffb     begin [end] value [incr [step]]    - fill bytes in Main Memory
    ffw     begin [end] value [incr [step]]    - fill words in Main Memory
    ffl     begin [end] value [incr [step]]    - fill longs in Main Memory

    weof    count                          - write end of file count times
    fsr     count                          - forward space record count times
    bsr     count                          - backward space record count times
    fsf     count                          - forward space file count times
    bsf     count                          - backward space file count times
    erase                                  - erase tape, default length
    wphys   byte_count [pattern]           - write bytes with optional pat
    rphys   byte_count [pattern to verify]
                                           - read bytes with opt pat to verify
    rewind                                 - rewind tape unit
    iu                                     - attach unit (IO_INIT config unit)
    connect                                - connect unit (IO_CONNECT)
    status                                 - read tape status(IO_RDSTATS_PHYS)
    unload                                 - unload tape (IO_UNLOAD)
    unitclr                                - clear tape error status
    identify                               - diagcmd adaptor identify(rev num)
    display  string                        - display a string to tape drive
    block -[l(loop)s(block size)f(pattern file)o(output)]
                                           - block rd/wr/verify
```

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

This section describes each command in the interactive debugger.

4.13.1 ?

Usage: **?**

Displays a list of the currently available debugger commands.

4.13.2 block

Usage: **block** *option*

Writes, reads, and verifies a block of data to tape, where *option* is one of the following:

f *input-file-name*	is the file containing the input pattern
l *loop*	is the loop count
o *output-file-name*	is the error output file
s *block-size*	is the desired block size

The user creates a pattern file *input-file-name* consisting of any number of hex characters (do not precede the character with '0x'), which is read into memory. It then creates a block of size *block-size* by fitting as many number of patterns in as will fit. The diagnostic then writes the block to the tape, reads it back in, and verifies in a loop of *loop* iterations.

4.13.3 bsf

Usage: **bsf** *count*

Backspaces the number of files equal to *count*. The status is returned in *status.extend*.

4.13.4 bsr

Usage: **bsr** *count*

Backspaces the number of records equal to *count*. The status is returned in *status.extend*.

4.13.5 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 is not set.

4.13.6 connect

Usage: **connect**

Connects to tape drive. This command enables the physical layer operations. The output is stored in *status.code* and *status.modifier.*

4.13.7 display

Usage: **display** *string*

Allows a string of up to 16 characters to be entered from the keyboard and displayed on the selected tape unit. If more than 16 characters are entered, only the first 16 characters will be displayed. If the tape has just been inserted or reset button was pushed on tape unit, you should issue a status command first. If not, a check condition will result from this command.

4.13.8 echo

Usage: **echo** [-n] [arg ...]

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

4.13.9 erase

Usage: **erase**

Erases a default length of tape on the selected tape unit. The current status is returned in *status.extend.*

4.13.10 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.

4.13.11 fb, fl, fw

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

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

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

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

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

The first format (e.g., **fb begin value** [incr [step]]) 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 following examples illustrate use of these commands.

1. *fl 2000000 12345678*

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

2. *fl 200000,0d1000 0*

   The above command zeroes 1000 longword (32 bit) locations starting at 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.

4.13.12 ffb, ffl, ffw

Usage: **ffb begin value** [incr [step]]
ffb begin end value [incr [step]]

ffl begin value [incr [step]]
ffl begin end value [incr [step]]

ffw begin value [incr [step]]
ffw begin end value [incr [step]]

Fills main memory with specified pattern in byte-at-a-time mode (ffb), longword-at-a-time mode (ffl), or word-at-a-time mode (ffw), where:

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

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

4.13.13 fsf

Usage: **fsf** *count*

Spaces the number of files in the forward direction equal to *count*. The status is returned in *status.extend*.

4.13.14 fsr

Usage: **fsr** *count*

Spaces the number of records in the forward direction equal to *count*. The status is returned in *status.extend*.

4.13.15 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

4.13.16 identify

Usage: **identify**

Displays the firmware revision number, the engineering revision number, and the day, month, and year in which the firmware in the PROM was generated.

4.13.17 iu

Usage: **iu**

Performs an attach to the tape unit. The output is stored in *status.code* and *status.extend.*

4.13.18 mb, mw, ml

Usage: **mb begin** [*end*] [*step*]
mw begin [*end*] [*step*]
ml begin [*end*] [*step*]

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
- **step** is the address increment (if this is omitted, the default value is access size)

If *end* is omitted, the debugger 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.

4.13.19 mmb, mmw, mml

Usage: **mmb begin** [*end*] [*step*]
mmw begin [*end*] [*step*]
mml begin [*end*] [*step*]

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
- **step** is the address increment (if this is omitted, the default value is access size)

If *end* is omitted, the debugger 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 -> mb 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.

4.13.20 pause

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

Waits 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.

4.13.21 quit

Usage: **quit**

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

4.13.22 rewind

Usage: **rewind**

Rewinds tape unit with default timeout value of 60 seconds.

4.13.23 rphys

Usage: **rphys** *byte_count* [*pattern*]

Reads the current record on tape, where *byte_count* specifies the number of bytes to read from the selected tape unit. *byte_count* also returns the actual number of bytes read upon completion of the command. *pattern* is an optional 32-bit pattern used to verify data read from the tape. The current tape status is returned in *status.extend.*

4.13.24 status

Usage: **status**

Returns the current tape status in *status.extend.*

4.13.25 unitclr

Usage: **unitclr**

Clears the error status of a tape unit. The current tape status is returned in *status.extend.*

4.13.26 unload

Usage: **unload**

Ejects the cartridge tape and unloads the selected tape unit.

4.13.27 weof

Usage: **weof** *count*

Writes the end-of-file (EOF) mark to tape, where *count* is the number of EOF marks to write to the tape.

4.13.28 wphys

Usage: **wphys** *byte_count* [*pattern*]

Writes a record on tape equal to *byte_count.* *pattern* is an optional 32-bit pattern written repeatedly on the tape during the write process.

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

A

Alaska, reporting problems from, telephone number for xii
Associated documents, how to order xii
Associated documents, listed xi

C

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

D

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

E

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

F

Files, test outputs to 3-1

H

Hardware requirements 4-1
Hawaii, reporting problems from, telephone number for xii
Help-for *tli4480* prompts 4-6

I

Initialization sequence 4-9
I/O, subsystem test, *io* for 1-2
I/O system, test program categories for 1-1
io, test category 1-2

K

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

M

mem, test category 1-2
Memory, subsystem test, *mem* for 1-2
Memory system, test program name for 1-1

N

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

O

Offline tests 1-2
Offline tests, functional, program for 1-3
Online tests 1-2
Online tests, functional, program for 1-3
Overview, diagnostic environment 1-1
Overview, *dshell* 3-1

P

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

R

Reader's Forum xii
Reporting problems xii
Revision sheet 3

S

Screens, test outputs to 3-1
Scripts, predefined 3-1
Self-tests 1-2
Self-tests, test program for 1-3
Service Processor Unit. *See* SPU
SP2, subsystem test, *spu* for 1-2
SP2, *.t* programs and 1-1
SP2, test program name for 1-1
SPU, *dshell* and, introduction 3-1
spu, test category 1-2
Standalone tests 1-2

Subsystems, *cat* for 1-1

T

.t 1-1
TAC, reporting problems to xii
TAC (Technical Assistance Center), problems, reporting to A-1
Tape Library Interface Subsystem Test 4-1
Tape units 1-2
Tape units, test program for 1-3
Technical Assistance Center (TAC), problems, reporting to A-1
Technical assistance, discussed xii
Terminals 1-2
Terminals, test program for 1-3
Test invocation 4-2
Test parameter menu, *tli4480* 4-5
Test programs, categories 1-1
Test programs, categories, table 1-2
Test programs, device types 1-2
Test programs, naming conventions 1-1
Test programs, types 1-2
Test programs, types, table 1-2, 1-3
Tests, options, selecting 3-1
Tests, output, selecting 3-1
tilde-escape sequences A-4
tilde-escape sequences, restrictions on use A-5
tli4480 class descriptions 4-10
tli4480 (Tape Library Interface Subsystem Test) 4-1
tli4480 (test parameter menu) 4-5
tli4480 (test parameter summary) 4-6
Trouble reports xii
trouble reports A-1

U

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

V

vers, program version number found by using A-2

W

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

CONVEX Tape Library Interface Subsystem Test
(*tli4480*) Diagnostics Manual
Document No. 760-003730-000
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)