CONVEX Fiber Distributed Data Interface (dev_vfddi) Diagnostics Manual

Order No. DHW-276

First Edition
December 1991

CONVEX Press
Richardson, Texas
United States of America

CONVEX Fiber Distributed Data Interface (`dev_vfddi`*) Diagnostics Manual*
Order No. DHW-276
First Edition

Printed in the United States of America

Revision history

CONVEX Fiber Distributed Data Interface
(`dev_vfddi`) Diagnostics Manual

Edition	Document No.	Date	Description
First	760-004030-001	December 1991	First release.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Tables

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

Preface

Purpose and audience

This manual explains how to run the `dev_vfddi` diagnostic, which checks the CONVEX fiber distributed data interface. This document is not a tutorial, but rather a reference for users of the `dev_vfddi` 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 `dev_vfddi` 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 sections:

- **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, "Fiber distributed data interface test (`dev_vfddi`)"**—Describes how to operate the diagnostic, including prerequisites, test invocation, hardware initialization sequence, and class descriptions.

Terminology

The following terminology conventions are used in this text:

- Fiber distributed data interface is abbreviated as FDDI.
- 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.
- 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.
- 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 *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.

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.

All CONVEX illustrations have an illustration file number at the bottom right-hand corner that is for CONVEX use only.

Examples of typical warnings, cautions, and notes as used in CONVEX documents follow:

WARNING

A warning highlights procedures or information necessary to avoid injury to personnel. The warning immediately precedes the critical information and includes a description of the hazard.

CAUTION

A caution highlights procedures or information necessary to avoid damage to equipment, damage to software, loss of data, or invalid test results. The caution immediately precedes the critical information and includes a description of the possible damage.

NOTE

A note highlights information of a supplemental nature. The note immediately precedes or follows the highlighted information.

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 (3400 Series)*, Order No. DHW-302
- *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 "DHW" or "DSW."

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. Using `contact` requires:

- UNIX-to-UNIX Communication 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: Don May
- Review team: Al Haddix, Carol Brayton, Kris Meier, Alan Stolleis, Lihwen Wu, Don May, Brian Bianchi, John Rachels
- Editorial services: Clare Bernier, Sheri Roloff, and 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 "off-line"; 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 (C200 Series)* or *CONVEX Processor Diagnostics Manual (C3400 Series)*, depending on the architecture of the machine under test.

1.2 Operating system

The Event-Governed Operating System (EGOS) is a simple operating system that the device tests use to handle interrupts, schedule processes, and generally allocate and control input/output processor (IOP) and VMEbus input/output processor (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 hardware configuration, and initializes its windows and resource allocation tables.

There are four types of EGOS systems, one for each type of channel control unit (CCU). There is one each for the Multibus interface, the VMEbus interface, the high-speed parallel channel controller (HSP) interface, and the high-performance parallel interface (HIPPI). The following sections explain the four types of EGOS systems and how EGOS is positioned within the overall operating system environment.

1.2.1 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 supplies the error interrupt handlers for the CCU error interrupts. It also contains support routines to control allocation of the various CCU-related resources.

1.2.2 EGOS for VMEbus interface (VIOP EGOS)

The VMEbus 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 two 68020 interrupt levels. Therefore, two is the maximum number of processes at any given time.

1.2.3 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 supplies the error interrupt handlers for the CCU error interrupts. It also contains support routines to control allocation of the various CCU-related resources.

1.2.4 EGOS for HIPPI (HIPPI EGOS)

EGOS for the HIPPI interface supports event-driven device drivers as well as process-type device drivers. HIPPI EGOS uses 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 up to 256 process priorities and is preemptive for higher priority processes.

HIPPI EGOS helps device drivers handle hardware interrupts by providing interrupt prolog and epilog code. This code deals with all Motorola 88100 microprocessor requirements for interrupts and exceptions. This functionality allows interrupt procedures to be written in a high-level language and to easily use EGOS functions.

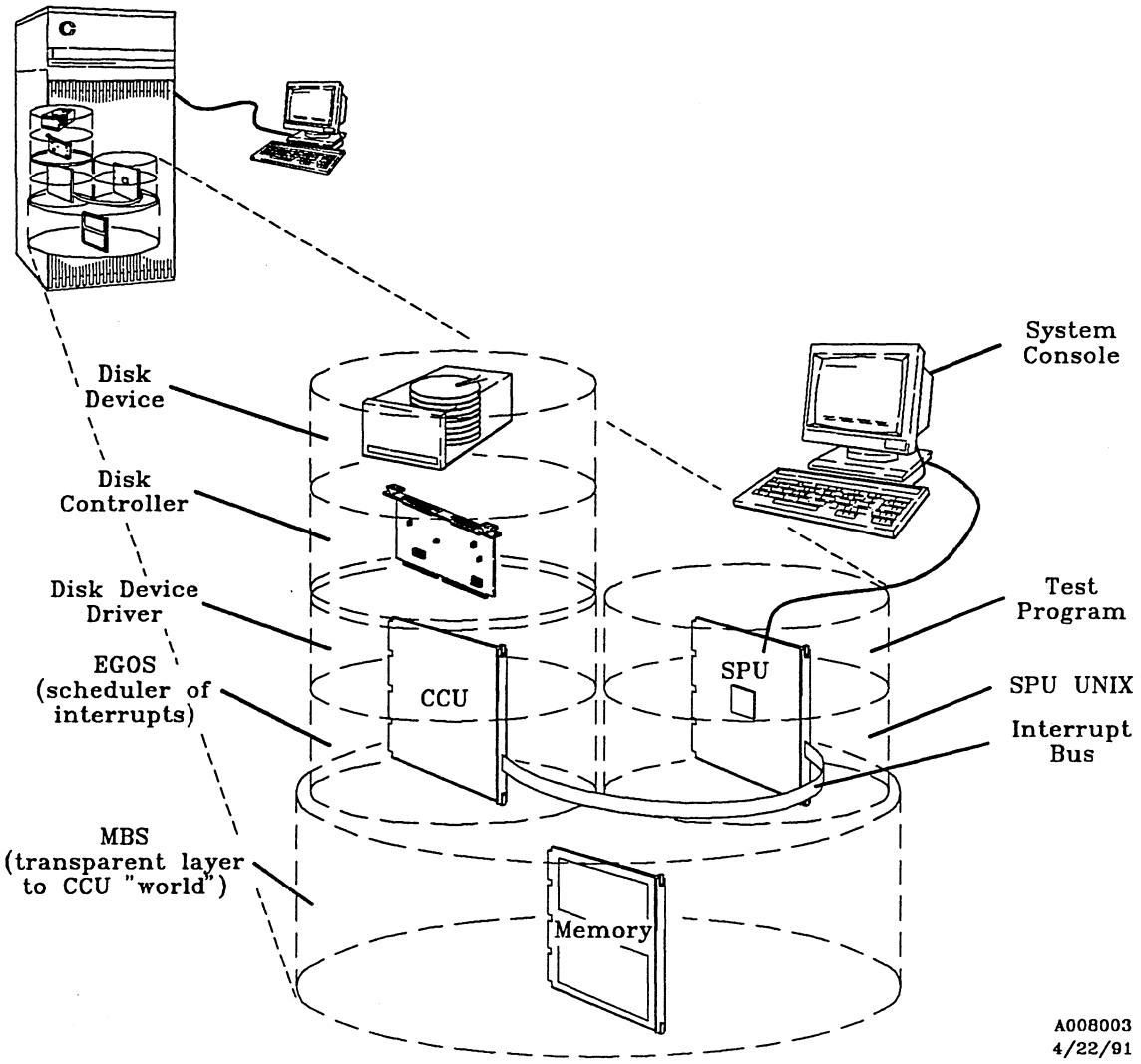
1.2.5 EGOS' position in the environment

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 OS. SPU OS 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 OS to be interrupted, and it retrieves the message from MBS. SPU OS 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.

Figure 1–1 illustrates the position of EGOS in the operating system environment.

Figure 1–1, EGOS' position in the environment



1.3 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).*

1.3.1 Diagnostic shell (dshell) overview

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
  - Select long or short error messages, or turn messages off
  - Execute either user-created or predefined command scripts

1.3.2 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`	Immediately terminates the `dshell` process and any test processes that may have been forked.
`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.

1.3.3 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 1–2 will be displayed on the screen.

Figure 1–2, 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
```

Chapter 2
Fiber distributed data interface test (dev_vfddi)

2.1 Overview

The dev_vfddi test is a functional test for the CONVEX fiber distributed data interface (FDDI) controller. This test verifies the proper operation of the CONVEX FDDI controller in the CONVEX I/O environment. dev_vfddi is designed to accomplish the following:

- Verify that the VIOP can access the controller's on-board RAM using the VMEbus.
- Execute the controller's built-in diagnostics.
- Verify that the controller can access the VIOP's local memory and the CONVEX machine's main memory using the VMEbus.
- Verify that the controller can communicate on an FDDI ring.
- Provide an interactive debugger that can aid in diagnosing a subtest failure.

The dev_vfddi test uses the same Event-Governed Operating System (EGOS) and Message-Based System (MBS) used by the ConvexOS operating system for all communications with the FDDI controller. This means that dev_vfddi tests the communication paths used in a normal operating environment.

2.2 Required equipment

Table 2–1 lists the minimum C2/3200 Series hardware required for the various classes of subtests. As indicated, some of the subtests require the FDDI controller under test to be connected to an FDDI ring. It is possible to use an active FDDI ring for this purpose.

Table 2–1, Hardware requirements

Subtests	Hardware requirements
All	Memory system[1]
All	CPX
All	SP2/SP4
All	PIA/PI2
All	VIOP
All	VBCU
All	VMEbus FDDI controller
Class 2	Fiber cable for loopback testing.
Class 4	An FDDI ring, consisting of the FDDI controller under test, at least one other active node, and enough fiber cable to connect the nodes in a ring configuration.
Class 5	The same requirements as those for class 4 subtests, except that at least one of the other active nodes on the FDDI ring must be another FDDI controller installed in the same C2/C3200 Series machine as the controller under test.

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

2.3 Test invocation

The `dev_vfddi` 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.

2.3.1 Initial test invocation

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

> **NOTE**
>
> Enter the following commands for the initial invocation of `dev_vfddi` 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)> sysreset
(spu)> mminit -s
(spu)> dshell
:test dev_vfddi [-option [...] ]
```

NOTE

After entering `dshell`, specific `dshell` parameters may be changed. Refer to Chapter 2, *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 controller firmware file name. If this option is omitted, /mnt/os/fddi.x00 is used as the controller firmware file name.
`-F` *file*	Use an alternate VIOP driver file name. If this option is omitted, /mnt/os/viop is used as the driver file name.
`-m`	Display all Common Message Interface (CMI) messages that pass between the VIOP and the SP2/SP4.
`-p` *file*	Use an alternate parameter file name. If this option is omitted, /tmp/dev_vfddi.tmp is used as the parameter file name.
`-q`	Quickly start the program by using the test parameters that are stored in the parameter file. This has the same effect as using the alternate invocation sequence.
`-t`	Disable CMI message timeouts.
`-v`	Print the version information for the program.
`-?`	Display a list of valid invocation options.
`+>` *file*	Capture the output from the program in a file, in addition to displaying it on the screen.

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

2.3.2 Alternate test invocation

Use the test invocation procedure shown in Figure 2–2 when invoking `dev_vfddi` multiple times. Using this sequence ensures that the test is invoked and executed with all the setup parameters supplied when the test was last executed with the initial invocation sequence.

Figure 2–2, Alternate test invocation sequence

```
(spu)> cd /mnt/test
(spu)> sysreset
(spu)> mminit -s
(spu)> dshell
:test dev_vfddix   [-option [...] ]
```

The only difference in this alternate invocation sequence is the `x` after `dev_vfddi`. When invoking `dev_vfddi` in this manner, no test parameter prompts are displayed. The diagnostic obtains all information from the parameter file created when the last initial invocation sequence was performed. The `mminit -s` and `sysreset` commands are required only if the state of the machine is unknown or if hard errors have occurred since the last system initialization.

2.4 Initialization sequence for `dev_vfddi`

Once the test is invoked, the diagnostic determines if the test was invoked for quick startup (either `dev_vfddix` or `dev_vfddi -q`). If so, the diagnostic reads the test parameters from the specified parameter file (default parameter file is /tmp/dev_vfddi.tmp). 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 VIOP driver code (if necessary).
5. 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.

2.4.1 Test parameter menu

If not invoked with `dev_vfddix`, the diagnostic displays test parameter prompts sequentially, allowing selection of default test parameters or specification of different values. Figure 2–3 shows all prompts, their possible answers (in brackets [ ]), and their default answers in parentheses ( ). 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 2–3, Test parameter menu

```
                    ENTER TEST PARAMETERS

       []      Encloses allowed input ranges or values
       ()      Encloses the default value
       ^       Returns to the previous prompt
       :nn     Returns to the prompt # nn
       :       Returns to the first unsatisfied prompt
       :?      Reviews previous entries
       ?       Provides additional help for each question

 1: Ioconfig File to Use [<filepath>,?]              (/ioconfig) -> RETURN

      PERIPHERAL CONFIGURATION DATA
    CCU    Chassis   Type    CSR   Int
   -------- ------- -------- ------ ---
 1) viop  0     0    LAN-208  0x6000  5
 2) viop  0     0    LAN-208  0x6200  6
 3) viop  1     1    LAN-208  0x6000  6

       *** Enter 0 for manual configuration ***

 2: Unit to Test [1-3,0,?]                                    (1) -> 0
 3: Test Unit VIOP CCU Slot Number [0-3,?]                    (0) -> RETURN
 4: Test Unit VMEbus Chassis Number [0-1,?]                   (0) -> RETURN
 5: Test Unit CSR [0x0-0xfc00,?]                         (0x6000) -> RETURN
 6: Test Unit Interrupt Level [1-7,?]                         (5) -> RETURN
 7: Perform Controller <-> Controller Subtests [y,n,?]        (n) -> y

      PERIPHERAL CONFIGURATION DATA
    CCU    Chassis   Type    CSR   Int
   -------- ------- -------- ------ ---
 1) viop  0     0    LAN-208  0x6000  5
 2) viop  0     0    LAN-208  0x6200  6
 3) viop  1     1    LAN-208  0x6000  6

       *** Enter 0 for manual configuration ***

 8: Second Unit to Test [1-3,0,?]                             (2) -> 0
 9: Second Unit VIOP CCU Slot Number [0-3,?]                  (0) -> RETURN
10: Second Unit VMEbus Chassis Number [0-1,?]                 (0) -> RETURN
11: Second Unit CSR [0x0-0xffe0,?]                      (0x6200) -> RETURN
12: Second Unit Interrupt Level [1-7,?]                      (6) -> RETURN
13: Perform External Loopback Subtests [y,n,?]               (y) -> RETURN
14: Prompt for External Loopback Cables [y,n,?]              (y) -> RETURN
15: Perform Optical Bypass Switch (OBS) Subtests [y,n,?]     (y) -> RETURN
16: Enter Debugger When Errors Found [y,n,?]                 (n) -> y
17: Use Defaults for Remaining Parameters [y,n,?]            (y) -> n

       ** Error Start/Finish Text Options (bit mapped) **
   0x0001: Enable error start text
   0x0002: Enable error finish text

18: Select Error Start/Finish Message Mode [0x0-0x3,?]      (0x0) -> 3
19: Error Start Character Sequence [<error start string>,?]
                                                    (\033[?5i)  -> RETURN
20: Error Finish Character Sequence [<error finish string>,?]
                                                    (\033[?4i)  -> RETURN
21: 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, followed by (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 the most recent prompt.

2.4.2 Prompt explanations

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

```
1: Ioconfig File to Use [<filepath>,?]                    (/ioconfig) ->
```

Enter the name of the /ioconfig file that contains information about the FDDI controllers installed in the C2/C3200 Series machine.

```
      PERIPHERAL CONFIGURATION DATA
     CCU     Chassis    Type     CSR    Int
    -------- ------- --------- ------ ---
1)  viop  0     0     LAN-208  0x6000  5
2)  viop  0     0     LAN-208  0x6200  6
3)  viop  1     1     LAN-208  0x6000  6

        *** Enter 0 for manual configuration ***

2: Unit to Test [1-3,0,?]                                          (1) ->
```

Enter the number of the FDDI controller to test. The controller information found in the specified /ioconfig file is displayed to aid in selection. If the controller is not listed, enter **0** at this prompt, and the next four prompts will allow you to enter information about the controller manually.

```
3: Test Unit VIOP CCU Slot Number [0-3,?]                          (0) ->
```

Enter the CCU slot number of the VIOP that communicates with the FDDI controller under test. This prompt is displayed only if the information for the test controller is being entered manually.

```
4: Test Unit VMEbus Chassis Number [0-1,?]                         (0) ->
```

Enter the number of the VMEbus chassis in which the FDDI controller under test is installed. This prompt is displayed only if the information for the test controller is being entered manually.

```
5: Test Unit CSR [0x0-0x6000,?]                                (0x6000) ->
```

Enter the VMEbus address of the controller status register (CSR) for the FDDI controller under test. This prompt is displayed only if the information for the test controller is being entered manually.

```
6: Test Unit Interrupt Level [1-7,?]                    (5) ->
```

Enter the VMEbus interrupt level for the FDDI controller under test. This prompt is displayed only if the information for the test controller is being entered manually.

```
7: Perform Controller <-> Controller Subtests [y,n,?]    (n) ->
```

> **NOTE**
>
> To run the class 4 diagnostic subtests, the FDDI controller must be connected to an FDDI ring containing at least one other node. If this ring has exactly one other node, and that node is in the same CONVEX machine as the controller under test, you must answer **y** to this prompt in order to run the class 4 subtests.

Enter **y** to execute subtests that test communication with other nodes on the FDDI ring. These are the class 5 subtests that require a second FDDI controller to be installed in the same C2/C3200 Series machine as the controller under test. If you enter **y**, the next five prompts will request information about this second controller.

```
      PERIPHERAL CONFIGURATION DATA
     CCU     Chassis    Type      CSR    Int
   -------- ------- --------- ------ ---
1) viop  0      0     LAN-208  0x6000  5
2) viop  0      0     LAN-208  0x6200  6
3) viop  1      1     LAN-208  0x6000  6

   *** Enter 0 for manual configuration ***

8: Second Unit to Test [1-3,0,?]                        (2) ->
```

Enter the number of the second FDDI controller to be used during the multinode subtests. The controller information found in the specified /ioconfig file is displayed to aid in selection. If the controller is not listed, enter **0** at this prompt, and the next four prompts will allow you to enter information about the controller manually.

```
9: Second Unit VIOP CCU Slot Number [0-3,?]             (0) ->
```

Enter the CCU slot number of the VIOP that communicates with the second FDDI controller. This prompt is displayed only if the information for the second controller is being entered manually.

```
10: Second Unit VMEbus Chassis Number [0-1,?]           (0) ->
```

Enter the number of the VMEbus chassis in which the second FDDI controller is installed. This prompt is displayed only if the information for the second controller is being entered manually.

```
11: Second Unit CSR [0x0-0xffe0,?]                  (0x6200) ->
```

Enter the VMEbus address of the controller status register (CSR) for the second FDDI controller. This prompt is displayed only if the information for the second controller is being entered manually.

```
12: Second Unit Interrupt Level [1-7,?]                    (6) ->
```

Enter the VMEbus interrupt level for the second FDDI controller. This prompt is displayed only if the information for the second controller is being entered manually.

```
13: Perform External Loopback Subtests [y,n,?]             (y) ->
```

Enter **y** to execute the external loopback subtests. These subtests require that one or two fiber cables be installed on the FDDI controller in a loopback configuration.

```
14: Prompt for External Loopback Cables [y,n,?]            (y) ->
```

Enter **y** to indicate that the external loopback subtests should prompt for the connection of the loopback cables. If the response to this prompt is **n**, the subtests will assume that the cables are already connected in a loopback configuration. This prompt is displayed only if the external loopback subtests are enabled by the previous prompt.

```
15: Perform Optical Bypass Switch (OBS) Subtests [y,n,?]   (y) ->
```

Enter **y** to perform class 6 subtests; these subtests verify the FDDI controller's ability to control an optical bypass switch (OBS).

```
16: Enter Debugger When Errors Found [y,n,?]               (n) ->
```

Enter **y** to enter debugger mode when a subtest failure occurs.

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

This option permits use of the default values for the remaining questions. If the response to this prompt is **y**, the remaining prompts is bypassed, and default values are used for their parameters.

```
    ** Error Start/Finish Text Options (bit mapped) **
0x0001: Enable error start text
0x0002: Enable error finish text

18: Select Error Start/Finish Message Mode [0x0-0x3,?]  (0x0) ->
```

This prompt allows a specified character sequence to be sent to the display terminal before or after (or both) an error (and its data) is displayed. The intended use for this feature is to selectively turn a printer on before an error message is displayed and to turn a printer off after an error message has been displayed. This ability saves paper when running the diagnostic multiple times or for long periods of time. It assumes a printer is connected to the auxiliary port on the display terminal where the diagnostic is running. It also assumes the auxiliary port can be turned on and off using an escape character sequence.

The response to this prompt should be the sum of the values for the desired options. For example, to enable the character sequences both before and after a failure message, enter the value 0x0003 (or simply 3), which is the sum of 0x0001 and 0x0002.

```
19: Error Start Character Sequence [<error start string>,?]
                                                    (\033[?5i) ->
```

Enter the special character sequence that should be sent to the display terminal before a subtest failure message is displayed. Any sequence of up to 64 characters may be used. Control characters may be included by using standard C-language-style escape sequences, as shown in the following examples.

`\033`	Octal value of the ASCII escape character (0x1b)
`\x1b`	Same as above value, but specified in hex
`\r`	Carriage return
`\n`	Line feed
`\t`	Tab character
`\b`	Backspace character
`\f`	Form feed character
`\a`	Bell ("alert")
`\v`	Reverse line feed character

The default sequence for this prompt is the "Enter Auto Print Mode" sequence for terminals that support the VT100 terminal protocol.

```
20: Error Finish Character Sequence [<error finish string>,?]
                                                    (\033[?4i) ->
```

Enter the special character sequence that should be sent to the display terminal after a subtest failure message is displayed. Any sequence of up to 64 characters may be used. Control characters may be included by using standard C-language-style escape sequences, as shown in the following examples.

`\033`	Octal value of the ASCII escape character (0x1b)
`\x1b`	Same as above value, but specified in hex
`\r`	Carriage return
`\n`	Line feed
`\t`	Tab character
`\b`	Backspace character
`\f`	Form feed character
`\a`	Bell ("alert") character
`\v`	Reverse line feed character

The default sequence for this prompt is the "Exit Auto Print Mode" sequence for terminals that support the VT100 terminal protocol.

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

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.

2.4.3 Test parameter summary

When all test parameter prompts are answered, the test program displays a TEST PARAMETER SUMMARY menu that shows the prompts that were answered and their responses. Figure 2–4 illustrates an example of a TEST PARAMETER SUMMARY screen. Actual values and responses vary according to the input.

Figure 2–4, Sample test parameter summary

```
                TEST PARAMETER SUMMARY

Ioconfig File to Use                                    : ioconfig
Unit to Test                                            : 0
Test Unit VIOP CCU Slot Number                          : 0
Test Unit VMEbus Chassis Number                         : 0
Test Unit CSR                                           : 0x6000
Test Unit Interrupt Level                               : 5
Perform Controller <-> Controller Subtests              : y
Second Unit to Test                                     : 0
Second Unit VIOP CCU Slot Number                        : 0
Second Unit VMEbus Chassis Number                       : 0
Second Unit CSR                                         : 0x6200
Second Unit Interrupt Level                             : 6
Perform External Loopback Subtests                      : y
Prompt for External Loopback Cables                     : y
Perform Optical Bypass Switch (OBS) Subtests            : y
Enter Debugger When Errors Found                        : y
Use Defaults for Remaining Parameters                   : n
Select Error Start/Finish Message Mode                  : 0x0003
        -> Enable error start text <-
        -> Enable error finish text <-
Error Start Character Sequence                          : \033[?5i
Error Finish Character Sequence                         : \033[?4i
Enter OK, or :NN to return to question NN               : OK
```

2.5 Class descriptions

Table 2–4 lists the six classes of subtests contained in `dev_vfddi`.

Table 2–4, `dev_vfddi` test classes

Class	Description
1	Built-in diagnostics tests
2	External loopback tests
3	Generic RC command verification tests
4	FDDI ring access tests
5	Multi-node communication tests
6	Optical bypass switch (OBS) tests

All subtests contained in `dev_vfddi` may be looped under the `dshell`.

2.6 Class 1 subtests

The class 1 subtests exercise the low-level functions of the FDDI controller. These subtests verify that the VIOP can communicate with the FDDI controller using the VMEbus and that the controller's memory and internal signal paths function properly. Most of these subtests use the FDDI controller's built-in diagnostic firmware to accomplish these tasks.

Table 2–5 lists the class 1 subtests, their descriptions, and the time required to execute each subtest.

Table 2–5, Class 1 subtests

Subtest	Description	Time (seconds)
1000	Power-up diagnostic result verification	00:10
1010	Controller short I/O RAM test	00:05
1020	Extended static RAM test	00:10
1030	Extended buffer RAM test	00:25
1040	Extended shared RAM test	00:05
1050	VMEbus short I/O RAM access verification	00:05
1060	VIOP RAM access verification	00:05
1070	Main memory access verification	00:05
1080	VMEbus address test	00:05

2.6.1 Subtest 1000, power-up diagnostic result verification

Immediately after the FDDI controller is powered up, the controller's CPU executes a series of built-in diagnostic tests. This series of tests is referred to as the power-up diagnostics, and it verifies the operation of most of the controller's hardware. The power-up diagnostics are also executed when the controller is reset by the VIOP. This subtest verifies that the controller's power-up diagnostics complete successfully.

2.6.2 Subtest 1010, controller short I/O RAM test

The FDDI controller contains an area of RAM that the VIOP can access using the short I/O VMEbus addressing space. This subtest verifies that the controller's short I/O RAM can be accessed by the VIOP and is functioning properly.

2.6.3 Subtest 1020, extended static RAM test

The FDDI controller's built-in diagnostics include a test that verifies the operation of its on-board static RAM. The controller's power-up series of tests includes only a quick version of this test, however. This subtest executes the extended version of the controller's built-in static RAM test.

2.6.4 Subtest 1030, extended buffer RAM test

The FDDI controller's built-in diagnostics include a test that verifies the operation of its on-board buffer RAM. The controller's power-up series of tests includes only a quick version of this test, however. This subtest executes the extended version of the controller's built-in buffer RAM test.

2.6.5 Subtest 1040, extended shared RAM test

The FDDI controller's built-in diagnostics include a test that verifies the operation of its on-board shared RAM. The controller's power-up series of tests includes only a quick version of this test, however. This subtest executes the extended version of the controller's built-in shared RAM test.

2.6.6 Subtest 1050, VMEbus short I/O RAM access verification

To exchange data with the VIOP, the FDDI controller performs DMA cycles on the VMEbus. This subtest verifies that the controller can access its own short I/O RAM using the VMEbus.

2.6.7 Subtest 1060, VIOP RAM access verification

This subtest is similar to subtest 1050 except that it verifies that the controller can access the VIOP's local RAM.

2.6.8 Subtest 1070, main memory access verification

This subtest is similar to subtest 1050 except that it verifies that the controller can access the CONVEX machine's main memory.

2.6.9 Subtest 1080, VMEbus address test

To communicate with the VIOP, the FDDI controller performs DMA cycles on the VMEbus. This subtest verifies that the controller can properly generate addresses on the VMEbus.

2.7 Class 2 subtests

The class 2 subtests verify that the FDDI controller's main data paths operate properly. These subtests require that a loopback fiber cable be physically connected to the controller's fiber-optic connectors.

Table 2–6 lists the class 2 subtests, their descriptions, and the time required to execute each subtest.

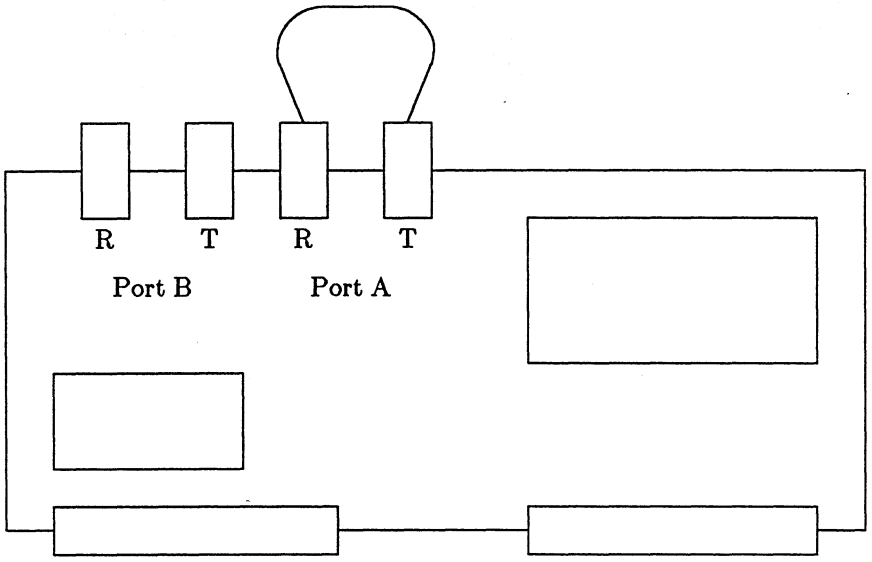
Table 2–6, Class 2 subtests

Subtest	Description	Time (seconds)
2000	External A port loopback test	00:10
2010	External B port loopback test	00:10

2.7.1 Subtest 2000, external A port loopback test

This subtest verifies the operation of the controller's data paths on the A port by transmitting and receiving frames in a loopback configuration. This subtest requires that either a loopback fiber cable or a loopback connector assembly be connected between this port's optic transmitter and receiver as shown in Figure 2–5.

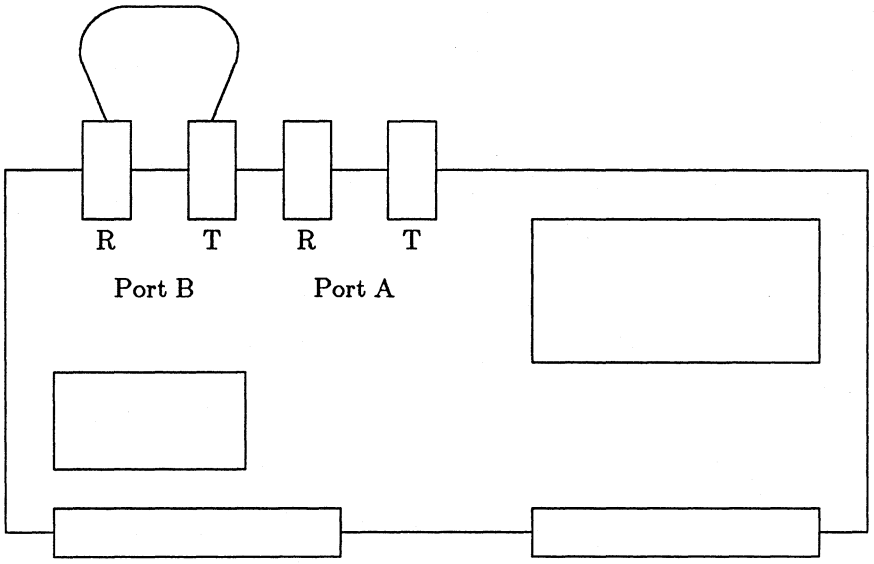
Figure 2–5, Fiber cable connection for subtest 2000



2.7.2 Subtest 2010, external B port loopback test

This subtest is similar to subtest 2000 except that it executes the external loopback test on the B port. This subtest requires that either a loopback fiber cable or a loopback connector assembly be connected between this port's optic transmitter and receiver as shown in Figure 2–6.

Figure 2–6, Fiber cable connection for subtest 2010



2.8 Class 3 subtests

The class 3 subtests verify that the FDDI controller's RC interface operates properly. The RC interface is the method by which the VIOP and the controller exchange commands and responses.

Table 2–7 lists the class 3 subtests, their descriptions, and the time required to execute each subtest.

Table 2–7, Class 3 subtests

Subtest	Description	Time (seconds)
3000	Controller boot verification	00:40
3010	VMEbus interrupt test	00:20
3020	Controller RC segment chaining test	00:10[1]

[1] This subtest runs an additional 20 seconds (approximately) if it is the first class 3 subtest executed.

2.8.1 Subtest 3000, controller boot verification

After the FDDI controller is powered up or reset, it responds to only a limited number of commands from the VIOP. To enable the remaining controller commands, the VIOP must boot the controller's firmware. This subtest issues a boot command to the controller and verifies the controller's response. It then resets the controller and again verifies its response. Finally, this subtest issues another boot command and verifies the controller's response.

2.8.2 Subtest 3010, VMEbus interrupt test

The FDDI controller can be configured to interrupt the VIOP whenever it has sent a response or indication. These interrupts are generated on the VMEbus. This subtest verifies that the controller generates VMEbus interrupts correctly.

2.8.3 Subtest 3020, controller RC segment chaining test

The FDDI controller and the VIOP communicate using a message passing scheme called the *RC interface*. To pass a message, the sender places it in an RC channel, which is an area in VIOP local memory reserved for this purpose. Only the owner of a channel can place a message in it. Therefore, there are two channels defined—one owned by the VIOP, and one owned by the FDDI controller. Each channel is divided into a number of buffers, or segments, which do not have to be contiguous. When a segment is filled by its owner, it informs the channel's receiver of the location of the next segment. Usually, the segments within a channel are linked together in a circular list so that old segments can be reused.

This subtest verifies that the controller properly follows RC segment chains in both the VIOP-to-controller and controller-to-VIOP RC channels.

2.9 Class 4 subtests

The class 4 subtests verify the FDDI controller's ability to access the FDDI ring. These subtests require that the controller be connected to a fiber ring with at least one other node. It is possible to use an active FDDI ring for this purpose.

Table 2–8 lists the class 4 subtests, their descriptions, and the time required to execute each subtest.

Table 2–8, Class 4 subtests

Subtest	Description	Time (seconds)
4000	Ring status test	00:35
4010	Default-address recognition test	00:25
4020	Modified-address recognition test	02:50
4030	Broadcast-address recognition test	00:10[1]
4040	CAM access test	02:30[1]
4050	CAM operation test	01:30[1]
4060	Scatter/gather test	00:05[1]
4070	VMEbus 16-bit data access test	00:05[1]
4080	Receive buffer starvation test	00:10[1]

[1] This subtest runs an additional 30 seconds (approximately) if it is the first class 4 subtest executed.

2.9.1 Subtest 4000, ring status test

The FDDI controller is able to logically connect and disconnect from the FDDI ring. It can also report to the VIOP the status of its connection to the fiber ring. This subtest verifies the controller's ability to properly connect to the FDDI ring, disconnect from the ring, and report its ring connection status.

2.9.2 Subtest 4010, default-address recognition test

The manufacturer of the FDDI controller embeds into the controller's hardware a station address that it can use as its FDDI ring node address. This subtest verifies the controller's ability to recognize FDDI frames that are sent to its default, manufacturer-supplied station address.

2.9.3 Subtest 4020, modified-address recognition test

This subtest verifies the controller's ability to change its default, manufacturer-supplied station address and to recognize FDDI frames that are sent to this modified station address.

2.9.4 Subtest 4030, broadcast-address recognition test

This subtest verifies the controller's ability to receive FDDI broadcast frames. Broadcast frames use a special destination address that is recognized by all nodes on the FDDI ring.

2.9.5 Subtest 4040, CAM access test

The VIOP can configure the FDDI controller to recognize up to 256 station addresses, which is useful for multicast messages. These addresses are stored in a content-addressable memory (CAM) on the controller. This subtest verifies the controller's ability to properly add and remove addresses from its CAM.

2.9.6 Subtest 4050, CAM operation test

This subtest verifies that the FDDI controller can receive frames addressed to the multicast addresses stored in its CAM.

2.9.7 Subtest 4060, scatter/gather test

The FDDI controller reads and writes FDDI frame data in the CONVEX machine's main memory. The data from a single frame can be distributed among many noncontiguous locations in main memory. The process of collecting the frame data for transmission is called gathering. Distribution of received frame data is called scattering. This subtest verifies the controller's ability to scatter and gather message data.

2.9.8 Subtest 4070, VMEbus 16-bit data access test

The FDDI controller can transfer data on the VMEbus using 16-bit and 32-bit accesses. Normally, the `dev_vfddi` program instructs the controller to use 32-bit accesses. This subtest verifies the controller's ability to transfer VMEbus data using 16-bit accesses.

2.9.9 Subtest 4080, receive buffer starvation test

The FDDI controller places data from frames that it receives from the FDDI ring into memory buffers specified by the VIOP. These receive-data memory buffers are located in the VIOP's local RAM. If there are no receive buffers available when a frame is received, the controller temporarily stores the frame data in its own local RAM.

This subtest verifies the controller's ability to store frame data in its own local RAM whenever there is a shortage of receive data memory buffers on the VIOP.

2.10 Class 5 subtests

The class 5 subtests verify the FDDI controller's ability to communicate with other nodes on the FDDI ring. These subtests require that both the controller under test and a second FDDI controller be connected to a fiber ring. Furthermore, the second controller must be in the same C2/C3200 Series machine as the controller under test. It is possible to use an active FDDI ring for this purpose.

Table 2–9 lists the class 5 subtests, their descriptions, and the time required to execute each subtest.

Table 2–9, Class 5 subtests

Subtest	Description	Time (seconds)
5000	Multi-node synchronous raw data test	00:10[1]
5010	Multi-node synchronous datagram test	00:10[1]
5020	Multi-node asynchronous raw data test	00:10[1]
5030	Multi-node asynchronous datagram test	00:10[1]

[1] This subtest runs an additional 20 seconds (approximately) if it is the first class 5 subtest executed.

2.10.1 Subtest 5000, multi-node synchronous raw data transfer test

The FDDI controller is capable of transmitting data on the FDDI ring in either synchronous or asynchronous mode. The data that is transmitted contains header information that can be supplied by the VIOP or the controller. If supplied by the VIOP, the TRANSMIT RAW DATA RC request is used. If the controller supplies the header information, the *TRANSMIT DATAGRAM* request is used.

This subtest verifies the FDDI controller's ability to transmit and receive raw data in synchronous mode.

2.10.2 Subtest 5010, multi-node synchronous datagram transfer test

This subtest is similar to subtest 5000 except that it uses the TRANSMIT DATAGRAM RC request to transmit and receive datagrams in synchronous mode.

2.10.3 Subtest 5020, multi-node asynchronous raw data transfer test

This subtest is similar to subtest 5000 except that it transmits and receives raw data in asynchronous mode.

2.10.4 Subtest 5030, multi-node asynchronous datagram transfer test

This subtest is similar to subtest 5020 except that it uses the TRANSMIT DATAGRAM RC request to transmit and receive datagrams in asynchronous mode.

2.11 Class 6 subtest

The FDDI controller is capable of operating with an optical bypass switch (OBS). The class 6 subtest verifies the FDDI controller's ability to control the OBS. This subtest requires that an OBS be connected to the FDDI controller.

Table 2–10 lists the class 6 subtest, its description, and the time required to execute it.

Table 2–10, Class 6 subtest

Subtest	Description	Time (seconds)
6000	Optical bypass switch test	01:00

This subtest verifies the FDDI controller's ability to change the mode of the optical bypass switch. An OBS has two modes of operation: bypass and thru. When in bypass mode, it isolates the controller from the FDDI ring. When in thru mode, it allows the controller to communicate on the ring.

2.12 Interactive debugger

The diagnostic provides an interactive debugger that allows the operator to perform some low-level operations that are helpful when diagnosing a subtest failure.

The following conventions are used in the debugger command descriptions:

- Words in **boldface** indicate operator-entered information that must be entered exactly as it appears.
- Words in *italics* indicate placeholders for operator-entered information that must be replaced with actual parameter values.
- Words enclosed in brackets [ ] indicate optional operator-entered information.
- Numeric values are displayed as hexadecimal values, and operator-entered numeric values are interpreted as hexadecimal values, that is, 400 is interpreted by the debugger as 0x400 (1024).
- The term *test unit* refers to the FDDI controller under test.
- The term *secondary unit* refers to the second FDDI controller that must be installed in the same C2/C3200 Series machine during the multi-node communication subtests.

For example, **get_mib** *id* [*index*] indicates that you must enter the word **get_mib** followed by an *id* value. Optionally, you may enter an *index* value after the ID value.

Once the interactive debugger is entered, online help commands are available. Figure 2–7 illustrates the display shown when you enter **help**.

Figure 2–7, 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.

    help    [COMMAND ...]                          - display general or specific help
    echo    [-n] [arg ...]                         - echo statements to display
    pause   [-n] [seconds]                         - pause for <C/R> or seconds
    quit                                           - exit debug mode
    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
    boot_rc                                        - reset unit and boot RC interface
    connect_ring                                   - connect to ring
    disconnect_ring                                - disconnect from ring
    fill_tx_data  begin value [value ...]          - fill longs in TX data buffer
    filter_smt    [0 | 1 | 2]                      - enable/disable SMT frame filter
    get_mib       id [index]                       - get an MIB parameter
    get_rc_response                                - show RC response/indication
    init_ccu  [-f] [CCU [chas [CSR [int]]]]        - load CCU driver code
    last_response                                  - show last RC response/indication
    mbs_message                                    - show last MBS message from CCU
    status                                         - show various diag parameters
    rc_cmd        timeout cmd [parm ...]           - execute RC request/directive
    show_rx_data  [addr]                           - show RX buffer data
    smt_trace     <1 | 2 | 3 | 4>                  - SMT frame/event trace control
    swap_units                                     - swap test unit and second unit
    tx_raw        count [class [options]]          - transmit data
    drain_responses [-q]                           - get all pending RC resp/ind
    watch_responses                                - monitor RC responses/indications
```

In addition to the help screen in Figure 2-7, 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, you enter **help r**.

2.13 Interactive debugger command descriptions

This section describes each command in the interactive debugger.

2.13.1 boot_rc

Usage: `boot_rc`

Resets the test unit, boots its RC interface, enables its VMEbus interrupts, and establishes its receive buffer list.

2.13.2 connect_ring

Usage: `connect_ring` [**-n**]

Instructs the test unit to logically connect to the FDDI ring, where **-n** means do not wait for the connection to complete.

2.13.3 disconnect_ring

Usage: `disconnect_ring`

Instructs the test unit to logically disconnect from the FDDI ring.

2.13.4 drain_responses

Usage: `drain_responses` [**-q**]

Retrieves and displays all pending RC responses or indications in the response queue for the FDDI controller under test, where **-q** indicates to retrieve but not display the responses or indications.

2.13.5 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.

2.13.6 fb, fw, fl

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

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

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

Fills areas of VIOP address space with a specified pattern in byte-at-a-time mode (`fb`), word-at-a-time mode (`fw`), or longword-at-a-time move (`fl`), where:

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

The first format (for example, `fb` *begin value*) 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.

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

The following example illustrates the use of this command.

```
Debug Mode->fw 300 308 1 1
```

In the example, `fw` fills VIOP locations 0x300 to 0x308 with the following values:

```
0x300 = 0x0001
0x302 = 0x0002
0x304 = 0x0003
0x306 = 0x0004
0x308 = 0x0005
```

2.13.7 ffb, ffw, ffl

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

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

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

Fills main memory with specified pattern in byte-at-a-time mode (`ffb`), word-at-a-time mode (`ffw`), or longword-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 (for example, `ffb` *begin value*) stores *value* at address *begin*.

The second format (for example, `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.

2.13.8 fill_tx_data

Usage: `fill_tx_data` *begin value* [*value* [...]]

Fills the TX data buffer with long-word values (32-bit quantities). The TX data buffer is used as the data source for frames that are transmitted on the FDDI ring using the `tx_raw debugger` command.

The parameters are

begin Long-word offset in the TX data buffer for the first value.

value One or more long-word values to be stored in the TX data buffer. If more than one *value* parameter is specified, the values are stored in the data buffer sequentially.

The following example illustrates the use of this command.

Debug Mode-> **fill_tx_data 3 1234 abcdef01**

In the example, `fill_tx_data` stores two long-words of data starting at long-word offset 3 in the TX data buffer, that is, 0x00001234 is stored at offset 3, and 0xabcdef01 is stored at offset 4.

2.13.9 filter_smt

Usage: `filter_smt` *action*

Enables and disables SMT frame filtering on the test unit. When enabled, the `show_rx_data` debugger command will not display SMT frames unless they are SMT echo-response frames. When `dev_vfddi` is invoked, SMT frame filtering is enabled.

action is one of the following values:

0 Disable SMT frame filtering

1 Enable SMT frame filtering

2 Enable SMT frame filtering and only show SMT echo responses (even LLC frames are filtered out in this case)

If this parameter is omitted, the current SMT frame filtering state is displayed.

2.13.10 get_mib

Usage: `get_mib` *id* [*index*]

Requests an MIB parameter from the test unit, where:

- *id* is the ID of the requested MIB parameter (from the SMT specification)
- *index* is the MAC, port, path or attachment index for the MIB parameter. If this parameter is omitted, a value of 0 is used, which denotes an SMT parameter.

2.13.11 get_rc_response

Usage: `get_rc_response`

Displays the next RC response or indication in the board-to-host channel for the test unit. This command also informs the test unit that the response or indication has been removed from the channel.

2.13.12 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

2.13.13 init_ccu

Usage: `init_ccu` [**-f**] [*CCU* [*chassis* [*CSR* [*interrupt*]]]]

Loads and probes the VIOP driver code (if necessary) and sets the parameters for the test unit. If the controller parameters are different than the current test unit's parameters, the current test unit becomes the secondary unit.

The parameters are

`-f`	Load the VIOP driver even if it's already loaded
CCU	CCU slot number for the test unit's VIOP
chassis	VMEbus chassis for the test unit
CSR	CSR address for the test unit
interrupt	VMEbus interrupt level for the test unit

The following example illustrates the use of this command.

```
Debug Mode->init_ccu 4 1 6000 3
```

In the example, `init_ccu` loads the driver code into the VIOP in CCU slot 4 (if it's not already loaded) and prepares to use the FDDI controller in VMEbus chassis 1 at CSR 0x6000 as the test unit. It also configures the test unit to use VMEbus interrupt level 3.

2.13.14 last_response

Usage: `last_response`

Displays the last RC response or indication that was received from the test unit.

2.13.15 mb, mw, ml

Usage: **mb** *begin,count*
mb *begin end* [*step*]
mb *begin*

mw *begin,count*
mw *begin end* [*step*]
mw *begin*

ml *begin,count*
ml *begin end* [*step*]
ml *begin*

Displays and/or modifies areas of VIOP 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 address
- *end* is the ending address
- *count* is the number of elements to display
- *step* is the address increment

The first format (for example, `mb` *begin,count*) displays *count* number of elements from the starting address.

The second format (for example, `mb` *begin end*) displays all elements from the address *begin* up to and including address *end*.

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

[<*value*>]	Write optional <value> to current address, advance to next address
[<*value*>]=	Write optional <*value*> to current address, and stay at the present address (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-> mw c03fc2
 <CCU:00c03fc2> = 2a1c  0000=ffff,1q
 <CCU:00c03fc2> = 0000
 <CCU:00c03fc4> = 4378
```

where **0000=ffff,1q** is an example of executing multiple commands on the same line. This sequence displays the word value at VIOP address c03fc2 and allows the operator to modify this value. The operator's response modifies the word to 0, rereads the value at c03fc2, modifies the value again to ffff, skips to address c03fc4, modifies its value to 1, then exits the interactive mode.

2.13.16 mbs_message

Usage: `mbs_message`

Displays the last MBS/CMI message that was received from the VIOP.

2.13.17 mmb, mmw, mml

Usage: **mmb** *begin,count*
mmb *begin end* [*step*]
mmb *begin*

mmw *begin,count*
mmw *begin end* [*step*]
mmw *begin*

mml *begin,count*
mml *begin end* [*step*]
mml *begin*

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

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

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

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

[<*value*>]	Write optional <value> to current address, advance to next address
[<*value*>]=	Write optional <*value*> to current address, and stay at the present address (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
```

where **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.

2.13.18 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.

2.13.19 quit

Usage: `quit`

Exits the interactive debugger.

2.13.20 rc_cmd

Usage: `rc_cmd` *timeout cmd* [*parm* [...]]

Sends an RC request or directive to the test unit and displays the RC response or indication received from the test unit, where:

timeout is the number of seconds to wait for a response/indication before indicating an error. If this parameter is 0, `dev_vfddi` will not wait for a response/indication.

cmd is the identification of the RC request/directive.

parm is zero or more long-word parameters for the command request/directive.

The following example illustrates this command:

```
Debug Mode-> rc_cmd 3 104 0 54
```

In the example, `rc_cmd` instructs the FDDI controller to perform RC command 0x104 (REQUEST STATION ADDRESS) with parameters 0x0 and 0x54. The `dev_vfddi` program then waits up to 3 seconds for an RC response or indication from the controller.

2.13.21 show_rx_data

Usage: `show_rx_data` [*addr*]

Displays data from a frame that was received by the test unit. This command also updates the receive buffer list to allow the test unit to reuse the RX data buffer(s) corresponding to the received frame.

addr is the main memory address of the RX data buffer that contains data for the received frame. If this parameter is omitted, and the next response in the test unit's board-to-host channel is an Rx FRAME indication, the corresponding RX data buffer(s) is used as the default.

2.13.22 smt_trace

Usage: `smt_trace` *action*

Enables and disables SMT frame and event tracing on the test unit.

action is one of the following values:

1 Enable SMT frame tracing

2 Disable SMT frame tracing

3 Enable SMT event tracing

4 Disable SMT event tracing

2.13.23 status

Usage: `status`

Displays various parameters for the test unit and the secondary unit.

2.13.24 swap_units

Usage: `swap_units`

Changes the focus of the `dev_vfddi` program such that the test unit becomes the secondary unit, and vice versa.

2.13.25 tx_raw

Usage: `tx_raw` *count* [*class* [*options*]]

Instructs the test unit to transmit the data in the TX data buffer onto the FDDI ring. See the `fill_tx_data` command for more information about the TX data buffer.

The parameters are

count Number of bytes of data to transmit. This number should include the three pad bytes specified in the MAC and LLC headers.

class Request class value created by adding values from the following list:

1 Enable response

2 Asynchronous transmission

If this parameter is omitted, the value of 1 (enable response) is used as the default.

options Transfer options word created by adding the desired VMEbus address modifier value to a single value from the following list:

100	16-bit VMEbus transfers
200	32-bit VMEbus transfers

If this parameter is omitted, the value 23d (32-bit, standard, supervisory, data access) is used as the default.

The following example illustrates this command:

```
Debug Mode-> tx_raw 24
```

In the example, `tx_raw` instructs the FDDI controller to send the first 36 (0x24) bytes in the TX data buffer onto the FDDI ring.

2.13.26 watch_responses

Usage: `watch_responses`

Continually watches for and displays RC responses and indications from the test unit. To terminate this function, press CTRL-C .

Index

N

O

P

Q

R

S

T

U

W