CONVEX Integrated Disk Channel Subsystem (idc4010) Diagnostics Manual

Order No. DHW-286

First Edition
March 1992

CONVEX Press
Richardson, Texas
United States of America

CONVEX Integrated Disk Channel Subsystem
(`idc4010`*) Diagnostics Manual*
Order No. DHW-286
First Edition

Copyright © 1992 CONVEX Computer Corporation
All rights reserved.

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

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

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

CONVEX and the CONVEX logo ("C") are registered trademarks of CONVEX Computer Corporation.
ConvexOS, C3400, C3800, and the C200 Series are trademarks of CONVEX Computer Corporation.
UNIX is a registered trademark of UNIX System Laboratories, Inc.

Printed in the United States of America

Revision history

CONVEX Integrated Disk Channel Subsystem
(`idc4010`) Diagnostics Manual

Edition	Document No.	Date	Description
First	760-007030-000	March 1992	First release.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Tables

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

Preface

Purpose and intended audience

This manual explains how to run the `idc4010` diagnostic, which verifies the operation of an integrated disk channel (IDC). This document is not a tutorial, but rather a reference for the users of the `idc4010` 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 `idc4010` diagnostic. This document is intended for:

- CONVEX customer support engineers and CONVEX manufacturing personnel
- Customers who install or maintain their own CONVEX supercomputer systems

Organization

This document consists of the following:

- **Chapter 1, "Diagnostics environment"**—Introduces theories and concepts that underlie diagnostics on CONVEX machines and provides an overview of the operating system and `dshell` utility used by the diagnostic tests.
- **Chapter 2, "Integrated disk channel subsystem test** (`idc4010`)"—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.

Notational conventions

Notational conventions are systems of characters, symbols, terms, or abbreviated expressions used to express technical facts or quantities as established by this guide. The following notational conventions are used in this document:

- **Boldface** indicates user-entered information for a computer program that should be entered exactly as it appears.
- *Italic* is used to define new terms, for user-supplied variables, for emphasis, and to indicate titles of publications.
- `Constant-width` is used for code examples, command names and options, error messages, screen output, and system calls.
- ▭ indicates a specific keyboard key to press. A hyphen between two keycap symbols indicates to press the two keys simultaneously. A space between two symbols indicates a sequence of keys to press.
- 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 *byte* is 8 bits.
- A *halfword* is 16 bits.
- A *word* is 32 bits.
- A *longword* is 64 bits.
- 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.
- 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.
- *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 Processor Diagnostics Manual (C3400 Series)*, Order No. DHW-302
- *CONVEX SPU UNIX Utilities Manual*, Order No. DHW-021
- *CONVEX SPU System Manager's Guide*, Order No. DSW-022
- *CONVEX Diagnostic Utilities Manual (C1, C120)*, Order No. DHW-072
- *CONVEX Diagnostic Utilities Manual (C200 Series)*, Order No. DHW-082

Ordering documents

To order the most current version of this or any other CONVEX document, send requests to:

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

Include the order number with the request. The order number is on the title page of the manual and begins with the letters "DSW" or "DHW."

Technical assistance

Hardware, software and documentation 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 Canada, call 1(800)345-2384.
- From all other locations, contact the nearest CONVEX office.

Using the `contact` utility

The TAC recommends using the `contact` utility to report a hardware, software, or documentation problem. The `contact` utility is an interactive program that helps the TAC track reports and route them to the CONVEX personnel most qualified to fix a problem. After you invoke `contact`, it prompts you for information about the problem. When you finish your report, `contact` mails it to the TAC electronically.

The TAC notifies you within 48 hours that your report has been received. To use `contact` requires:

- UNIX-to-UNIX Communications Protocol (UUCP) connection to the TAC.
- Full path name of the program or utility in question.
- Version number of the program or utility in question.

Refer to the contact(1) man page for complete details.

Acknowledgments

I would like to thank the following people for their contributions to this manual:

- Technical contributor: Alex Chan
- Document review team: Martin Mennig, Kris Meier
- Editorial Services: Sheri Roloff, Peggy Gilloon

This document would not have been possible without their help.

Cari Tuttle
CONVEX I/O Documentation

THIS PAGE INTENTIONALLY LEFT BLANK

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 OS. 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 diagnostics in this manual are intended to be executed "offline," that is, while ConvexOS is not being executed by any of the central processing units (CPUs) in the system.

The service processor, together with SPU OS, various diagnostic utilities, and the test programs themselves, comprise the CONVEX diagnostic environment. This chapter provides an overview of the operating system and `dshell` utility used by the diagnostic tests. For more information about the diagnostic environment, refer to the *CONVEX Processor Diagnostics Manual (C1, C120)*, *CONVEX Processor Diagnostics Manual (C200 Series)*, or *CONVEX Processor Diagnostics Manual (C3400 Series)*, depending on the architecture of the machine under test.

1.2 `dshell` utility 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. This section provides a brief overview of the `dshell` utility, including a brief explanation of the utility and a list of the utility's commands. For a complete description of the `dshell` utility, refer to the Dshell chapter of the *CONVEX Diagnostic Utilities Manual (C200 Series)*.

1.2.1 Diagnostic shell (`dshell`) overview

The `dshell` has two functions:

- Selecting diagnostics for execution
- Selecting test options as listed:
  - 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
  - Select long or short error messages, or turn messages off
  - Execute either user-created or predefined command scripts

1.2.2 `dshell` commands

Table 1–1 summarizes the various `dshell` commands and their functions.

Table 1–1, `dshell` commands

Command	Function
`!` [*command*]	Accesses or forks a SPU OS shell to execute the command that follows `!`.
`exit` or `quit`	Immediately terminates the `dshell` process and any test processes that may have been forked.
CTRL-C	Returns user to the `dshell` command level if no subtest is running. If subtest is running, provides options to continue or abort subtest.
CTRL-B	Immediately terminates the `dshell` and any associated active processes. Core is dumped.
`help`	Displays a standard help menu. The menu describes the correct command syntax for each `dshell` command and gives a terse description of what each command does.
`status`	Generates a report on the current state of the `dshell` command options. This report gives the name of each option, its current value, and an explanation of its current effect.
`log` [*options*]	Provides a mechanism for specifying the number of failures allowed to occur before a test or subtest terminates execution.
`loop` [*options*]	Causes `dshell` to repeat the execution of a test or subtest.
`msgs` [*options*]	Enables or disables different levels of test, class, and subtest result messages.
`pause` [*options*] [*nn*]	Returns program control to the `dshell` at the beginning, end, or failure of all or specific subtests.
`test` [*testname*] [*options*]	Executes specific tests and displays test, class, and subtest menus.

1.2.3 Syntax help for dshell commands

The syntax for each **dshell** command can be obtained by typing the command without options and pressing RETURN. For example, by typing **loop** and pressing RETURN, the syntax help in Figure 1–1 will be displayed on the screen.

Figure 1–1, 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 2
Integrated disk channel subsystem test (idc4010)

2.1 Overview

idc4010 is a diagnostic test suite for the CONVEX integrated disk channel (IDC) subsystem. The IDC subsystem consists of:

- IDC IPI-3 interface channel control unit (CCU)
- Internal cabling from the CCU to the CONVEX external IPI-3 connectors
- Downloaded CCU driver software

Specifically, idc4010 accomplishes the following:

- Verifies that the IDC 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 IDC subsystem is functional, reliable, and capable of supporting ConvexOS
- Provides an interactive debugger that can execute commands from a script file.

CCU communications use the message-based system (MBS) used by ConvexOS. The intent is to test the communications paths used in a normal operating environment.

The idc4010.t diagnostic is unique as a CONVEX diagnostic in two ways:

- It is a dedicated CCU diagnostic; it does not access a chassis controller such as a VMEbus SCSI controller.
- 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 or 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.

2.2 Required equipment

Table 2–1 lists the required hardware for the IDC subsystem.

Table 2–1, Hardware requirements

C200 Series
Memory system[1]
CPX
SP2 or SP4
PIA or PI2
Integrated disk channel subsystem

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

Class 2 subtests require an external loopback cable or connector; these are standard cables and connectors that you must modify to run these tests.

The special external loopback cable is a regular IPI jumper cable (CONVEX Part Number 604-500007-001), to which you must make the following modificatons:

BUSA bit 0-7 of end 1	connected to	BUSB bit 0-7 of end 2
BUSA parity of end 1	connected to	BUSB parity of end 2
BUSB bit 0-7 of end 1	connected to	BUSA bit 0-7 of end 2
BUSB parity of end 1	connected to	BUSA parity of end 2
DC Ground of end 1	connected to	DC Ground of end 2

Other signal pins are disconnected.

The special external loopback connector is a regular IPI male connector, to which you must make the following modificatons:

SELO connected to ATNI
MASO connected to SLVI
SYNO connected to SYNI

2.3 Test invocation

The `idc4010` test executes under the diagnostic shell (`dshell`) and supports all the features of the `dshell`. The `dshell` permits tests to be initiated in any order.

To invoke the `idc4010` test, use the procedure shown in Figure 2–1. All responses in **boldface** are entered by the user.

> **NOTE**
>
> Use the following test invocation sequence for the initial invocation of `idc4010` 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 2–1, Initial Test Invocation Sequence

```
(spu)> cd /mnt/test
(spu)> initall
(spu)> dshell
:test idc4010 [-option]  [...] ]
```

> **NOTE**
>
> After entering `dshell`, specific `dshell` parameters may be changed. Refer to Chapter 2 in the *CONVEX Diagnostic Utilities Manual (C200 Series)* for more information.

Table 2–2 defines the options ([-*option*]) available for the `test` command.

Table 2–2, `test` command options

Option	Description
`-d`	Enter the debugger mode of the program without executing any subtests.
`-f` *file*	Use an alternate parameter save file name. If this option is omitted, the parameter file used is /tmp/idc4010.tmp.
`-i`	Go to interactive mode only; no tests are executed.
`-r` *script-filename*	Execute using commands from script file.

Entering only **test idc4010** executes all idc4010 subtests sequentially. To execute one or more classes or individual subtests, use the -c or -s options during test invocation, respectively.

2.4 Initialization sequence for idc4010

Once the test is invoked, the diagnostic determines if the test was invoked for quick startup with idc4010x.t. If so, the diagnostic reads the test parameters from the specified parameter file (default parameter file is /tmp/idc4010.save). If not, the following actions are taken:

1. A list of prompts is displayed sequentially, allowing parameters to be set by the user or defaults accepted.
2. A TEST PARAMETER SUMMARY is displayed, listing all prompts and their responses.
3. Input parameters are written to the parameter file (default or user-specified).
4. The diagnostic downloads the CCU driver code (if necessary).
5. The diagnostic passes the current test parameters to the CCU driver.
6. If the -d option was not specified, the test code is started; otherwise, the test enters the interactive debugger.

The following sections describe these steps in more detail.

> **NOTE**
>
> The boot_db file (default /mnt/boot_db) determines what CCU and memory are installed. If this file is non-existent, it can be created from the (spu)> prompt with the command `scn_util -b > /mnt/boot_db`.

2.5 Test parameter menu

If not invoked with idc4010x.t, the diagnostic displays test parameter prompts sequentially, allowing selection of default test parameters or specification of different values. Figure 2–2 shows all prompts, their possible answers in brackets [ ], and their default answers in parentheses ( ). The prompts and responses in the figure appear sequentially on the screen, one line at a time. The figure illustrates *all* questions that can be displayed during test parameter input; however, some questions may be omitted, depending on answers to previous questions. In all cases, questions are numbered sequentially.

Figure 2–2, 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 IDC to test [0-15,?]                  (0) -> RETURN
 2: Use Defaults for Remaining Parameters [y,n,?]              (y) -> n
 3: Print error output: 1[on] 0[off] [0-1,?]                   (1) -> RETURN
 4: Trace messages [0-1,?]                                     (0) -> RETURN
 5: Trace output control (screen/file) [0-1,?]                 (0) -> RETURN
 6: Auto menu:  0[off], 1[on] [0-1,?]                          (1) -> RETURN
 7: Auto banner:  0[off], 1[on] [0-1,?]                        (1) -> RETURN
 8: Error Reporting Verbosity [0-5,?]                          (0) -> RETURN
 9: <cr> req'd for continue. 1[yes] 0[no] [0-1,?]              (1) -> RETURN
10: auto-logging 1[on] 0[off] [0-1,?]                          (1) -> RETURN
11: Ticker tock: on[1], off[0] [0-1000,?]                      (0) -> RETURN
12: Enable Debug Monitor [y,n,?]                               (n) -> RETURN
13: Select Printer On/Off Mode [0x0-0x3,?]                    0x0) -> 0x3
14: Enter OK, or :NN to return to question NN [OK]            (OK) -> RETURN
```

For help or information during test parameter entry, enter one of the characters shown in Table 2–3, then press RETURN.

Table 2–3, 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.

2.5.1 Prompt explanations

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

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

Enter the slot number of the IDC board that you wish to test. For all machines except the C3800, the valid range is from 0–15, and for the C3800 the valid range is from 32–39.

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

Enter **y** to use the default values for all remaining prompts. If the response to this prompt is **y**, the remaining prompts are bypassed, and default values are used for their parameters.

```
3: Print error output: 1[on] 0[off]  [0-1,?]           (0) -> RETURN
```

Enter **1** if you want to print error output. Otherwise, no error output will be printed.

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

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

```
5: 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.

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

Enter **1** to automatically display the menu after a RETURN is entered.

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

Enter **1** to automatically update the banner after a RETURN is entered.

```
8: 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.

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

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

```
10: 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.

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

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

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

Enter **y** to automatically enter the debugger in the event of an error.

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

13: 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 `idc4010` 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 performing a logical OR operation on the two options together (0x03).

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

This option lets you return to a specified question number and change the answer. If **OK** or RETURN is entered, the test parameters are saved in the test parameter file, and they are no longer changeable.

2.5.2 Test parameter summary

When all prompts are answered, the screen displays a TEST PARAMETER SUMMARY menu that shows the prompts that were answered and their responses. Figure 2–3 illustrates an example of a TEST PARAMETER SUMMARY screen. Actual values and responses vary according to the input.

Figure 2–3, Sample test parameter summary

```
                         TEST PARAMETER SUMMARY

CCU number (slot) of IDC to test                           : 0
Use Defaults for Remaining Parameters                      : n
Print error output: 1[on] 0[off]                           : 1
Trace messages                                             : 0
Trace output control (screen/file)                         : 0
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
Ticker tock: on[1], off[0]                                 : 0
Enable Debug Monitor                                       : n
Select Printer On/Off Mode                                 : 0x0000
Enter OK, or :NN to return to question NN                  : OK
```

2.6 Class descriptions

Table 2–4 lists the classes of subtests contained in the `idc4010` diagnostic.

Table 2–4, `idc4010` test classes

CLASS	DESCRIPTION
0	Scan-ring/EPROM-based board tests
1	Downloaded RAM-based subsystem board tests
2	Manufacturing support board 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.

2.7 Class 0 subtests

Class 0 subtests test data and program memory, channel buffer memory, control gate arrays, registers, and PBUS interface logic. They verify that the IDC 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 2–5 lists all class 0 subtests, their descriptions, and the approximate time in seconds required to execute each subtest.

Table 2–5, Class 0 subtests

Subtest	Description	Time (seconds)
1	Board reset test	1
2	Slot verification test	6
100	Data RAM bit functionality test	10
110	Data RAM column functionality test	4
120	Data RAM uniqueness test	1
130	Data RAM parity test	1
200	Instruction RAM bit functionality test	10
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	6
320	PMAP RAM uniqueness test	1
330	PMAP RAM parity test	1
400	PBUS header generation test	1
410	PBUS access test	3
500	Data buffer port 0 bit functionality test	8
501	Data buffer port 0 column functionality test	3
502	Data buffer port 0 uniqueness test	3
503	Data buffer port 0 parity test	3
510	Data buffer port 1 bit functionality test	10
511	Data buffer port 1 column functionality test	3
512	Data buffer port 1 uniqueness test	3
513	Data buffer port 1 parity test	3
520	Data buffer port 2 bit functionality test	8
521	Data buffer port 2 column functionality test	3
522	Data buffer port 2 uniqueness test	3
523	Data buffer port 2 parity test	3
530	Data buffer port 3 bit functionality test	10
531	Data buffer port 3 column functionality test	3
532	Data buffer port 3 uniqueness test	3
533	Data buffer port 3 parity test	3
604	DPED 0 column functionality test	3
605	DPED 0 parity test	4
614	DPED 1 column functionality test	2
615	DPED 1 parity test	3
616	DPED 2 column functionality test	2
617	DPED 2 parity test	3
618	DPED 3 column functionality test	3
619	DPED 3 parity test	4
620	Data buffer all ports uniqueness test	8

Table 2-5 is continued on the next page.

Table 2–5, Class 0 subtests (continued)

Subtest	Description	Time (seconds)
630	DICE 0 bit functionality test	32
631	DICE 0 column functionality test	13
632	DICE 0 uniqueness test	8
633	DICE 0 parity test	5
640	DICE 1 bit functionality test	32
641	DICE 1 column functionality test	13
642	DICE 1 uniqueness test	8
643	DICE 1 parity test	5
650	DICE 2 bit functionality test	32
651	DICE 2 column functionality test	13
652	DICE 2 uniqueness test	8
653	DICE 2 parity test	5
660	DICE 3 bit functionality test	32
661	DICE 3 column functionality test	13
662	DICE 3 uniqueness test	8
663	DICE 3 parity test	5
700	PIGA bit functionality test	8
701	PIGA column functionality test	4
702	PIGA uniqueness test	7
800	Data RAM protection test	350
900	EEPROM write protection test	6

Table 2–6 lists the default data pattern used by subtests 100, 200, 300, 500, 510, 520, 530, 630, 640, 650, 660, and 700, unless an alternate pattern is specified at `idc4010` initialization.

Table 2–6, Class 0 data pattern

Hexadecimal test pattern	
0x00000000	0xAAAAAAAA
0xffffffff	0x11111111
0x55555555	0xeeeeeeee

2.7.1 Subtest 1, board reset test

Subtest 1 resets the IDC board and checks the results of the self-test. The self-test on the IDC board includes an EPROM checksum test.

This subtest uses the scan-ring to set, then release, the reset signal of the IDC 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.

2.7.2 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 common message interface (CMI) data structures. If the slot ID yields the wrong location, it will not be possible to download and communicate with the CCU via CMI.

2.7.3 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 2-6. Only longword access is checked.

2.7.4 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 (moving from least significant bit to most significant bit).

2.7.5 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.

2.7.6 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.

2.7.7 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 2-6. Only longword accesses are checked.

2.7.8 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 (moving from least significant bit to most significant bit).

2.7.9 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 checked for uniqueness.

2.7.10 Subtest 230, instruction RAM parity test

Subtest 230 tests instruction RAM parity checking. It performs a walking 1s and 0s 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.

2.7.11 Subtest 240, instruction RAM execution test

Subtest 240 tests that the code in the instruction RAM can be correctly executed. Code that computes the EEPROM checksum is copied out of EEPROM into IRAM. The IRAM code is called, computing the checksum and returning to EEPROM. If an instruction fault occurs, an error is reported back to the SPU.

2.7.12 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 2–6. Only longword access is checked.

2.7.13 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 (moving from least significant bit to most significant bit).

2.7.14 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.

2.7.15 Subtest 330, PMAP RAM parity test

Subtest 330 tests that parity checking is working for the PMAP registers. The first PMAP register is written with a zero and bad parity. The 88100 processor then reads the PMAP register and verifies that the access 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 IDC. The IDC 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.

The PMAP registers are word-accessible only.

2.7.16 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 the lower, 32 bits of the PB2LBREG diagnostic register, 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, and both the expected and the actual values are reported back to the SPU.

2.7.17 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 IDC agree on the location of main memory and of the proper byte ordering in that memory.

2.7.18 Subtests 500, 511, 521, 531 (data buffer bit functionality tests)

Subtests 500, 511, 521, and 531 test the bit functionality of the data buffers for each port. They perform a true/complement pattern test for all locations in the data buffer using the patterns in Table 2–6. Only longword access is checked. Table 2–7 lists the subtest number and the port it tests.

Table 2–7, Data buffer bit functionality subtests

Subtest	Port tested
500	Port 0
511	Port 1
521	Port 2
531	Port 3

2.7.19 Subtests 501, 511, 521, and 531 (data buffer column functionality tests)

Subtests 501, 511, 521, and 531 test the column functionality of the data buffer for each port. They perform a walking 1s and 0s test on the first word in each bank of the data buffer (moving from least significant bit to most significant bit). Table 2–8 lists the subtest number and the port it tests.

Table 2–8, Data buffer column functionality subtests

Subtest	Port tested
501	Port 0
511	Port 1
521	Port 2
531	Port 3

2.7.20 Subtests 502, 512, 522, and 532 (data buffer uniqueness tests)

Subtests 502, 512, 522, and 532 test the location uniqueness of the data buffer for each port. 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. Table 2–9 lists the subtest number and the port it tests.

Table 2–9, Data buffer uniqueness subtests

Subtest	Port tested
502	Port 0
512	Port 1
522	Port 2
532	Port 3

2.7.21 Subtests 503, 513, 523, and 533 (data buffer parity tests)

Subtests 503, 513, 523, and 533 test that the parity checking is working for the data buffer for each port. 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 the access generated a data fault and that the FSR logs an LBUS fault. The process is repeated with a pattern of 0xfefefefe (odd number of bits in each byte). Table 2–10 lists the subtest number and the port it tests.

Table 2–10, Data buffer parity subtests

Subtest	Port tested
503	Port 0
513	Port 1
523	Port 2
533	Port 3

NOTE

For subtests 513 and 523, a window access on a PMAP register with bad parity is not detected by the ITC. The ITC 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.

The PMAP registers are word-accessible only.

2.7.22 Subtests 604, 614, 616, and 618 (DPED column functionality tests)

Subtests 604, 614, 616, and 618 test the column functionality of the data path and error detection (DPED) basic control register at address ff0090 for each port. They perform a walking 1s and 0s test on the lower 30 bits of the four-byte word (moving from least significant bit to most significant bit). Table 2–11 lists the subtest number and the port it tests.

Table 2–11, DPED column functionality subtests

Subtest	Port tested
604	Port 0
614	Port 1
616	Port 2
618	Port 3

2.7.23 Subtests 605, 615, 617, and 619 (DPED parity tests)

Subtests 605, 615, 617, and 619 test that the parity checking is working for the DPED for each port. 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 LBUS fault. the process is repeated with a pattern of 0xfefefefe (odd number of bits in each byte). Table 2–12 lists the subtest number and the port it tests.

Table 2–12, DPED parity subtests

Subtest	Port tested
605	Port 0
615	Port 1
617	Port 2
619	Port 3

2.7.24 Subtest 620, data buffer all ports uniqueness test

Subtest 620 tests the location uniqueness of the data buffer. It writes an incrementing value to each location in the data buffers (16 kbytes), then verifies that all locations contain the expected value. Only longword accesses are checked for uniqueness.

2.7.25 Subtests 630, 640, 650, and 660 (DICE bit functionality tests)

Subtests 630, 640, 650, and 660 test the bit functionality of the device interface command execution (DICE) registers at address ff0000 for each port. They perform a true/complement pattern test for CREG, OPREG, TCREG, HCREG, FCREG, XREG, and HREG in the DICE, using the patterns in Table 2–6. Only longword access is checked, and the most significant bit of OPREG is not checked. Table 2–13 lists the subtest number and the port it tests.

Table 2–13, DICE bit functionality subtests

Subtest	Port tested
630	Port 0
640	Port 1
650	Port 2
660	Port 3

2.7.26 Subtests 631, 641, 651, and 661 (DICE column functionality tests)

Subtests 631, 641, 651, and 661 test the column functionality of the DICE registers at address ff0000 for each port. They perform a walking 1s and 0s test on the first word of OPREG, TCREG, HCREG, FCREG, XREG, and HREG (moving from least significant bit to most significant bit, or to second most significant bit of OPREG). Table 2–14 lists the subtest number and the port it tests.

Table 2–14, DICE column functionality subtests

Subtest	Port tested
631	Port 0
641	Port 1
651	Port 2
661	Port 3

2.7.27 Subtests 632, 642, 652, and 662 (DICE uniqueness tests)

Subtests 632, 642, 652, and 662 test the location uniqueness of the DICE registers at address ff0000 for each port. They write an incrementing value to each location in OPREG, TCREG, HCREG, FCREG, XREG, and HREG, then verifies that all locations contain the expected value. Only longword accesses are checked for uniqueness and the most significant bit of OPREG is not checked. Table 2–15 lists the subtest number and the port it tests.

Table 2–15, DICE uniqueness subtests

Subtest	Port tested
632	Port 0
642	Port 1
652	Port 2
662	Port 3

2.7.28 Subtests 633, 643, 653, and 663 (DICE parity tests)

Subtests 633, 643, 653, and 663 test that the parity checking is working for the DICE registers for each port. The first DICE register is written with a zero and bad parity. The 88100 processor then reads the DICE register and verifies that the access 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). Table 2–16 lists the subtest number and the port it tests.

Table 2–16, DICE parity subtests

Subtest	Port tested
633	Port 0
643	Port 1
653	Port 2
663	Port 3

2.7.29 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 2–6. Only longword access is checked.

2.7.30 Subtest 701, PIGA column functionality test

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

2.7.31 Subtest 702, PIGA uniqueness test

Subtest 702 tests PIGA location uniqueness. It writes an incrementing value to each location in the PIGA, then verifies that all locations contain the expected value. Only longword access is checked.

2.7.32 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.

2.7.33 Subtest 900, EEPROM write protection test

Subtest 900 verifies that the local processor on the CCU cannot write its EEPROM unless it is write enabled.

2.8 Class 1 subtests

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

CAUTION

Run these tests with no devices connected to the IPI interface. Otherwise, unexpected results may occur.

Table 2–17 lists all class 1 subtests, their descriptions, and the approximate time in seconds required to execute each subtest.

Table 2–17, Class 1 subtests

Subtest	Description	Time (seconds)
1000	Port 0 channel-to-memory data path test	04[1]
1010	Port 1 channel-to-memory data path test	04
1020	Port 2 channel-to-memory data path test	04
1030	Port 3 channel-to-memory data path test	04
1100	Port 0 memory-to-channel data path test	04
1110	Port 1 memory-to-channel data path test	04
1120	Port 2 memory-to-channel data path test	04
1130	Port 3 memory-to-channel data path test	04
1400	Port 0 alignment test	04
1410	Port 1 alignment test	04
1420	Port 2 alignment test	04
1430	Port 3 alignment test	04
1500	Port 0 arbiter test	04
1510	Port 1 arbiter test	04
1520	Port 2 arbiter test	04
1530	Port 3 arbiter test	04

[1] This subtest may take 25–30 seconds because of the time needed to download the driver.

2.8.1 Subtests 1000, 1010, 1020, and 1030 (channel-to-memory data path tests)

Subtests 1000, 1010, 1020, and 1030 simulate a READ disk drive operation. A 4,096-byte record is sent from a channel to memory, and data in both the data buffer and memory is checked and verified. Any discrepancy is reported for each channel.

Subtest 1000 is performed twice, to check the operation of both register sets for each port in the PIGA.

Table 2–18 lists the subtest number and the port it tests.

Table 2–18, Channel-to-memory data path subtests

Subtest	Port tested
1000	Port 0
1010	Port 1
1020	Port 2
1030	Port 3

2.8.2 Subtests 1100, 1110, 1120, and 1130 (memory-to-channel data path tests)

Subtests 1100, 1110, 1120, and 1130 simulate a WRITE disk drive operation. A 4,096-byte record is sent from memory to a channel, and data in both the data buffer and memory is checked and verified. Both register sets are tested for operation for each port in the PIGA. Any discrepancy is reported for each channel. Table 2–19 lists the subtest number and the port it tests.

Table 2–19, Memory-to-channel data path subtests

Subtest	Port tested
1100	Port 0
1110	Port 1
1120	Port 2
1130	Port 3

2.8.3 Subtests 1400, 1410, 1420, and 1430 (alignment tests)

Subtests 1400, 1410, 1420, and 1430 write data to main memory that is not mapped and expect a PBUS error. Both register sets are tested for operation for each port in the PIGA. Table 2–20 lists the subtest number and the port it tests.

Table 2–20, Alignment subtests

Subtest	Port tested
1400	Port 0
1410	Port 1
1420	Port 2
1430	Port 3

2.8.4 Subtests 1500, 1510, 1520, and 1530 (arbiter tests)

Subtests 1500, 1510, 1520, and 1530 simulate a READ disk drive operation. A 4,096-byte record is sent from a channel to memory with buffer count register initially set to -1. Starting memory locations are checked to be sure that it is not written by the arbiter logic. Buffer counter register is then incremented to 1. Data buffer and memory is checked and verified. Any discrepancy is reported. Table 2–21 lists the subtest number and the port it tests.

Table 2–21, Arbiter subtests

Subtest	Port tested
1500	Port 0
1510	Port 1
1520	Port 2
1530	Port 3

NOTE

Subtest 1500 requires the latest buffer arbiter logic and must be run on Rev D of the IDC firmware.

2.9 Class 2 subtests, manufacturing support board tests

The class 2 subtests verify that the ITC main data paths operate properly, using an external loopback cable or connector. Subtest 2000 requires the connection of a special external loopback cable; subtest 2010 requires the connection of a special external loopback connector.

The special external loopback cable is a regular IPI jumper cable (CONVEX Part Number 604-500007-001) to which you must make the following modifications:

BUSA bit 0-7 of end 1	connected to	BUSB bit 0-7 of end 2
BUSA parity of end 1	connected to	BUSB parity of end 2
BUSB bit 0-7 of end 1	connected to	BUSA bit 0-7 of end 2
BUSB parity of end 1	connected to	BUSA parity of end 2
DC Ground of end 1	connected to	DC Ground of end 2

Other signal pins are disconnected.

The special external loopback connector is a regular IPI male connector to which you must make the following modifications:

SELO	connected to	ATNI
MASO	connected to	SLVI
SYNO	connected to	SYNI

Table 2–22 lists all class 2 subtests, their descriptions, and the approximate time in seconds required to execute each subtest.

Table 2–22, Class 2 subtests

Subtest	Description	Time (seconds)
2000	Transceiver bus A/bus B bit functionality test	4
2010	Transceiver control signals bit functionality test	4

2.9.1 Subtest 2000, transceiver bus A/bus B bit functionality test

Subtest 2000 writes a pattern of 1s and 0s to the bus A signal on one channel and verifies the pattern by reading the bus B signal on another channel.

2.9.2 Subtest 2010, transceiver control signals bit functionality test

Subtest 2010 writes a pattern of 1s and 0s to the control signal and verifies that the channel is in the correct state.

2.10 Interactive commands

`idc4010` provides commands that can be invoked interactively from the <ROM> prompt. The diagnostic must first be invoked using the *-i* option. Table 2–23 lists each command followed by a synopsis of its meaning.

Table 2–23, Interactive commands

Command	Meaning
`!`	Executes ConvexOS 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 */tmp/idc4010.save*
`set`	Changes settable control constant
`trace`	Enables message tracing
`trout`	Controls trace output

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

2.10.1 !

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

The format for this command is

! *command*

where *command* is the ConvexOS 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 is returned to the diagnostic.

2.10.2 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                                                  date
    ccu[no.][status]                                          clk[status]
state: CURSTATE trace: STATE to: WHERE verb: LEVEL print: STATE
```

where

name	is the name of the diagnostic.
version	is the build/release version.
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.
clk *rate*	is the clock rate of IDC, where *rate* can be normal or margin.
state *curstate*	is the initialization level of CCU/diagnostic, where *curstate* is one of the following:

- 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 disk 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 is 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* is one of the following:	
	FILE	all trace output will be stored in the file /tmp/idc4010.log. /tmp/idc4010.log 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 *level*	Is the current verbosity level as set in the initial parameter selection or the `set -s verb` command. *level* 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	When enabled via the initial parameter selection or the `set -s printer 1` command, all error messages will be printed.	

2.10.3 `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
- x — Hexadecimal

2.10.4 `-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:

- 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 class 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`	Executes each subtest in class 0 ten consecutive times, then repeats the process ten times, then repeats 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.

2.10.5 `debug`

Enters the interactive debugger.

> **NOTE**
>
> The debugger can only be entered from the normal path mode. This command does not work from the <ROM> prompt; in that case, you must issue an `init` command before `debug` will work.

The format for this command is

init 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 by downloading the current driver. It then prompts for further input with the <debug> prompt.
`debug -n`	Enters the debugger without downloading the driver. 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.

2.10.6 `flags`

Displays the `dshell` test flags.

2.10.7 `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
- i Displays all internal commands
- n Displays all common but normally not displayed 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

2.10.8 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 an *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

2.10.9 log

Closes 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*]

where *option* is one of the following:

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

2.10.10 man

Provides online information about diagnostic commands.

The format for this command is

man *token*

where *token* is the command whose details are to be displayed.

2.10.11 Prt_err

Executes the error path for the given error code. Will display the error details to the screen.

The format for this command is

Prt_err *err_code*

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

2.10.12 `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` and so on) is to create a file called /tmp/idc4010.save.

2.10.13 `-s`

Executes a specific subtest, with looping provided.

The format for this command is

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

where *option* is one of the following:

- 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 once.
`-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.

2.10.14 `save`

Saves the current configuration to the file /tmp/idc4010.save.

2.10.15 `set`

Updates settable control constants within the interactive diagnostic without having to exit and reenter 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 2–24 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 2–24, Settable constant values

Name	Lval	Uval	Default	Meaning
banner	0	1	1	Auto banner update after (RETURN)
ccu	0	15	0	Selected CCU
continue	0	1	1	Auto continue after error: 1[yes] or 0[no]
dmode	0	1	0	Jump to debugger on error: 1[on] or 0[off]
log	0	1	1	Turn logging 1[on] or 0[off]
menu	0	1	1	Auto menu display after (RETURN)
print	0	1	0	Print error output: 1[on] or 0[off]
tock	0	1000	0	Post-cursor time update
trace	0	1	0	Message tracing: 1[on] or 0[off]
trout	0	1	0	Trace output: 1[file] or 0[screen]
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 menu 0
init -f
```

This command suppresses auto initialization. The following `init -f` is recommended to force initialization in order to ensure that future commands can be executed without error (the purpose of auto-init). This command is a means of delayed initialization.

2.10.16 `trace`

Enables message tracing to the file /tmp/idc4010.log or to the screen. A message trace consists of a CMI/MBS message with certain interpreted fields, the time of receipt or transmission, and the trace count. Tracing 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 opens a file called /tmp/idc4010.log. The `init -f` forces an initialization of the CCU, saving all message traffic associated with the probe, attach, and connect operations. Then the file is closed, making it safe for viewing.

2.10.17 `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 /tmp/idc4010.log

off, 0 Sends output to the screen

2.11 Interactive debugger

The `idc4010` diagnostic provides an interactive debugger that allows the operator to perform some low-level interactions that are helpful when diagnosing a subtest failure. Invoke the debugger with one of the following methods:

- Use the **-d** option when invoking the diagnostic (**idc4010**[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, perform a logical OR operation using **0x10** in 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 2–4 illustrates the information displayed when you enter **help**.

Figure 2–4, Interactive debugger online help

```
Input base specification:
    OdNN - decimal,  OxNN 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
    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
```

In addition to the help screen in Figure 2–4, 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**.

2.12 Interactive debugger command descriptions

This section describes each command in the interactive debugger.

2.12.1 `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

2.12.2 ?

Usage: **?**

Displays a list of the currently available debugger commands.

2.12.3 `cd`

Usage: **cd** [*directory*]

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

2.12.4 `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.

2.12.5 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 CCU memory with specified pattern in byte-at-a-time mode (`fb`), halfword-at-a-time mode (`fw`), or word-at-a-time mode (`fl`), where:

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

The first format (for example, `fb` *begin value* [*incr* [*step*]]) stores *value* at address *begin*.

The second format (for example, `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 for a byte, two for a halfword, or four for a word.

The following examples illustrate the use of these commands.

1. *fl 2000000 12345678*

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

2. *fl 200000,0d1000 0*

   This command zeroes 1000 longword (32-bit) locations starting at main memory address 0x200000.

3. *fb 20000,0d40 10 4 2*

   This command fills 40 even bytes starting at CCU address 0x20000 with a value that begins at ten and increments by four each time.

2.12.6 `ffb, ffl, ffw`

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

Fills main memory with specified pattern in byte-at-a-time mode (ffb), halfword-at-a-time mode (ffw), or word-at-a-time mode (ffl)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 for a byte, two for a halfword, or four for a word.

2.12.7 `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`), halfword-at-a-time mode (`mw`), or word-at-a-time mode (`ml`), where:

- *begin* is the starting address.
- *end* is the ending address.
- *step* is the address increment (if omitted, 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 (reread).
[<*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-> ml c00000
 <CCU:00c00000> = 1c  00=ff,1q
```

00=ff,1q is an example of executing multiple commands on the same line. This sequence displays the word value at IDC address c00000 and allows the operator to modify this value. The operator's response modifies the word to 0, rereads and displays the new value at c00000, modifies the value again to 0xff, skips to address c00000, modifies its value to 0x1, then exits the interactive mode.

2.12.8 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`), halfword-at-a-time mode (`mmw`), or 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 omitted, 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 (reread).
[<*value*>]^[*N*]	Write optional <*value*> to current address, move to address *N* (address 0 if *N* is omitted).
[<*value*>]+[*N*]	Write optional <*value*> to current address, advance to the next address (*N* addresses, if *N* is specified).
[<*value*>]-[*N*]	Write optional <*value*> to current address, back up to the previous address (*N* addresses, if *N* is specified).
[<*value*>]**q**	Write optional <*value*> to current address, exit interactive mode.

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

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

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, rereads 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.

2.12.9 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.

2.12.10 quit

Usage: **quit**

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

Index

Symbol

A

B

C

D

E

F

H

I

L

M

N

O

P

CONVEX COMPUTER CORPORATION

CONVEX INTEGRATED DISK CHANNEL SUBSYSTEM DIAGNOSTICS

760-007030-000

PRINTED IN U.S.A.