September, 1989
Order Number: 314305-001

iPSC®/2 DIAGNOSTICS

REFERENCE MANUAL

NOTICE

Material in this document is for informational purposes only and is subject to change without notice. Intel Corporation assumes no responsibility for any errors that may appear in this document.

intel® Corporation

REV.	REVISION HISTORY	DATE
	Original issue	11/88
001	Revision	8/89

RESTRICTED RIGHTS

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the rights in Technical Data and Computer Software clause at 52.227-7013. Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION

CHAPTER 2 OPERATION

CHAPTER 3 CUBE DIAGNOSTIC PROGRAM

APPENDIX A POWER UP TESTS

APPENDIX B NCT SPECIAL FEATURES

APPENDIX C NODE BOOT MONITOR

APPENDIX D DIAGNOSTIC CHANNEL LIBRARY

APPENDIX E CDP TEST MANAGER LIBRARY

APPENDIX F LOOPBACK CONNECTORS

LIST OF ILLUSTRATIONS

LIST OF TABLES

INTRODUCTION 1

PURPOSE

This manual provides detailed information for a typical Intel Manufacturing, Engineering, or Field Service engineer who must run, understand, and enhance the iPSC/2 diagnostics. This manual is not intended for iSC customers.

SCOPE

This manual assumes that you know the iPSC/2 system theory of operation. If you do not, you should read the *iPSC®/2 User's Guide* and *iPSC®/2 System Adminstrator's Guide*. It is also suggested that you attend the user's training class. A class is also provided for Field Service engineering.

ORGANIZATION

Chapter 1 describes this manual and other applicable documents. In addition, it introduces the Diagnostics Programs with a brief explanation of scope and purpose. It also includes basic descriptions of the iPSC/2 Diagnostic Hardware Model.

Chapter 2 describes the operation of the iPSC/2 Diagnostic Programs.

Chapter 3 describes the Cube Diagnostic Program (CDP), including hardware under test, test algorithms, and error messages.

Appendix A describes the iPSC/2 Power Up Tests, including the hardware under test, test algorithms, and error messages / indicators produced.

Appendix B describes the special features of the Node Confidence Test (NCT) firmware that provide assistance in troubleshooting node boards to the component-level.

Appendix C describes the node board Boot Monitor firmware and how this firmware loads software code and dumps memory contents.

Appendix D describes the C library functions that manipulate the Unit Services Modules (USMs). These functions provide an interface for controlling the node board resets, interrupts, and communications without requiring understanding of the low-level details of the USMs.

Appendix E describes the C library functions for creating custom test suites for the Cube Diagnostic Program.

Appendix F describes the system loopback connectors.

APPLICABLE DOCUMENTS

For more information, refer to the other manuals of the iPSC/2 set. These include the following:

iPSC®/2 User's Guide
Provides a general description of the iPSC/2 system.

iPSC®/2 Installation Manual
Provides instructions for installing the iPSC/2 system.

iPSC®/2 System Adminstrator's Guide
Provides adminstrative information and briefly describes the iPSC/2 system.

iSBC® 386AT User's Guide
Provides information about the System Resource Manager Power On Self Test.

QA+
Provides information about the diagnostics that run on the iSBC® 386AT.

Maxtor Disk Manual
Provides information about the hard disk drives.

INTRODUCTION TO THE DIAGNOSTICS

The iPSC/2 diagnostic programs check the hardware at three distinct stages in the power-up and booting process. Each stage corresponds to an increasingly higher level of confidence in the functionality of the system. The stages are:

- **Power Up.** Three EPROM-resident diagnostic tests are automatically activated when power is applied to the iPSC/2 System. First, the Power On Self Test (POST) verifies the operation of the basic components in the System Resource Manager (SRM). Second, the Unit Services Module (USM) verifies its own internal operation and resets the node boards. Third, the Node Confidence Test (NCT) verifies the operation of every node board in each computational unit. Boot Monitors associated with the POST and the NCT are also activated at power-on. For more information, refer to Appendix A, "Power Up Tests", and Appendix B, "NCT Special Features".

- **Stand-alone.** Once power has been applied to the system, but before the SRM boots into its operating system (UNIX), the QA+ series of tests, available on diskette, may be run. These tests check the SRM boards and disk drives to guarantee hardware functionality and to facilitate the debugging of intermittent failures.

- **Hard Disk-Resident.** After the Power-Up and Stand-alone tests have run and UNIX has booted on the SRM, the iPSC/2 system communications links and the individual node boards may be further tested by using the hard disk-resident Cube Diagnostics Program (CDP). For more detailed information, refer to Chapter 3, "Cube Diagnostic Program".

Diagnostic Hardware Model

A model of the iPSC/2 system is shown in Figure 1-1. This model is used in the manual to describe the diagnostic tests and how the tests apply to a strategy of fault isolation. The model is a sample representation of the iPSC/2 system. The user is expected to be technically proficient enough to apply this basic model to all iPSC/2 system configurations. The user is also expected to be technically qualified and trained on the iPSC/2 theory of operation. It is not the intent of this manual to teach you how the iPSC/2 operates.

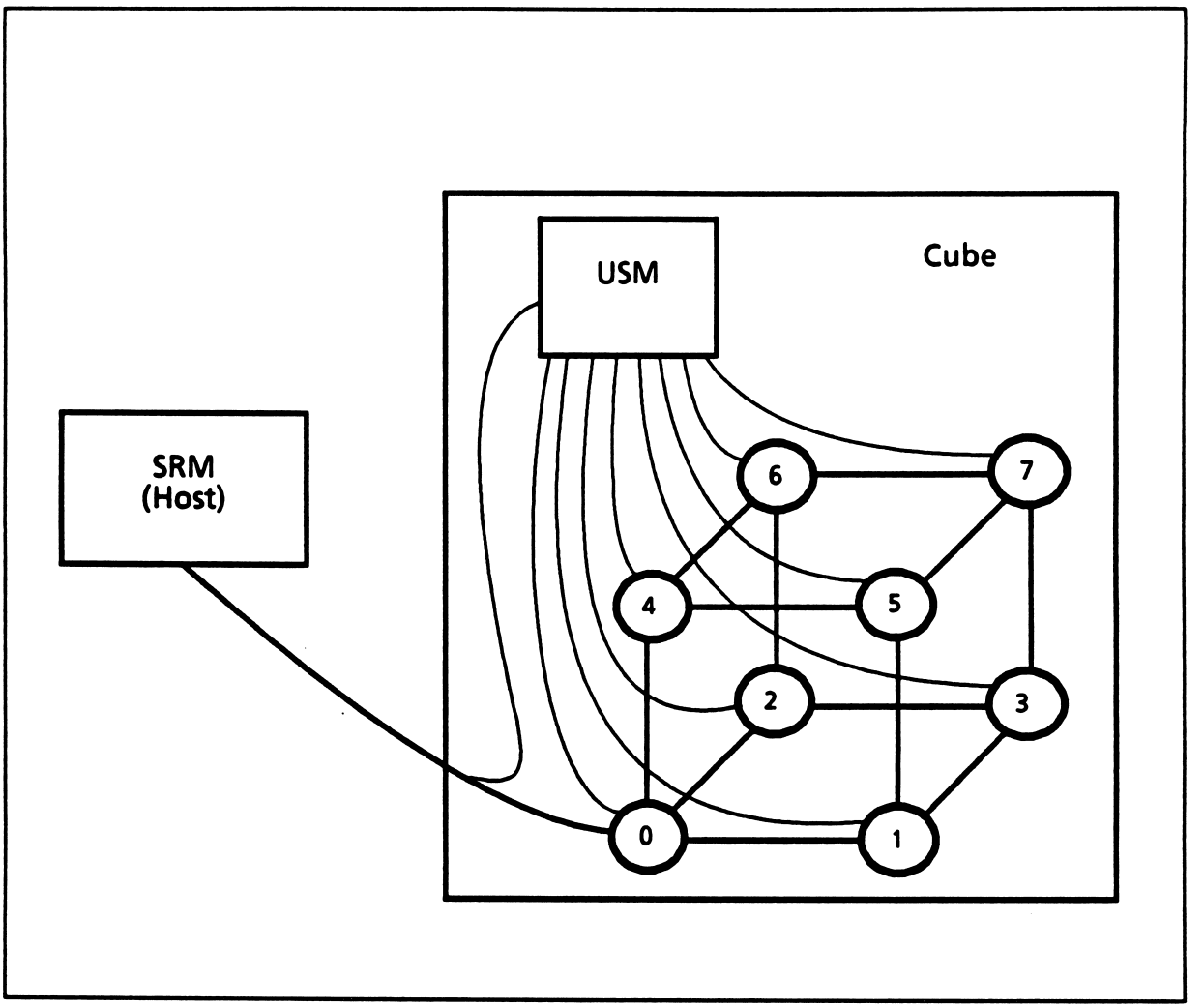


Figure 1-1. iPSC®/2 Diagnostic Hardware Model

The model shows the SRM at the left and the cube at the right. The interconnecting cable carries both the RS-422 signals and the Direct Connect Module (DCM) signals between the host and cube. Internally, the cable is connected to a communications board (not shown), where the RS-422 signals and DCM signals are split. The RS-422 signals (called the Diagnostic Channel) are applied to the USM. The DCM signals (called the Host Channel) are applied to node 0.

Thus, the USM provides a global communications path to each of the nodes for the Diagnostic Channel. The Host channel, on the other hand, fans out from node 0 through its DCM into a hypercube topology. Each of the nodes contains a DCM. The DCM routes the Host Channel messages through the hypercube.

Theory of Diagnostics Versus Tests

A diagnostic utilizes the results of various tests to aid in the actual isolation of a fault; a test determines if a fault occurs. In other words, a diagnostic helps to isolate a failure to the defective part; whereas a test merely fails upon encountering a fault. The iPSC/2 diagnostics utilize an inward/out diagnostic testing strategy.

The inward/out strategy reduces the initial test area to a hard core -- that is, it finds the smallest possible unit with which to begin testing. Next, the test procedure is systematically expanded to cover each logically connecting unit. This is done in such a way that, with each test or iteration, the number of possible fault points is kept to an absolute minimum (thereby increasing the chances of correctly isolating the fault). Finally, when faults do occur, the program cues the user with an error message sufficient to identify the source of the fault.

The diagnostic programs referred to in this manual are designed to isolate faults to Field Replaceable Units (FRUs) such as node boards. However, the NCT also provides advanced features for component-level isolation on the node board. Refer to Appendix A, "Power Up Tests", and Appendix B, "NCT Special Features" for more detailed information on the NCT.

Diagnostic Specification

The performance of the iPSC/2 diagnostics is listed in Table 1-1.

Table 1-1. iPSC®/2 Diagnostic Specifications

	POST	NCT	CDP
Max. Run Time	20 sec.	7 sec.	30 min.
Fault Detection	70%	70%	95%
Fault Isolation to FRU Level	85%	99%	90%
Fault Isolation to Component Level	60%	60%	N/A

OPERATION 2

INTRODUCTION

This chapter describes the procedures for operating the iPSC/2 Diagnostics. It describes the operation of the Power Up Tests, the standard operation of the Cube Diagnostic Program (CDP), and the CDP options.

POWER UP TESTS

There are three separate power up tests in the iPSC/2 system. These are:

1. Power On Self Test (POST) for the SRM,
2. USM Confidence Test for the USM, and
3. Node Confidence Test (NCT) for each of the node boards.

Power On Self Test

The Power On Self Test (POST) automatically runs when the System Resource Manager (SRM) power is applied. The pass or fail indication from POST is displayed on the SRM console. Additional details about the POST are contained in the *iSBC® 386 AT User's Guide*.

USM Confidence Test

The USM Confidence Test runs when the cube power is applied. The pass or fail indication from the USM confidence test is displayed on the USM from panel LEDs. A pass is indicated when both USM LEDs are off. A failure is indicated when the red LED is on and the green LED is blinking. Additional details about the USM Confidence Test are provided in Appendix A, "Power Up Tests".

Node Confidence Test

The Node Confidence Test (NCT) runs when the cube power is applied. It is also caused to run by CDP when it forces the node boards to reset. The pass or fail indication from the NCT is displayed on the node front panel LEDs. A pass is indicated when the node board green LED is on and the red LED is off. Any combination other than green on and red off indicates a fault on the node board. When CDP runs the NCT, it automatically collects status information from the nodes and reports the node pass or fail status. Additional details about the NCTs are provided in Appendix A, "Power Up Tests", and Appendix B, "NCT Special Features".

CUBE DIAGNOSTIC PROGRAM

Power on testing verifies the hardware integrity of the SRM and the cube node boards in a standalone environment. To check the communication links between the nodes and the SRM, use the Cube Diagnostic Program (CDP). Additional details about CDP are provided in Chapter 3, "Cube Diagnostic Program".

Running CDP

To prepare for running CDP, perform the following steps:

1. To invoke CDP you must be the superuser and be in the */usr/ipsc/diag* directory. To get to that directory, type:

   ```
   cd /usr/ipsc/diag
   ```

2. Make certain that no other users are on the system and that all cube file server and cube communications server activity is stopped. You can stop the file server and communications server by typing:

   ```
   bootcube -D1-2
   ```

   You can determine if other users are on the system by typing:

   ```
   who
   ```

3. To invoke CDP, type:

   ```
   ./cdp
   ```

The Main Menu is displayed as follows:

```
Cube Diagnostic Program V2.0
Copyright Intel Corporation, 1985,1986,1987,1988,1989

CDP [V2.0] Main Menu
 1.   Exit to UNIX
 2.   Options Menu
 3.   Diagnostic Link Tests
 4.   Node Standalone Tests
 5.   Host Link Tests
 6.   I/O Link Tests
 7.   Node Link Tests
 8.   Cube Link Tests
 9.   Generic Link Tests
 10.   Optional-Hardware Tests
 11.   Run Standard Tests

      Enter Selection # [2] :
```

and you are prompted to enter the test that you wish to execute.

4. Enter the desired test. If you select Run Standard Tests, all tests except Extended tests will run using default values.

CDP Options

CDP has several optional features that are available to the user. These features are accessed via the CDP Options Menu, which is selected from the CDP Main Menu by item # 2. The Options Menu display is as follows:

```
Options Menu
   1.  Return to Main Menu
   2.  Configuration
   3.  Error Log
   4.  Test Summary
   5.  Enter Shell
   6.  Ignore/Recognize Tests

       Enter Selection # [1] : __
```

The following paragraphs describe each entry in the Options Menu.

RETURN TO MAIN MENU

This item returns to the CDP Main Menu, as shown previously.

CONFIGURATION

There are three configuration functions. The Display Configuration function lists the current settings of each configurable option. The Modify Configuration function permits changes to the configurable options. The third configuration function reads the */usr/ipsc/conf/cubeconf* file and resets the options accordingly.

Table 2-1 explains the configuration options. The contents of the */usr/ipsc/conf/cubeconf* file can be altered to make permanent changes to the default settings of each configuration option.

CDP reads the */usr/ipsc/conf/cubeconf* file each time it is invoked. This allows users to change the configuration options in the *cubeconf* file without having to run the CDP Options Menu. For example, the following entry in the *cubeconf* file sets the default value of the CDP stop on error flag so that it will continue on error instead of stopping after the first error. (As shown in the table, CDP ordinarily stops after the first error.)

```
cdp_halt        0
```

For more detailed information, refer to the *iPSC®/2 System Administrator's Guide.*

ERROR LOG

NOTE

Error messages displayed on the console are also sent to an error log file named *cdp_errlog*. The error message filed is always at the verbose message level.

The error log option displays the following selections:

```
Pick Error Log Function:
   1.  Return to Options Menu
   2.  Display Log
   3.  Clear Log
   4.  Turn Log Off

      Enter Selection # [1] :
```

These selections permit displaying, clearing, or turning off the error logging.

Table 2-1. CDP Configuration Options

KEYWORD	DESCRIPTION	DEFAULT
cdp_mode	Controls the mode of operation: 1 = Menu Mode, 0 = Batch Mode.	1
cdp_emsg	Controls the displayed Error Message detail: 0 = silent, 1 = terse, 2 = normal, 3 = verbose.	3
cdp_halt	Controls the Stop On Error flag: 0 = continue on error, 1 = stop after first error.	1
slots	Controls the number of compute node slots under test. Valid values are 1, 2, 4, 8, 16, 32, 64, and 128.	32
reset_delay	Controls the maximum time (seconds) allowed for the Node Confidence Tests to complete.	7
test_node	Controls the node under test. A Node ID outside the range of 0 to 254 tests all nodes.	-1
serial_timeout	Controls the maximum time (seconds) allowed for a Diagnostic Channel read or write.	7
cube_timeout	Controls the maximum time (seconds) allowed for a DCM Channel message receive or send.	6
baud	Controls the Diagnostic Channel Baud rate.	38400
backplane_rev	Indicates the Cube Unit backplane revision.	B
host_rev	Indicates the Host DCM revision.	G
cdp_smsg	Controls the Test Status Message detail: 0 = silent, 1 = terse, 2 = normal, 3 = verbose.	3
dcm_reset	Controls DCM resets after an error: 1 = resets DCMs after an error, 0 = does not.	1
dcm_msg_len	Controls the DCM test message sizes. Valid range is 4 to 16384 at each 4 byte boundary.	244
dcm_repeat	Controls how many DCM messages are transmitted before any is received.	5
slot keyword	Controls the contents of a Cube Unit cardcage slot.	NODE4G7
USM keyword	Controls the contents of a card cage USM slot.	A00B.32

TEST SUMMARY

The test summary option displays the following menu:

Test Summary Menu

```
1.  Exit to Options Menu
2.  Change executed tests in sumnmary [executed only]
3.  Change pass/fail tests in summary [failed only]
4.  Perform the Summary

   Enter Selection # [4] :
```

If you choose 2, three selections are displayed to allow the test summary to include:

```
Tests that executed only
Tests that executed or not
Tests that have not executed
```

If you choose 3, three selections are displayed to allow the test summary to include:

```
Tests that failed only
Tests that passed or failed
Tests that passed only
```

If you choose 4, a test summary is displayed per the parameters specified in options 2 and 3. The summary is displayed as shown in the following example:

```
Tests shown in this summary have executed only and failed only

Menu Name / Test Name          Trials      Errors      Failures
---------------------          ---------   --------    --------
    Node Standalone
        Compute Node DCM Loopback    10          1           1

Press <Return> to continue.
```

ENTER SHELL

The enter shell option allows you to enter a UNIX shell. After you exit the shell, the Options Menu will return.

IGNORE/RECOGNIZE TESTS

The ignore/recognize tests option displays all of the CDP tests. You may then select any test and toggle the test to be ignored or recognized. Remember, ignored tests are termed "Extended Tests" by CDP.

CUBE DIAGNOSTIC PROGRAM 3

INTRODUCTION

This chapter describes the Cube Diagnostic Program (CDP). First, it describes the overall Menu structure. Second, it describes each Test Menu as it pertains to the major parts of the system hardware under test. Third, it describes each of the standard tests in detail, including the detailed hardware under test. Fourth, it describes the extended tests. Fifth, it describes the optional-hardware tests. Finally, it describes the CDP method of handling the hardware state as it pertains to running tests.

The CDP tests check the full functionality of the iPSC/2 system hardware. The tests also provide a high degree of flexibility for focusing on a specific subsystem or interface link. The actual system hardware under test is determined from the */usr/ipsc/conf/cubeconf* file. See Chapter 2 for details.

MENU STRUCTURE

CDP contains a Main Menu, an Options Menu, and several Test Menus. The structure of the menus is shown in Figure 3-1. The Main Menu is the first menu that signs on after CDP is invoked from UNIX. From the Main Menu, you can select the Options Menu, select a Test Menu, or Run Standard Tests. It is also possible to travel from the Options Menu to a UNIX Shell. And, it is also possible to travel directly from a Test Menu to the Options Menu.

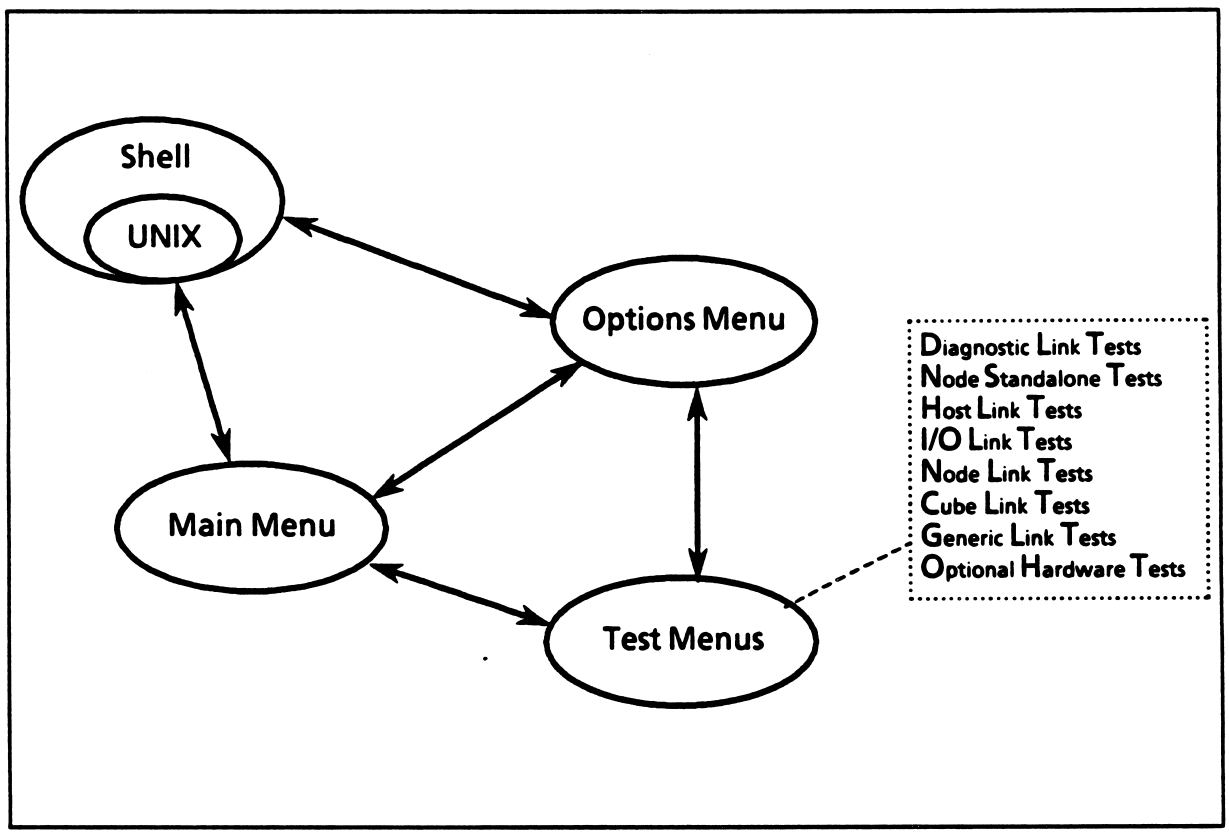


Figure 3-1. CDP Menu Structure

The Options Menu controls the operation of CDP. Refer to Chapter 2 for details.

TEST MENU STRUCTURE

Each one of the Test Menus is structured as follows:

```
<menu name> Tests Menu
 1.  Return to <previous menu name> Menu
 2.  Options Menu
 3.  <first test name> Test
 4.  <next test name> Test
      .
      .
 m.  <last test name> Test
 n.  Run Standard Tests

   Enter Selection # [1] : _
```

The first menu component, *menu name*, is the name of the menu (for example, Diagnostic Link Tests Menu). The first entry:

```
1.  Return to <previous menu name> Menu
```

allows you to proceed backward in the selection tree. The last entry:

```
n.  Run Standard Tests
```

runs all standard tests listed in the menu. A standard test is a test not defined as Extended. Extended Tests are identified with `[extended test]` beside the names, have special requirements such as loopback connectors, and are used primarily by factory trained Manufacturing and Engineering personnel. The last line:

```
Enter Selection # [1] : _
```

prompts for a number. If you press <Return> without entering a number, the default answer listed between the brackets `[1]` is assumed. If you enter a number listed beside a test name, the test runs.

DIAGNOSTIC LINK TESTS MENU

The Diagnostic Link Tests (DLT) Menu contains tests that check the RS-422 diagnostic channel between the SRM and each Node through the USM (see Figure 3-2). These tests ensure that the system is capable of loading more complex tests and the node operating system (NX).

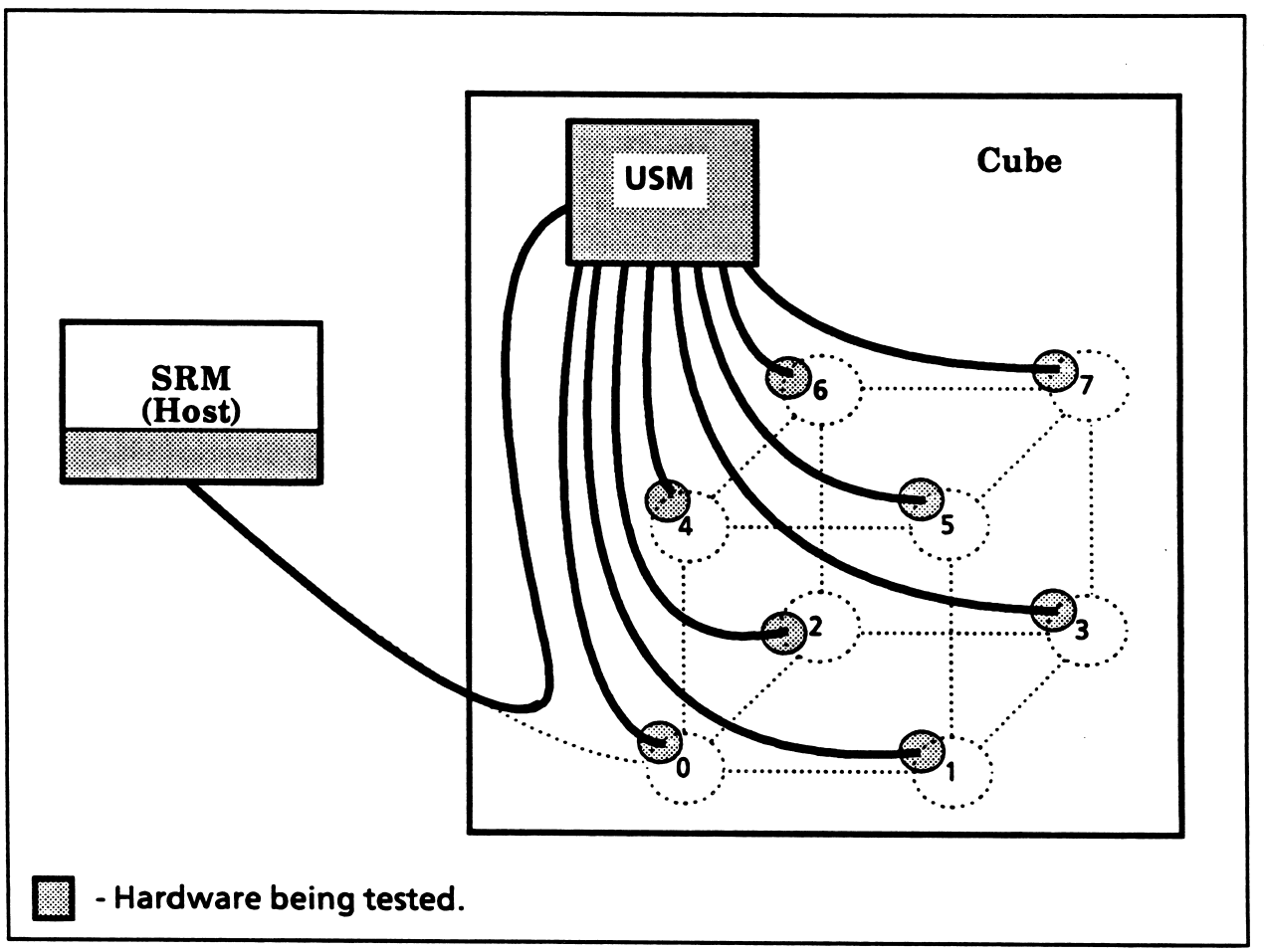


Figure 3-2. iPSC®/2 Hardware Tested By The Diagnostic Link Tests

NODE STANDALONE TESTS MENU

The Node Standalone Tests (NST) Menu contains tests that check the circuitry on each node board that will operate in a standalone environment (see Figure 3-3).

This circuitry is independent of node-to-node communications.

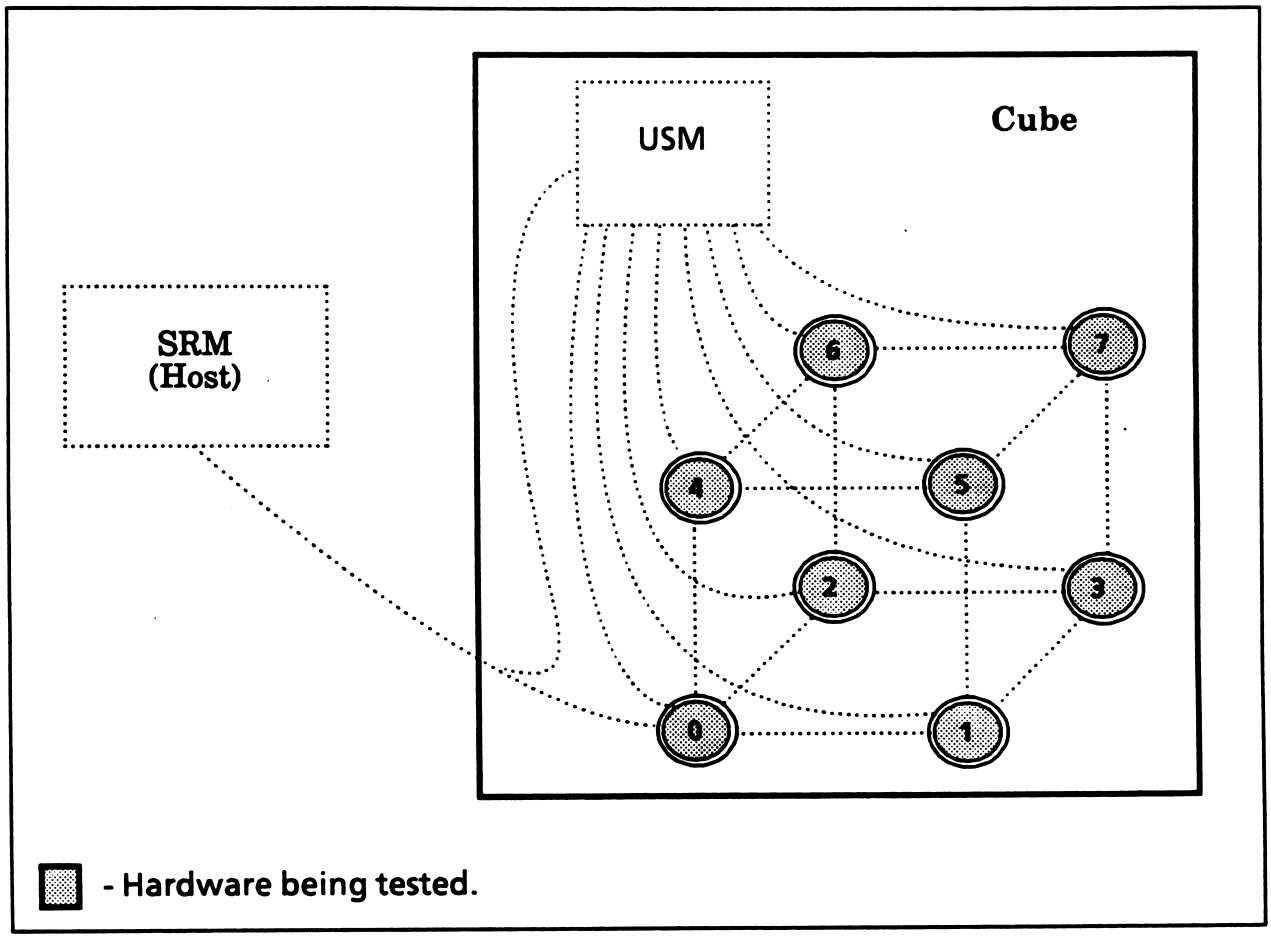


Figure 3-3. Hardware Tested By The Node Standalone Tests

The Node Standalone Tests consist of tests that are firmware resident and tests that are RAM resident. The firmware resident tests are called the Node Confidence Tests (see Appendix A). The RAM resident tests are packaged with a diagnostic monitor that is downloaded as needed. Additional details of each of the RAM resident tests are provided later in this chapter. There is also a section later in this chapter entitled "Hardware Test States" that directly relates to the manner in which the Node Standalone tests operate.

HOST LINK TESTS MENU

The Host Link Tests (HLT) Menu contains tests that check the host channel between the SRM and node board 0 (see Figure 3-4). If the SRM cannot communicate reliably with node 0, it will probably not communicate dependably with any node. This channel handles all of the communications for every application program that runs on the iPSC/2 system.

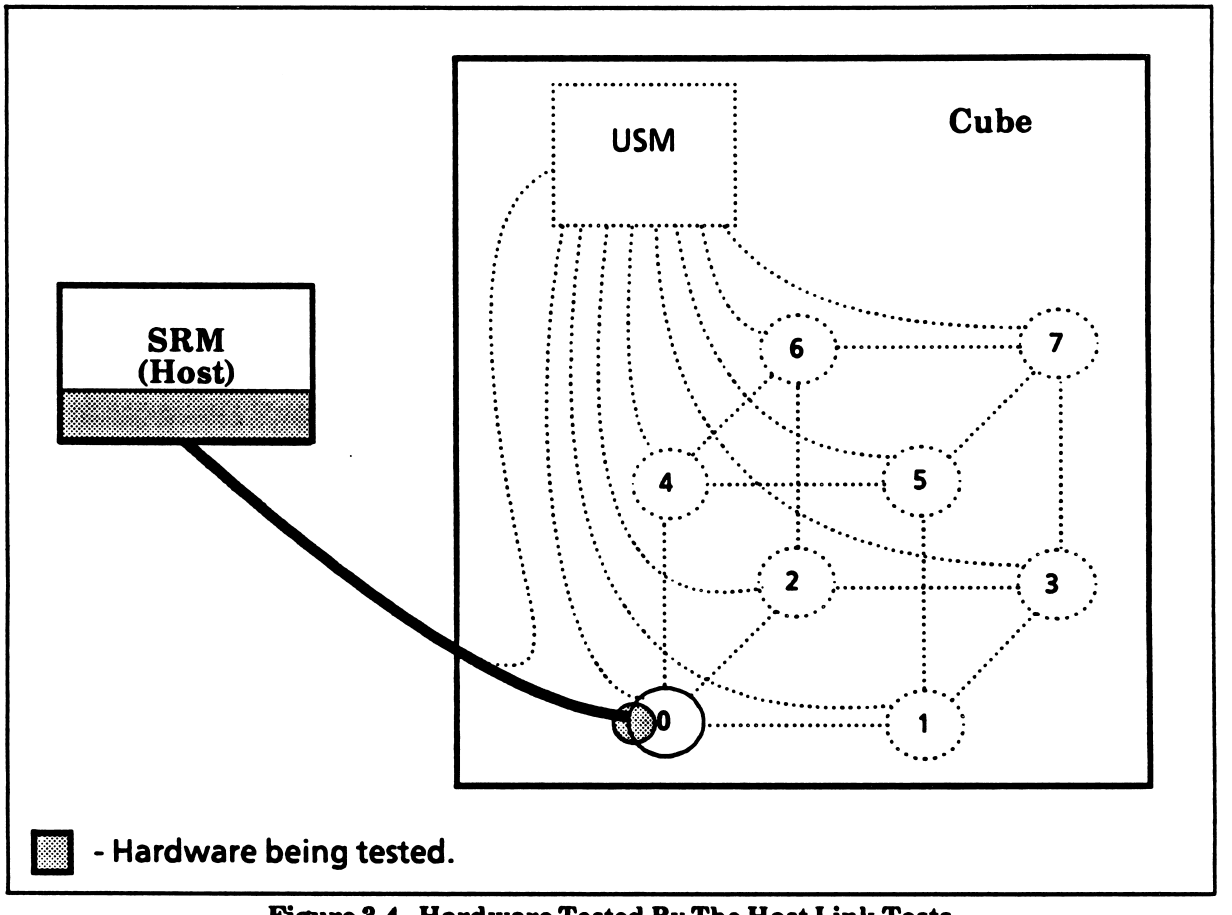


Figure 3-4. Hardware Tested By The Host Link Tests

I/O LINK TESTS MENU

The I/O Link Tests (IOLT) Menu contains tests that check the DCM channel between each compute node and I/O node (see Figure 3-5).

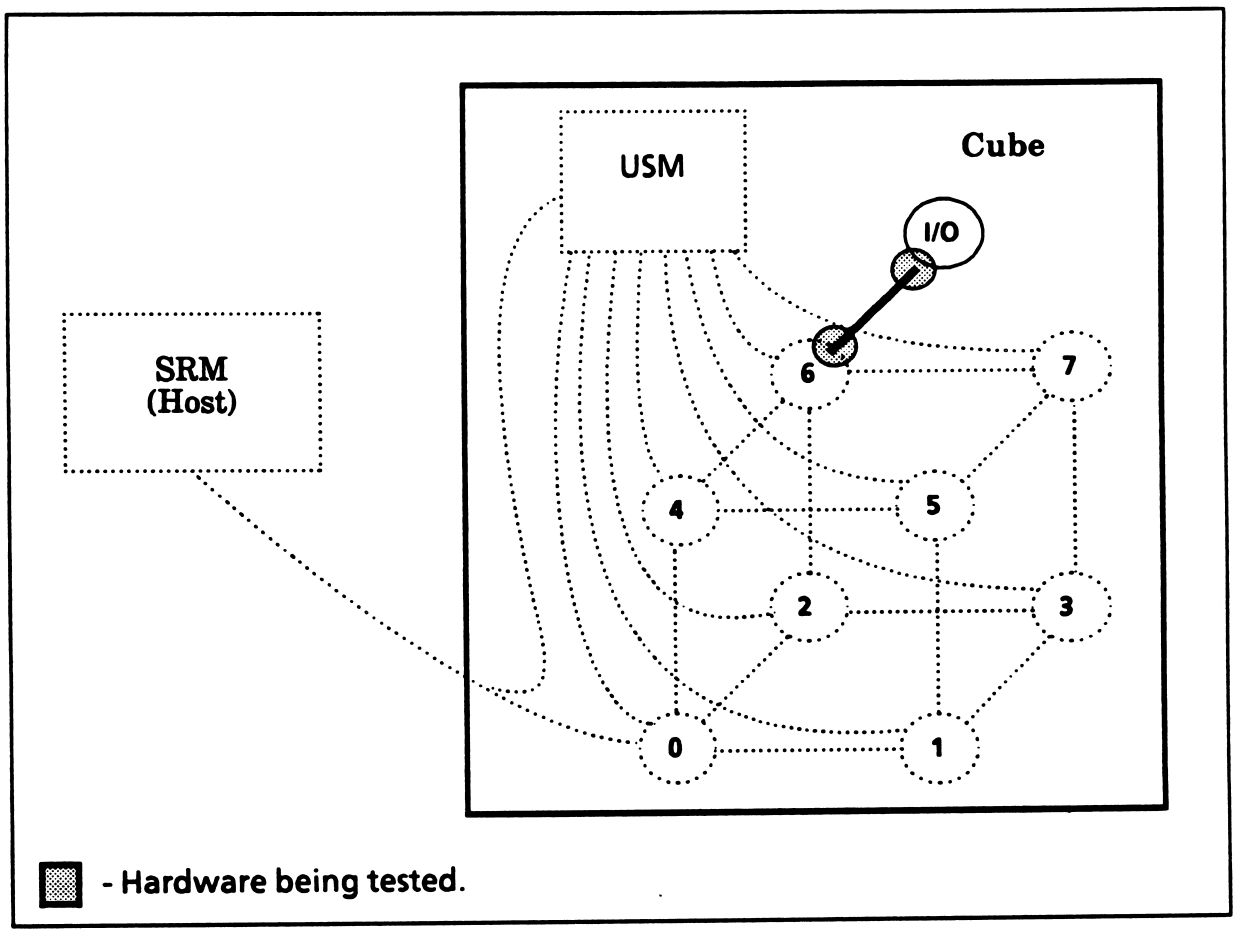


Figure 3-5. Hardware Tested By The I/O Link Tests

NODE LINK TESTS MENU

The Node Link Tests (NLT) Menu contains tests that check the node-to-node communications (see Figure 3-6). These tests are completely independent of the Host Channel, which eliminates the host as a suspect.

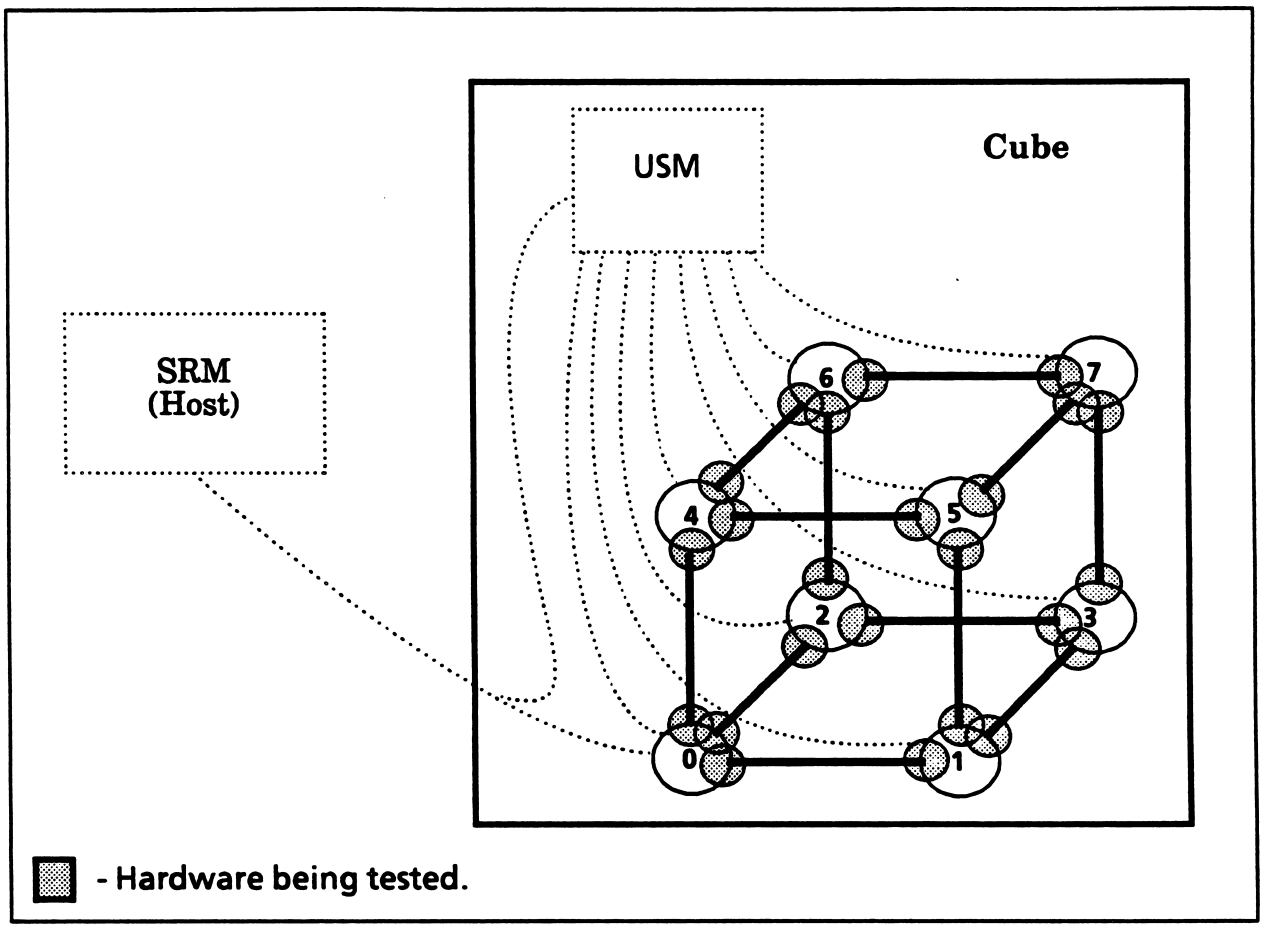


Figure 3-6. Hardware Tested By The Node Link Tests

CUBE LINK TESTS MENU

The Cube Link Tests (CLT) Menu contains tests that check the host-to-node 0 communications and node-to-node communications (see Figure 3-7). The CLTs test the system in a worst case environment with a maximum of communications traffic. An overnight run of the CLTs is recommended for intermittent communication problems. It is also recommended that, for more persistent intermittent problems, the *dcm_msg_len* parameter be increased to the maximum size via the Options Menu.

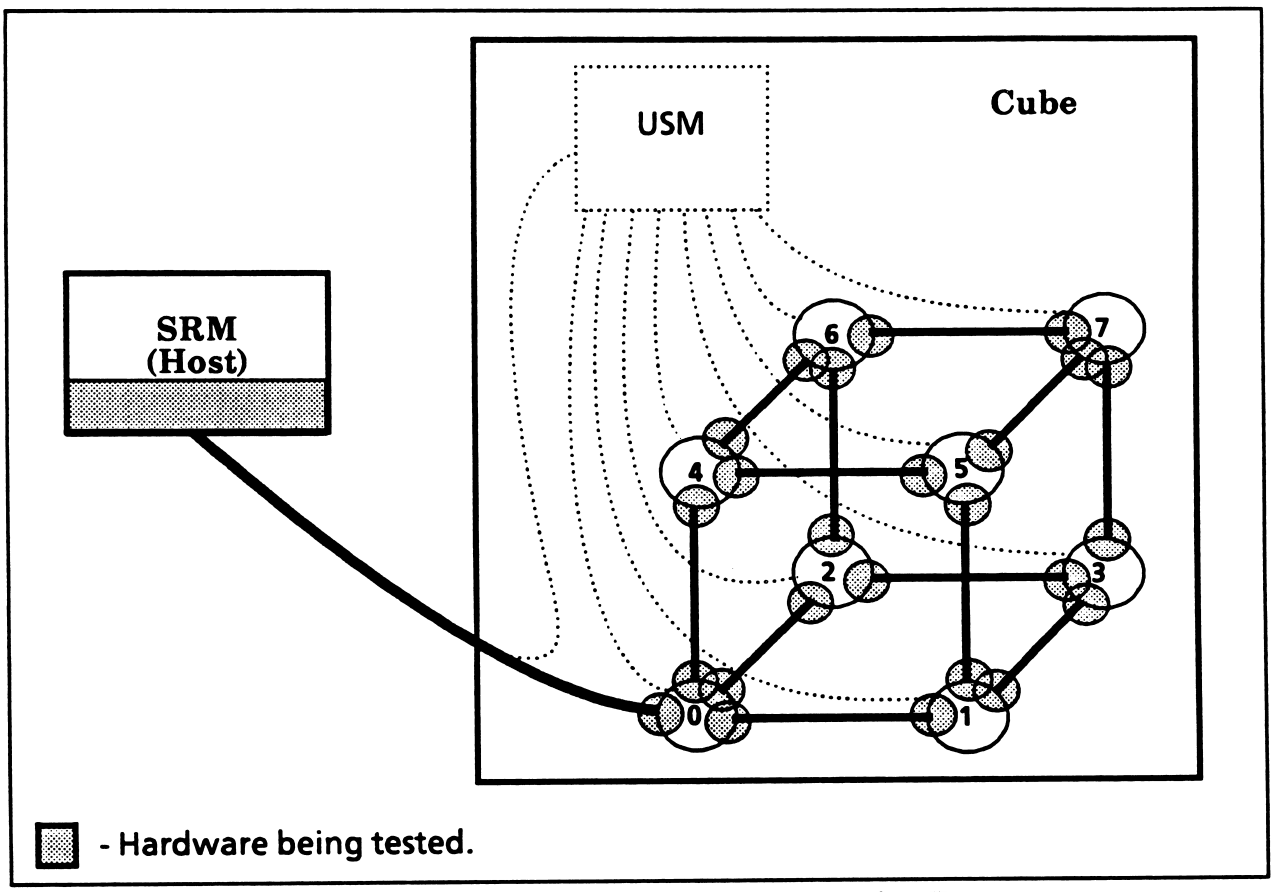


Figure 3-7. Hardware Tested By The Cube Link Tests

GENERIC LINK TESTS MENU

The Generic Link Tests (GLT) Menu contains tests that allow you to select the links under test. For example, one of the GLTs will allow you to select a specific link, such as between node 2 and node 1, and test only that link. The GLTs are ordinarily used by iSC Engineering personnel to check the quality of DCM signals after a hardware modification.

DIAGNOSTIC LINK TESTS DETAILED DESCRIPTION

The Diagnostic Link Tests (DLTs) are listed in Table 3-1.

Table 3-1. Diagnostic Link Tests

TEST NAME
SRM UART Loopback Test
Cable & USM Backplane Loopback Test [Extended]
USM Echo Test
USM & Node Backplane Loopback Test [Extended]
Node Echo Test

The DLTs write and read the pattern of bytes shown in Table 3-2 (and the complements of those bytes) using the Diagnostic Link. (Patterns are shown in Table 3-2 in left-to-right top-to-bottom order.) An attachment is made to the Diagnostic Link via the UNIX asynchronous driver with device */dev/tty01*. Each read operation sets a timer for *serial_timeout* seconds and, if a character is not received before the timer expires, the channel is presumed to be hung and a timeout error message is generated.

Table 3-2. DLT Test Patterns

00	01	02	03	04	05	06	07
08	09	0A	0B	0C	0D	0E	10
20	30	40	50	60	70	80	90
A0	B0	C0	D0	E0	5A	C7	53
7C	88	44	22	11			

NOTE

The *serial_timeout* parameter can be modified from the Options Menu as shown in Chapter 2.

The two extended DLTs are described later in this chapter. The following paragraphs describe each of the standard DLTs.

Diag. Link: SRM UART Loopback Test

The hardware tested by the Diag. Link: SRM UART Loopback Test is shown in Figure 3-8.

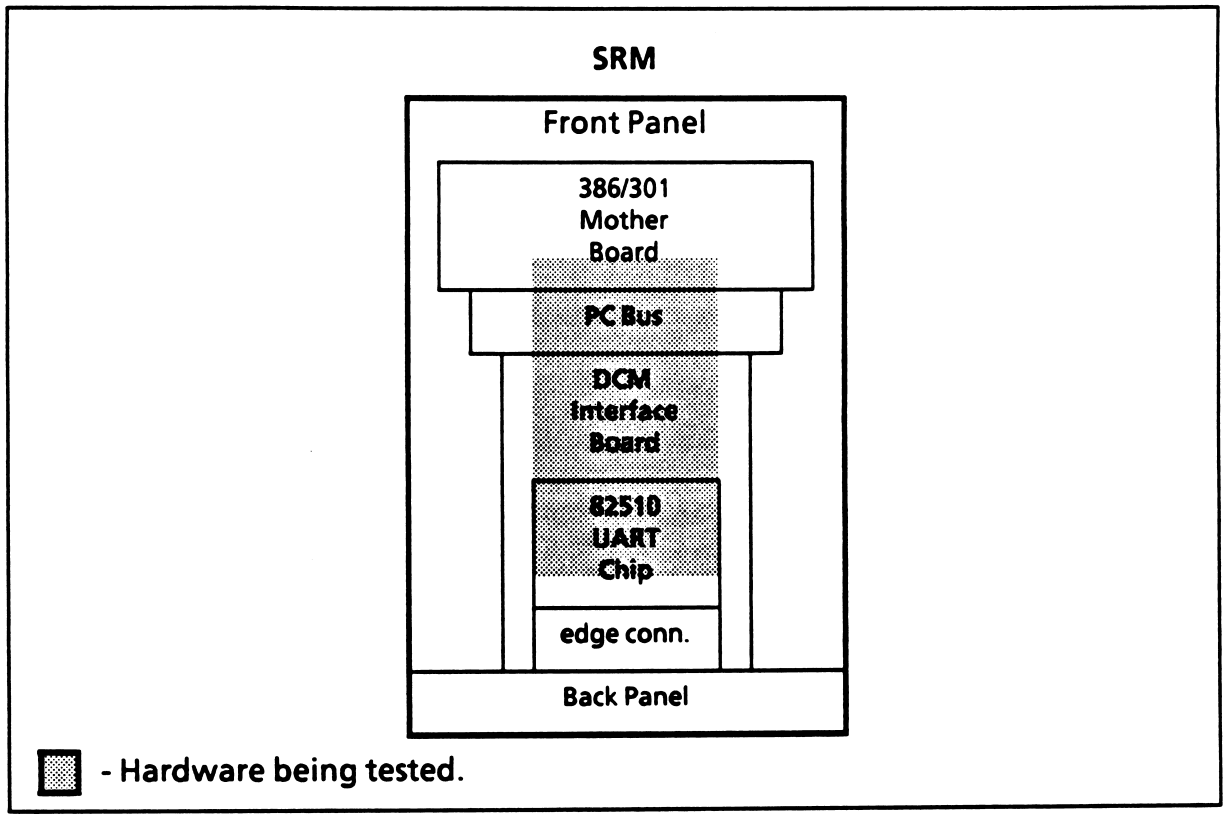


Figure 3-8. Hardware Tested By The Diag Link: SRM UART Loopback Test

The Diag. Link: SRM UART Loopback Test performs as follows:

1. The UART (82510 chip) is set into an internal loopback mode so that it buffers one character at a time for write operations internally and forwards the buffered character back during a read operation. This mode checks as much of the interface hardware as possible without requiring that the SRM and Cube be physically connected. The internal loopback mode is set by outputting directly to the UART mode registers. This practice is **not** recommend for normal applications.

2. Writes a test character, reads it back, compares the character read with the character written, and generates an error message if there is a difference.

3. Repeats the above step for each test pattern and its complement.

4. Removes the UART from the internal loopback mode.

Diag. Link: USM Echo Test

The hardware tested by the Diag. Link: USM Echo Test is shown in Figure 3-9.

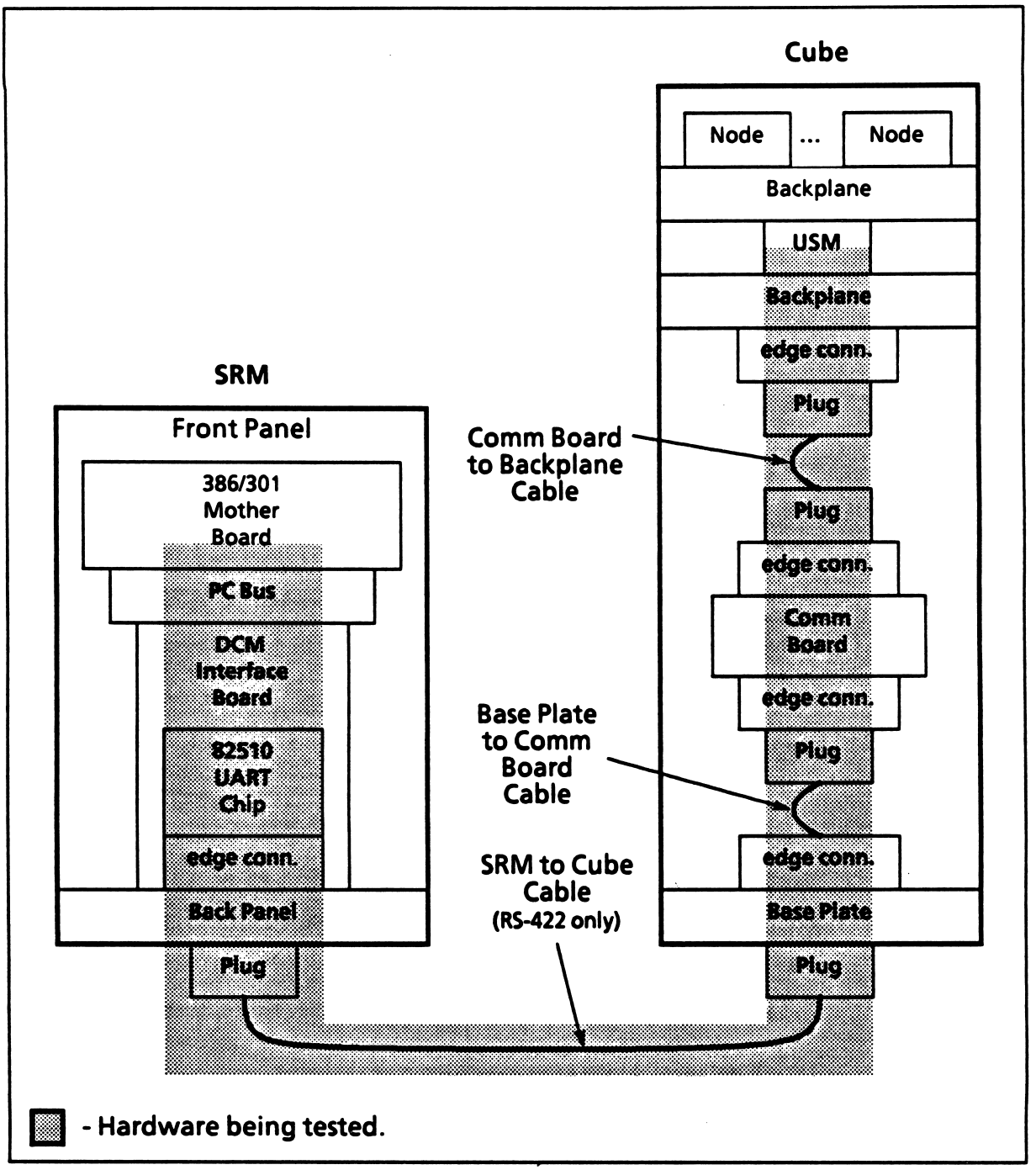


Figure 3-9. Hardware Tested By The Diag Link: USM Echo Test

The Diag. Link: USM Echo Test performs as follows:

1. The USM is placed into the Echo Mode (See Appendix D for details) so that it buffers one character at a time for write operations internally and forwards the buffered character back during a read operation.
2. Writes a test character, reads it back, compares the character read with the character written, and generates an error message if there is a difference.
3. Repeats the above step for each test pattern and its complement.
4. Removes the USM from the Echo Mode.

Diag. Link: Node Echo Test

The hardware tested by the Diag. Link: Node Echo Test is shown in Figure 3-10.

The Diag. Link: Node Echo Test performs as follows:

1. The USM is placed into the Pipe Mode (See Appendix D for details) so that it forwards all characters back and forth between the SRM and the nodes.
2. The first node is placed into the Echo Mode (See Appendix C for details) so that it buffers one character at a time for write operations internally and forwards the buffered character back during a read operation.
3. Writes a test character, reads it back, compares the character read with the character written, and generates an error message if there is a difference.
4. Removes the node from Echo Mode.
5. Repeats the above steps for each test pattern and its complement.
6. Repeats above steps for each other node.
7. Removes the USM from the Pipe Mode.

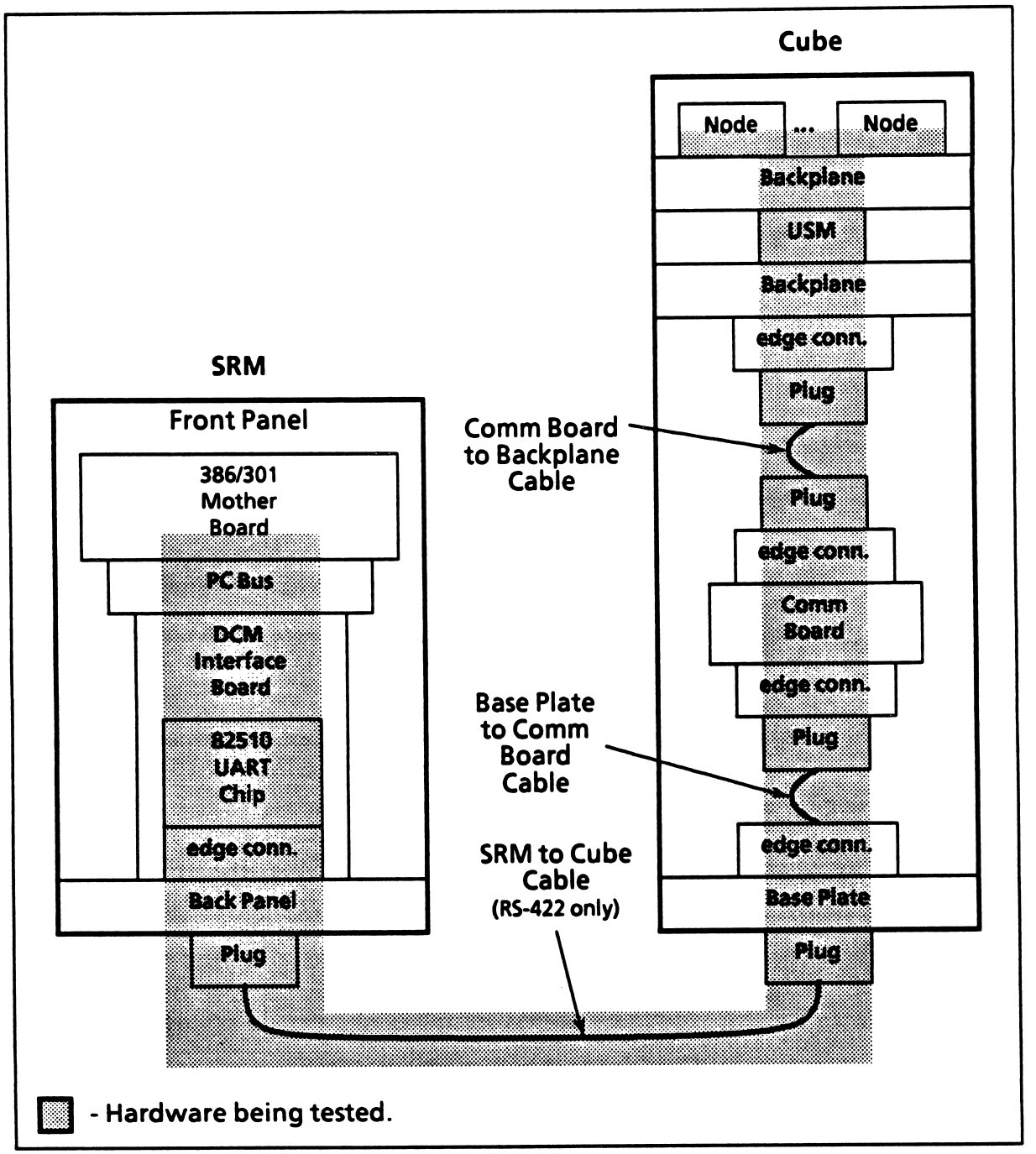


Figure 3-10. Hardware Tested By The Diag Link: Node Echo Test

NODE STANDALONE TESTS DETAILED DESCRIPTION

The Node Standalone Tests (NSTs) are listed in Table 3-3.

Table 3-3. Node Standalone Tests

TEST NAME
Node Confidence Test
RAM Size Test
RAM Data Lines Test
RAM Address Lines Test
Compute Node DCM Loopback Test
Compute Node DCM Checksum Test

Node Standalone: Node Confidence Test

The Node Confidence Test (NCT) is a series of tests that reside in the firmware of each node board. The NCTs are also commonly referred to as the node power-up tests. CDP executes the NCTs by commanding the USM to reset each of the node boards. As soon as each NCT completes, it reports test completion back to the USM. CDP polls the USM for *reset_delay* seconds and generates a timeout message if an NCT does not complete. After completion, CDP collects the NCT status from each node board. See Appendices A and B for additional details about the NCTs.

NOTE

All of the Node Standalone Tests are downloaded with a Diagnostic Monitor except the NCTs. See the section near the end of this chapter entitled "Hardware Test States" for further details.

Node Standalone: RAM Size Test

The Node Standalone: RAM Size Test verifies the accuracy of the system memory configuration. For example, it may indicate that node 5 has a 1-megabyte RAM module, but the configuration as specified in the *cubeconf* file indicates 4 megabytes. This verifies the configuration and helps to ensure the node RAM configuration. Without such a test, it would be possible to replace a node inadvertently with a spare node that has a different size RAM module.

Node Standalone: RAM Data Lines Test

The Node Standalone: RAM Data Lines Test writes, reads and verifies the 32 data bits of RAM. Four patterns are used to check the integrity of the data bus. The patterns are: all zeros, walking one, walking zero, and all ones. The walking one pattern sets a single bit at a time and walks the one across each of the 32 bits in the data bus, one bit at a time. The walking zero pattern uses the same algorithm.

Node Standalone: RAM Address Lines Test

The Node Standalone: RAM Address Lines Test writes, reads, and verifies each of the individual 32 bit locations in RAM above the first megabyte with a unique data pattern. This is accomplished as follows:

1. The 32-bit hexadecimal test pattern of A5A5A5A5 is written to all of the RAM locations above the first megabyte.
2. The first location is then verified, complemented, and re-verified.
3. The step above repeats for each individual RAM location above the first megabyte.

NOTE

The NCTs check the first megabyte of RAM and the Node Standalone: RAM Address Lines Tests check all of the RAM above the first megabyte.

Node Standalone: Compute Node DCM Loopback Test

The Node Standalone: DCM Loopback Test writes and reads a message that is composed of the pattern of dwords (32 bits) shown in Table 3-5. The message is sent to the node DCM using a null routing probe. This forces the DCM to loop the message back during a read operation. The message is then read and compared with what was written, and an error message is generated if there was a difference. This checks as much of the DCM interface hardware as possible without requiring that nodes and host-node communications be physically connected.

Node Standalone: Compute Node DCM Checksum Test

The Node Standalone: DCM Checksum Test writes and reads a message composed of the pattern of dwords (32 bits) shown in Table 3-5. The message is sent to the node DCM with a corrupt checksum using a null routing probe. This forces the DCM to loop the message back during a read operation. The message is then read and the test expects a checksum to be detected. If a checksum is not detected, an error message is generated. This checks the node DCM checksum detection circuits without requiring that nodes and host-node communications be physically connected.

HOST LINK TESTS DETAILED DESCRIPTION

The Host Link Tests (HLTs) are listed in Table 3-4. The HLTs write and read DCM messages between the host and node board 0 via the Host Link. The messages are composed of the pattern of dwords (32 bits) shown in Table 3-5. (The pattern is shown in left-to-right and top-to-bottom order in Table 3-5.)

Table 3-4. Host and I/O Link Tests

Test Name
SRM FIFO Loopback Test
Cable & Node Backplane Loopback Test [Extended]
Host/Node 0 Channel 7 Router Test
Large Message Test
Node 0 Receive Test
Node 0 Transmit Test
DCM Checksum Test

Table 3-5. HLT Test Patterns (in hexadecimal)

00000000	00000000	FFFFFFFF	FFFFFFFF	A5A5A5A5	A5A5A5A5
5A5A5A5A	5A5A5A5A	00000000	F0F0F0F0	0F0F0F0F	F0F0F0F0
0F0F0F0F	FFFFFFFF	00000000	01010101	DBDBDBDB	00000000
FFFFFFFF	0000FFFF	00FF00FF	0F0F0F0F	33333333	55555555
AAAAAAAA	CCCCCCCC	F0F0F0F0	FF00FF00	FFFF0000	00000001
00000010	00000100	00001000	00010000	00100000	01000000
10000000					

The data patterns are copied into the message buffer as many times as necessary to fill the buffer. An attachment is made to the Host Link via the UNIX DCM driver with */dev/cube*. Each read operation sets a watchdog timer for *cube_timeout* seconds and, if a message is not received before the timer expires, the channel is presumed to be hung and a timeout message is generated.

NOTE

Chapter 2 describes how the *dcm_msg_len* parameter can be modified to vary the size of the DCM message buffer. The *cube_timeout* parameter is also modifiable.

The extended HLT is described later in this chapter. The following paragraphs describe each of the standard HLTs.

Host Link: SRM FIFO Loopback Test

Figure 3-11 shows the hardware tested by the Host Link: SRM FIFO Loopback Test.

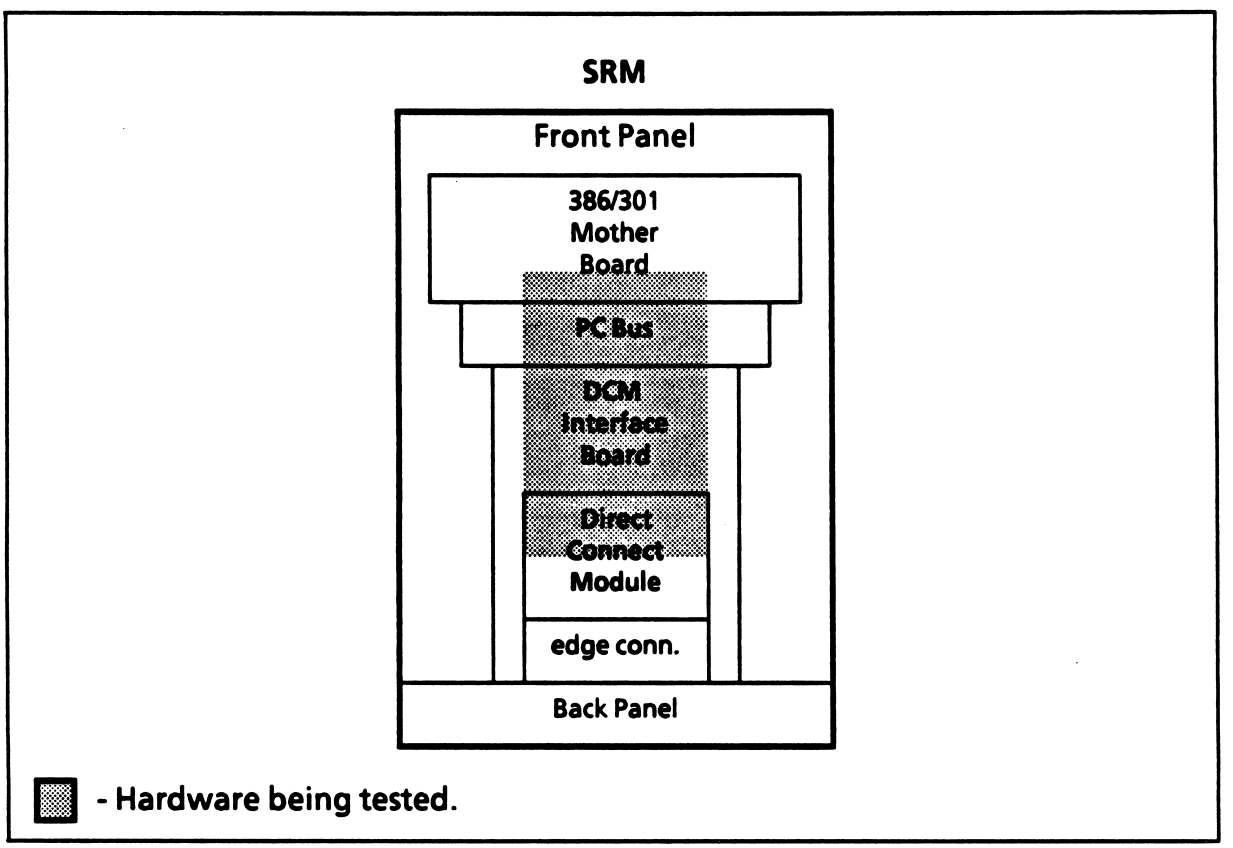


Figure 3-11. Hardware Tested By The Host Link: SRM FIFO Loopback Test

The Host Link: SRM FIFO Loopback Test performs as follows:

1. The host DCM is set into an internal loopback mode so that it buffers a message for write operations internally and forwards the buffered message back during a read operation. This mode checks as much of the DCM interface hardware as possible without requiring that the SRM and Cube be physically connected. The internal loopback mode is set by outputting directly to the DCM mode registers. This practice is **not** recommended for normal applications.

2. Writes a test message, reads it back, compares the character read with the character written, and generates an error message if there was a difference.

3. Removes the host DCM from the internal loopback mode.

Host Link: Host/Node 0 Channel 7 Router Test

The Host sends a DCM message to node 0 with a special routing probe. The probe routes the message directly through the node 0 DCM and the message returns to the Host. The Host then reads and checks the message.

NOTE

The Host Link: Host/Node 0 Channel 7 Router Test checks the interface between the Host and the node 0 DCM. The interface between node 0 and the node 0 DCM is not exercised by this test. However, it is exercised by the Host Link: Node 0 Receive Test and the Host Link: Node 0 Transmit Test.

The hardware tested by the Host Link: Host/Node 0 Channel 7 Router Test is shown in Figure 3-12.

Host Link: Large Message Test

This test is the same as the preceding test except that a test message length equal to the host FIFO size is used.

Host Link: Node 0 Receive Test

The Host sends a DCM message to node 0. Node 0 is then instructed to receive and check the message (see Figure 3-12).

Host Link: Node 0 Transmit Test

Node 0 is instructed to send a message to the Host. The Host then reads and checks the message (see Figure 3-12).

Host Link: DCM Checksum Test

A message containing a corrupt checksum is sent to node 0 with a routing probe that routes it directly back to the Host. The message is read and checked for checksum errors. This test verifies the checksum detection circuit on the Host DCM (see Figure 3-12).

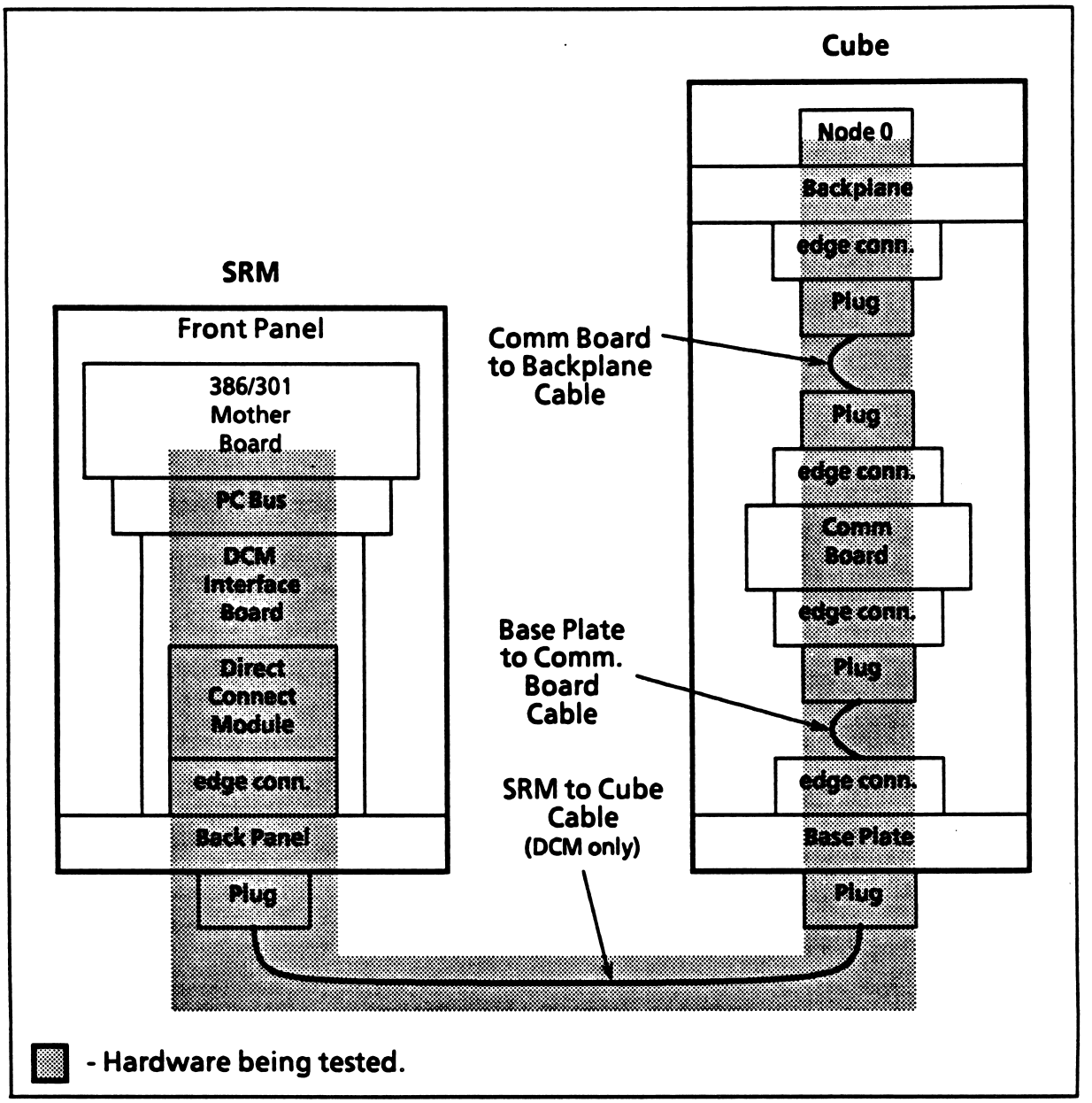


Figure 3-12. Hardware Tested By The Host Link: Host/Node 0 Channel 7 Router, Long Message, Node 0 Receive, Node 0 Transmit, and DCM Checksum Tests

I/O LINK TESTS DETAILED DESCRIPTION

The I/O Link Tests are the same as the Host Link Tests except, of course, that the channel between each compute node and I/O node is checked rather than the host channel. Table 3-6 shows the correspondence between the names of the tests.

Table 3-6. I/O Link Tests/Host Link Tests Name Correspondence

I/O Link Test Name	Host Link Test Name
Cable & I/O Node Loopback Test [Extended]	Cable & Node Backplane Loopback Test [Extended]
I/O Compute Node Channel 7 Router Test	Host/Node 0 Channel 7 Router Test
I/O Node Transmit Test	Node 0 Transmit Test
I/O Node Receive Test	Node 0 Receive Test
I/O Node DCM Checksum Test	DCM Checksum Test

NODE LINK TESTS DETAILED DESCRIPTION

The Node Link Tests (NLTs) are listed in Table 3-7. The NLTs write and read DCM messages between the compute node boards. The messages consist of the same patterns of dwords (32 bits) as the Host Link Tests (see Table 3-5). The Diagnostic Monitor is used via the Diagnostic Link to perform the actual DCM write and read operations as well as checking the messages after those are read.

Table 3-7. Node Link Tests

Test Name
Single Xmit & Recv Test
Concurrent Receive Test
Router Test
Channel Arbitration Test
Concurrent Router Test
Multi-Hop Test
Neighbor Concurrent Comm Test
Node Concurrent Comm Test

The NLTs are similar to the Cube Link Tests. The difference is that the Host is not checked by the NLTs.

NOTE

The NLTs are described with the Cube Link Tests. Remember, the only difference between NLT and CLT is that CLT also checks the Host and the I/O nodes.

Node Link: Single Xmit & Recv Test

The Node Link: Single Xmit & Recv Test individually checks each of the node-to-node links as listed in Table 3-8 in a 0 - 1, 0 - 2, 0 - 4, etc. pattern.

Table 3-8. Node Neighbors (*sheet 1 of 4, nodes 0 - 31*)

Node #	CH 0	CH 1	CH 2	CH 3	CH 4	CH 5	CH 6	CH 7
0	1	2	4	8	16	32	64	host
1	0	3	5	9	17	33	65	-
2	3	0	6	10	18	34	66	-
3	2	1	7	11	19	35	67	-
4	5	6	0	12	20	36	68	-
5	4	7	1	13	21	37	69	-
6	7	4	2	14	22	38	70	-
7	6	5	3	15	23	39	71	-
8	9	10	12	0	24	40	72	-
9	8	11	13	1	25	41	73	-
10	11	8	14	2	26	42	74	-
11	10	9	15	3	27	43	75	-
12	13	14	8	4	28	44	76	-
13	12	15	9	5	29	45	77	-
14	15	12	10	6	30	46	78	-
15	14	13	11	7	31	47	79	-
16	17	18	20	24	0	48	80	-
17	16	19	21	25	1	49	81	-
18	19	16	22	26	2	50	82	-
19	18	17	23	27	3	51	83	-
20	21	22	16	28	4	52	84	-
21	20	23	17	29	5	53	85	-
22	23	20	18	30	6	54	86	-
23	22	21	19	31	7	55	87	-
24	25	26	28	16	8	56	88	-
25	24	27	29	17	9	57	89	-
26	27	24	30	18	10	58	90	-
27	26	25	31	19	11	59	91	-
28	29	30	24	20	12	60	92	-
29	28	31	25	21	13	61	93	-
30	31	28	26	22	14	62	94	-
31	30	29	27	23	15	63	95	-

Table 3-8. Node Neighbors (*sheet 2 of 4, nodes 32 - 63*)

Node #	CH 0	CH 1	CH 2	CH 3	CH 4	CH 5	CH 6	CH 7
32	33	34	36	40	48	0	96	-
33	32	35	37	41	49	1	97	-
34	35	32	38	42	50	2	98	-
35	34	33	39	43	51	3	99	-
36	37	38	32	44	52	4	100	-
37	36	39	33	45	53	5	101	-
38	39	36	34	46	54	6	102	-
39	38	37	35	47	55	7	103	-
40	41	42	44	32	56	8	104	-
41	40	43	45	33	57	9	105	-
42	43	40	46	34	58	10	106	-
43	42	41	47	35	59	11	107	-
44	45	46	40	36	60	12	108	-
45	44	47	41	37	61	13	109	-
46	47	44	42	38	62	14	110	-
47	46	45	43	39	63	15	111	-
48	49	50	52	56	32	16	112	-
49	48	51	53	57	33	17	113	-
50	51	48	54	58	34	18	114	-
51	50	49	55	59	35	19	115	-
52	53	54	48	60	36	20	116	-
53	52	55	49	61	37	21	117	-
54	55	52	50	62	38	22	118	-
55	54	53	51	63	39	23	119	-
56	57	58	60	48	40	24	120	-
57	56	59	61	49	41	25	121	-
58	59	56	62	50	42	26	122	-
59	58	57	63	51	43	27	123	-
60	61	62	56	52	44	28	124	-
61	60	63	57	53	45	29	125	-
62	63	60	58	54	46	30	126	-
63	62	61	59	55	47	31	127	-

Table 3-8. Node Neighbors (*sheet 3 of 4, nodes 64 - 95*)

Node #	CH 0	CH 1	CH 2	CH 3	CH 4	CH 5	CH 6	CH 7
64	65	66	68	72	80	96	0	-
65	64	67	69	73	81	97	1	-
66	67	64	70	74	82	98	2	-
67	66	65	71	75	83	99	3	-
68	69	70	64	76	84	100	4	-
69	68	71	65	77	85	101	5	-
70	71	68	66	78	86	102	6	-
71	70	69	67	79	87	103	7	-
72	73	74	76	64	88	104	8	-
73	72	75	77	65	89	105	9	-
74	75	72	78	66	90	106	10	-
75	74	73	79	67	91	107	11	-
76	77	78	72	68	92	108	12	-
77	76	79	73	69	93	109	13	-
78	79	76	74	70	94	110	14	-
79	78	77	75	71	95	111	15	-
80	81	82	84	88	64	112	16	-
81	80	83	85	89	65	113	17	-
82	83	80	86	90	66	114	18	-
83	82	81	87	91	67	115	19	-
84	85	86	80	92	68	116	20	-
85	84	87	81	93	69	117	21	-
86	87	84	82	94	70	118	22	-
87	86	85	83	95	71	119	23	-
88	89	90	92	80	72	120	24	-
89	88	91	93	81	73	121	25	-
90	91	88	94	82	74	122	26	-
91	90	89	95	83	75	123	27	-
92	93	94	88	84	76	124	28	-
93	92	95	89	85	77	125	29	-
94	95	92	90	86	78	126	30	-
95	94	93	91	87	79	127	31	-

Table 3-8. Node Neighbors *(sheet 4 of 4, nodes 96-127)*

Node #	CH 0	CH 1	CH 2	CH 3	CH 4	CH 5	CH 6	CH 7
96	97	98	100	104	112	64	32	-
97	96	99	101	105	113	65	33	-
98	99	96	102	106	114	66	34	-
99	98	97	103	107	115	67	35	-
100	101	102	96	108	116	68	36	-
101	100	103	97	109	117	69	37	-
102	103	100	98	110	118	70	38	-
103	102	101	99	111	119	71	39	-
104	105	106	108	96	120	72	40	-
105	104	107	109	97	121	73	41	-
106	107	104	110	98	122	74	42	-
107	106	105	111	99	123	75	43	-
108	109	110	104	100	124	76	44	-
109	108	111	105	101	125	77	45	-
110	111	108	106	102	126	78	46	-
111	110	109	107	103	127	79	47	-
112	113	114	116	120	96	80	48	-
113	112	115	117	121	97	81	49	-
114	115	112	118	122	98	82	50	-
115	114	113	119	123	99	83	51	-
116	117	118	112	124	100	84	52	-
117	116	119	113	125	101	85	53	-
118	119	116	114	126	102	86	54	-
119	118	117	115	127	103	87	55	-
120	121	122	124	112	104	88	56	-
121	120	123	125	113	105	89	57	-
122	123	120	126	114	106	90	58	-
123	122	121	127	115	107	91	59	-
124	125	126	120	116	108	92	60	-
125	124	127	121	117	109	93	61	-
126	127	124	122	118	110	94	62	-
127	126	125	123	119	111	95	63	-

CUBE LINK TESTS DETAILED DESCRIPTION

The Cube Link Tests (CLTs) are listed in Table 3-9. The CLTs test the host-to-node links, the node-to-node links, and the I/O node-to-node links. The links are tested with DCM messages that are composed of the same patterns of dwords (32 bits) as the Host Link Tests and Node Link Tests (see Table 3-5).

Table 3-9. Cube Link Tests

TEST NAME
Concurrent Receive Test
Router Test
Channel Arbitration Test
Concurrent Router Test
Multi-Hop Test
Host/Neighbor Concurrent Comm Test
Host/Node Concurrent Comm Test
Host/Node Chan Mask Con Comm Test [Extended]
Folded Loopback Test [Extended]

The CLTs are very similar to the Node Link Tests. The difference is that the NLTs check only node-to-node links.

The extended CLT is described later in this chapter. The following sections describe each of the standard CLTs. The standard CLT descriptions also apply to all of the Node Link Tests except the Node Link: Single Xmit & Recv Test.

Cube & Node Link: Concurrent Receive Tests

The Cube & Node Link: Concurrent Receive Tests check each of the capabilities of the DCM to receive concurrently a message from its neighbors. The hardware tested is shown in Figure 3-13.

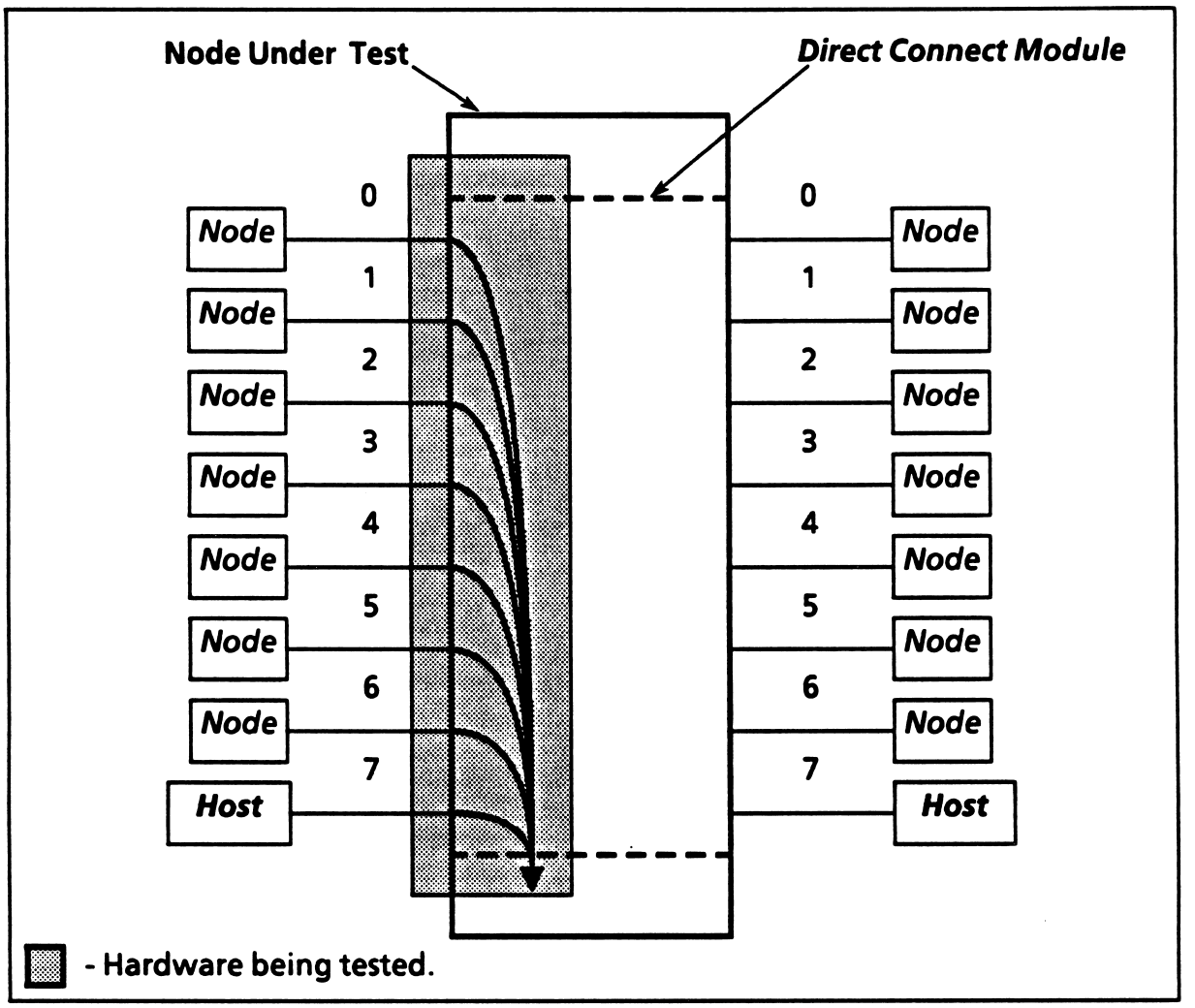


Figure 3-13. Hardware Tested By The Node & Cube Link: Concurrent Receive Test

The Concurrent Receive Test performs as follows:

1. Each node is individually placed under test.
2. All neighbors are simultaneously commanded to transmit to the Node Under Test.
3. The Node Under Test checks all of the messages.

Cube & Node Link: Router and Concurrent Router Tests

The Cube & Node Link: Router and Concurrent Router Tests check the DCM message routing. The Router Tests check each router individually and the Concurrent Router Tests check the routers simultaneously. The hardware under test is shown in Figure 3-14.

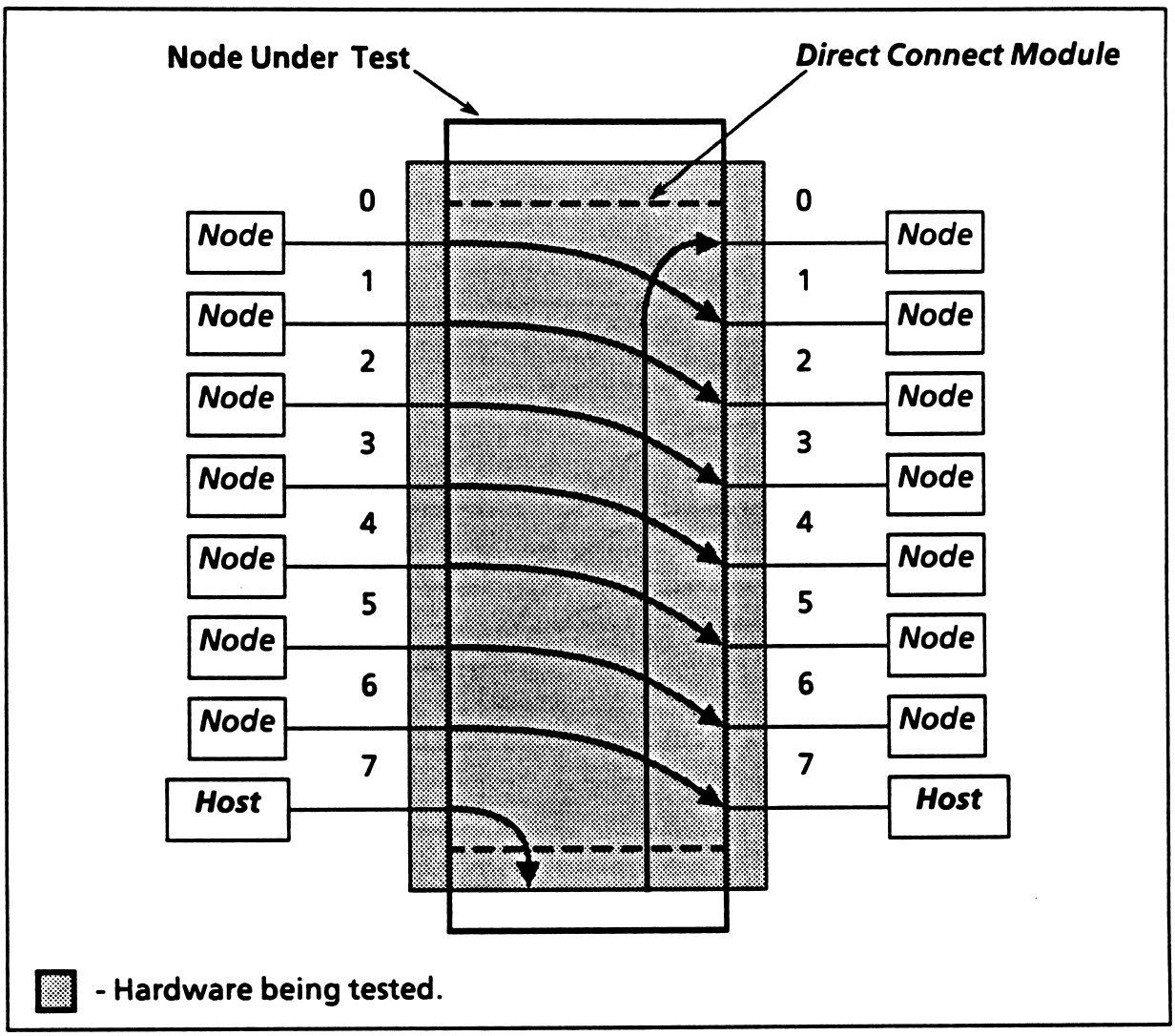


Figure 3-14. Hardware Tested By The Node & Cube Link: Router and Concurrent Router Test

The Router and Concurrent Router Tests check each node individually. The input neighbors of the Node Under Test are commanded to transmit to the output neighbors of the Node Under Test. Then, the output neighbors check the messages.

Cube & Node Link: Channel Arbitration Tests

The Cube & Node Link: Channel Arbitration Tests check the capabilities of each DCM router for arbitrating messages simultaneously through each channel. The hardware shown in Figure 3-15 is an example of channel 3 under test.

The Channel Arbitration Tests check each channel of each node individually. All input neighbors of the Node/Channel Under Test as well as the Node Under Test simultaneously transmit to the output neighbor of the Channel Under Test. The output neighbor then checks all messages.

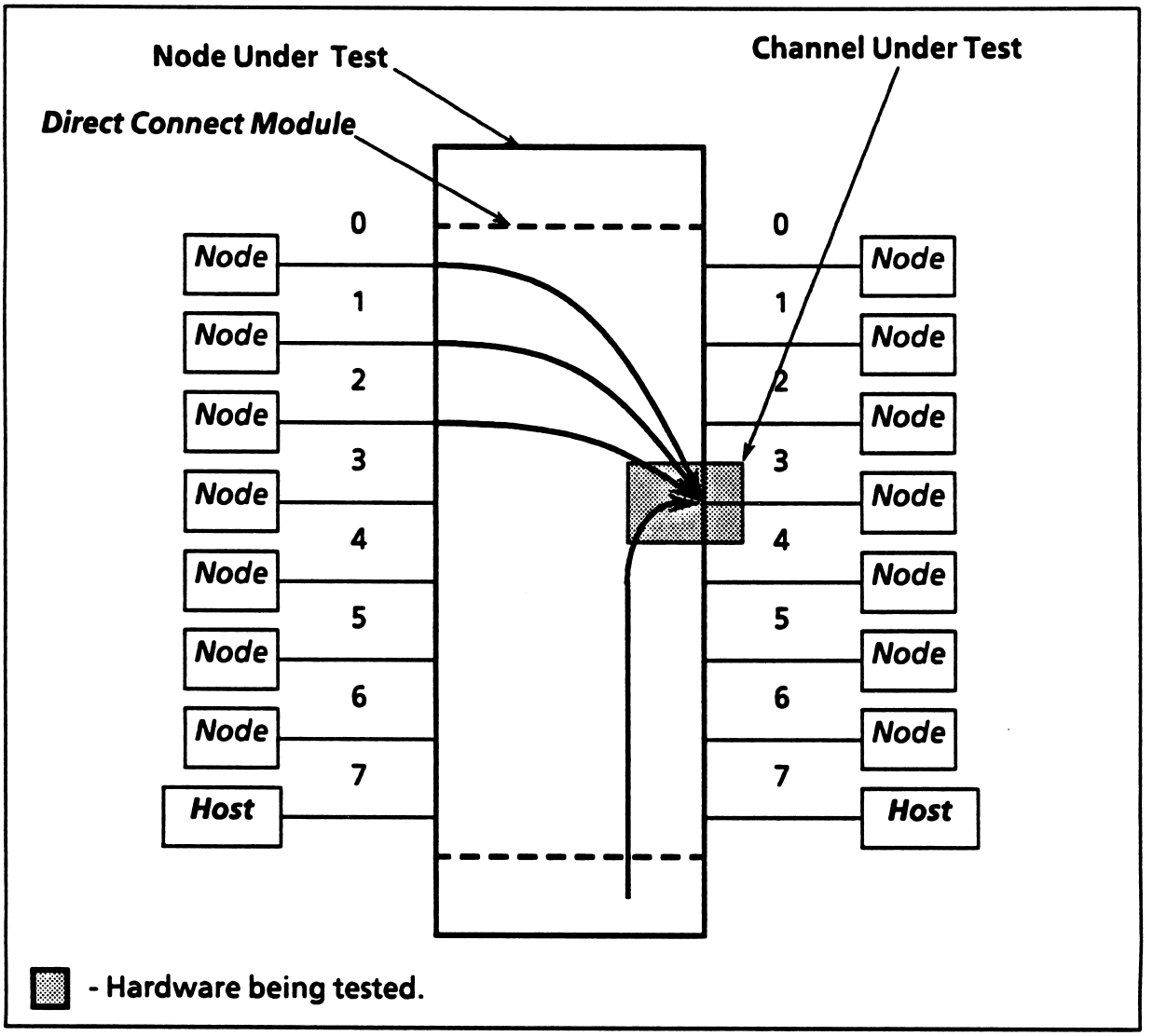


Figure 3-15. Hardware Tested By The Node & Cube Link: Channel Arbitration Test

Cube & Node Link: Multi-Hop Tests

The Cube & Node Link: Multi-Hop Tests individually check the capability of each node to multi-hop communications. Multi-hop communications skip at least two channels. For example, node 0 to node 7 multi-hops through nodes 1 and 3 via channels 0 and 1 (see Table 3-8).

Cube & Node Link: Neighbor & Host/Neighbor Concurrent Comm Tests

The Cube & Node Link: Neighbor & Host/Neighbor Concurrent Comm Tests simultaneously check all node DCM capabilities for communicating between each node and its neighbor. Each node transmits to its neighbor, receives, and checks messages as fast as possible. The hardware under test is shown in Figure 3-16.

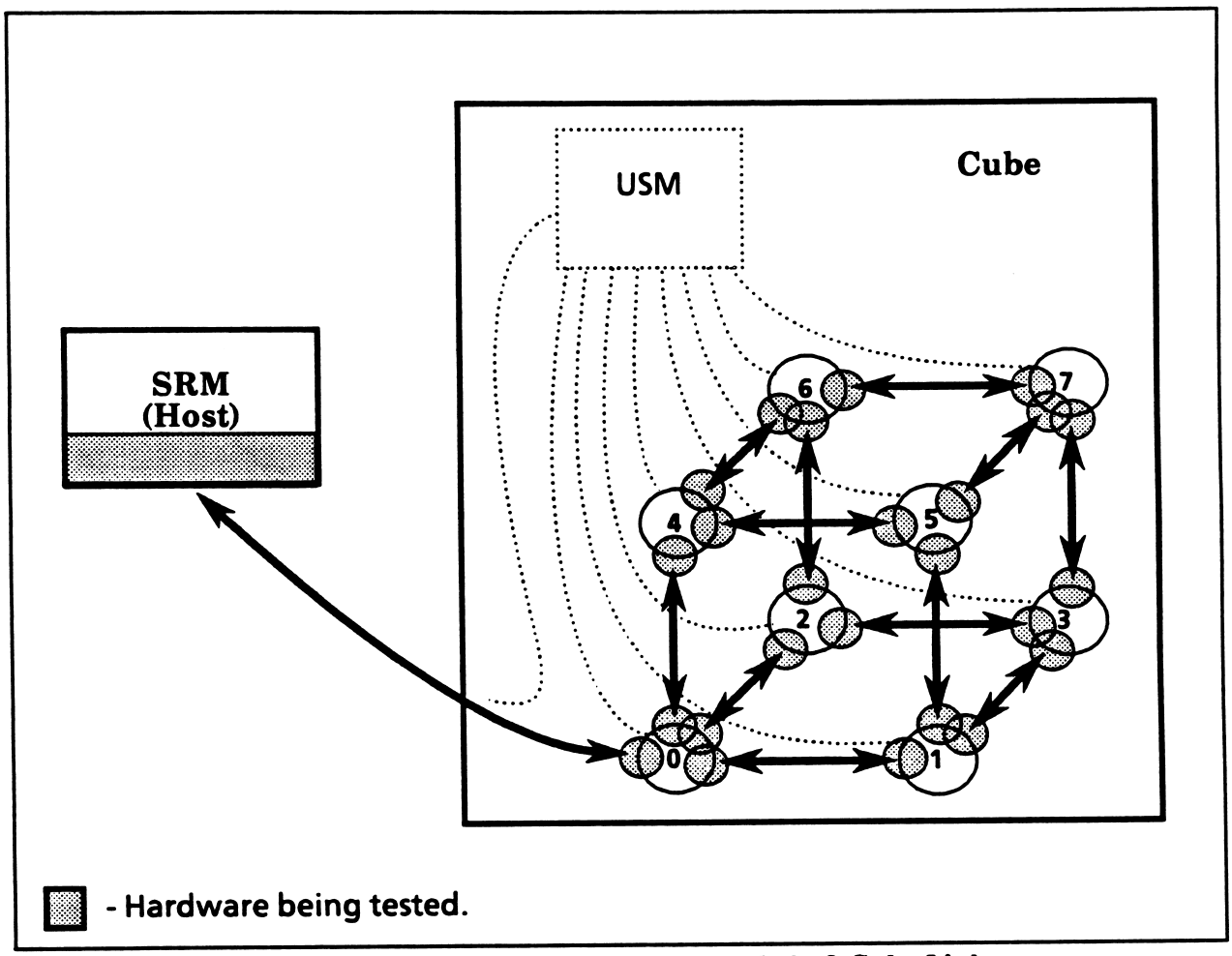


Figure 3-16. Hardware Tested by The Node & Cube Link: Neigh. & Host/Neighbor Concurrent Comm Tests

Cube & Node Link: Node & Host/Node Concurrent Comm Tests

The Cube & Node Link: Node & Host/Node Concurrent Comm Tests simultaneously check all node DCM capabilities for communicating between each node and every other node in the system. Each node transmits to all other nodes including itself, receives, and checks messages as fast as possible. The hardware under test is shown in Figure 3-17.

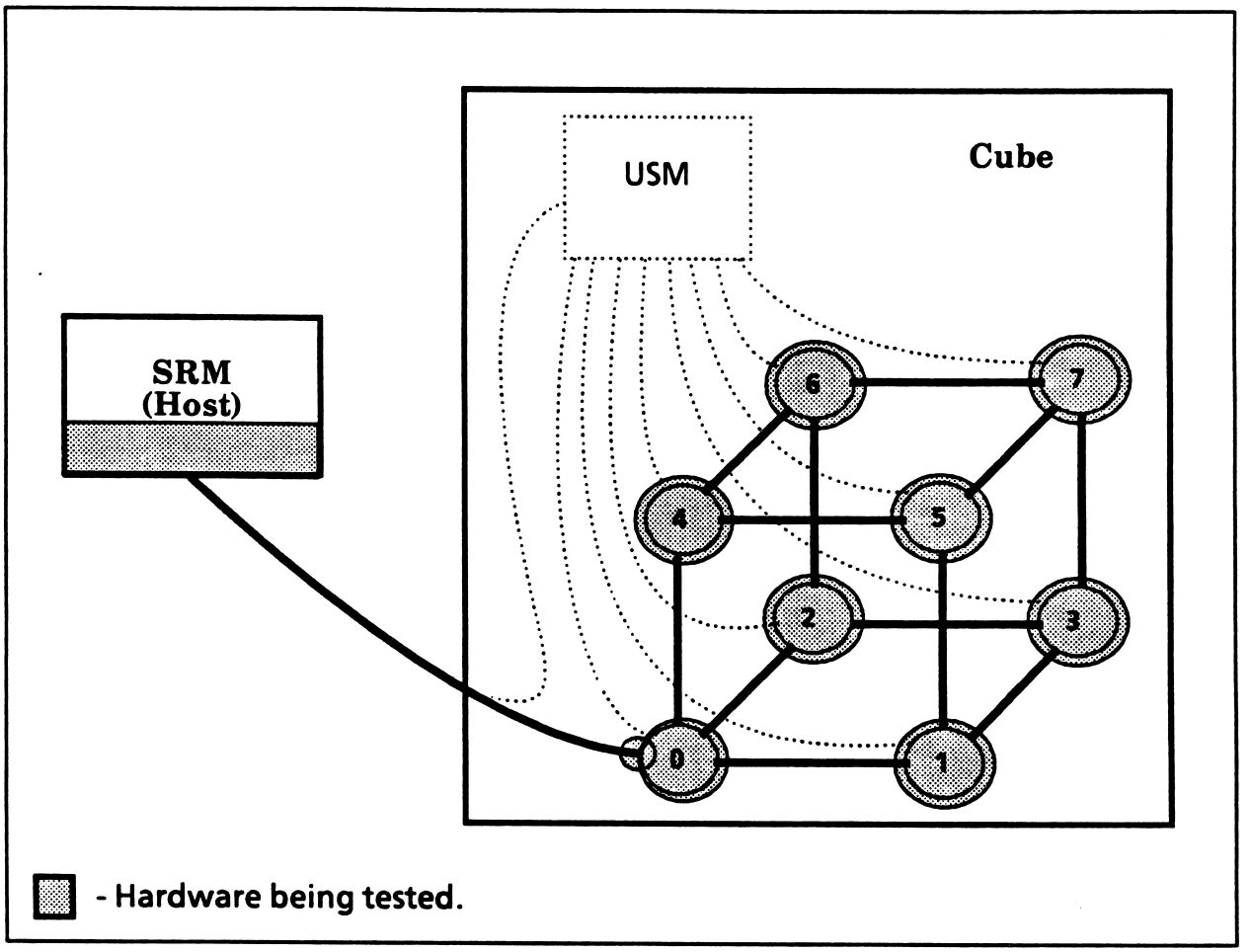


Figure 3-17. Hardware Tested by The Node & Cube Link: Node & Host/Node Concurrent Comm Tests

GENERIC LINK TEST DETAILED DESCRIPTION

The Generic Link Tests (GLTs) consist of one Extended test called the Multi-Generic Test.

EXTENDED TESTS DETAILED DESCRIPTION

The Extended Tests are listed in Table 3-10. Extended tests are used primarily by factory trained Manufacturing and Engineering personnel and require special hardware or knowledge of the system. Extended tests are not typically run in the field.

Table 3-10. Extended Tests

Test Name
Diag Link: Cable & USM Backplane Loopback Test
Diag Link: USM & Node Backplane Loopback Test
Host Link: Cable & Node Backplane Loopback Test
I/O Link: Cable & I/O Node Loopback Test
Cube Link: Host/Node Chan Mask Con Comm Test
Cube Link: Folded Loopback Test
Generic Link: Multi-Generic Test

NOTE

To obtain the most performance from CDP, read the section entitled "Hardware Test States" near the end of this section.

The following paragraphs describe each of the Extended Tests in detail, except for the optional-hardware tests.

Diag Link: Cable & USM Backplane Loopback Test [Extended]

The Cable & USM Backplane Loopback Test [Extended] assists in fault isolation of Diagnostic Link cable and cube backplane problems between the SRM and Cube USM. The test is designed to run with a loopback connector installed (see Appendix F). The loopback points are shown in Figure 3-18.

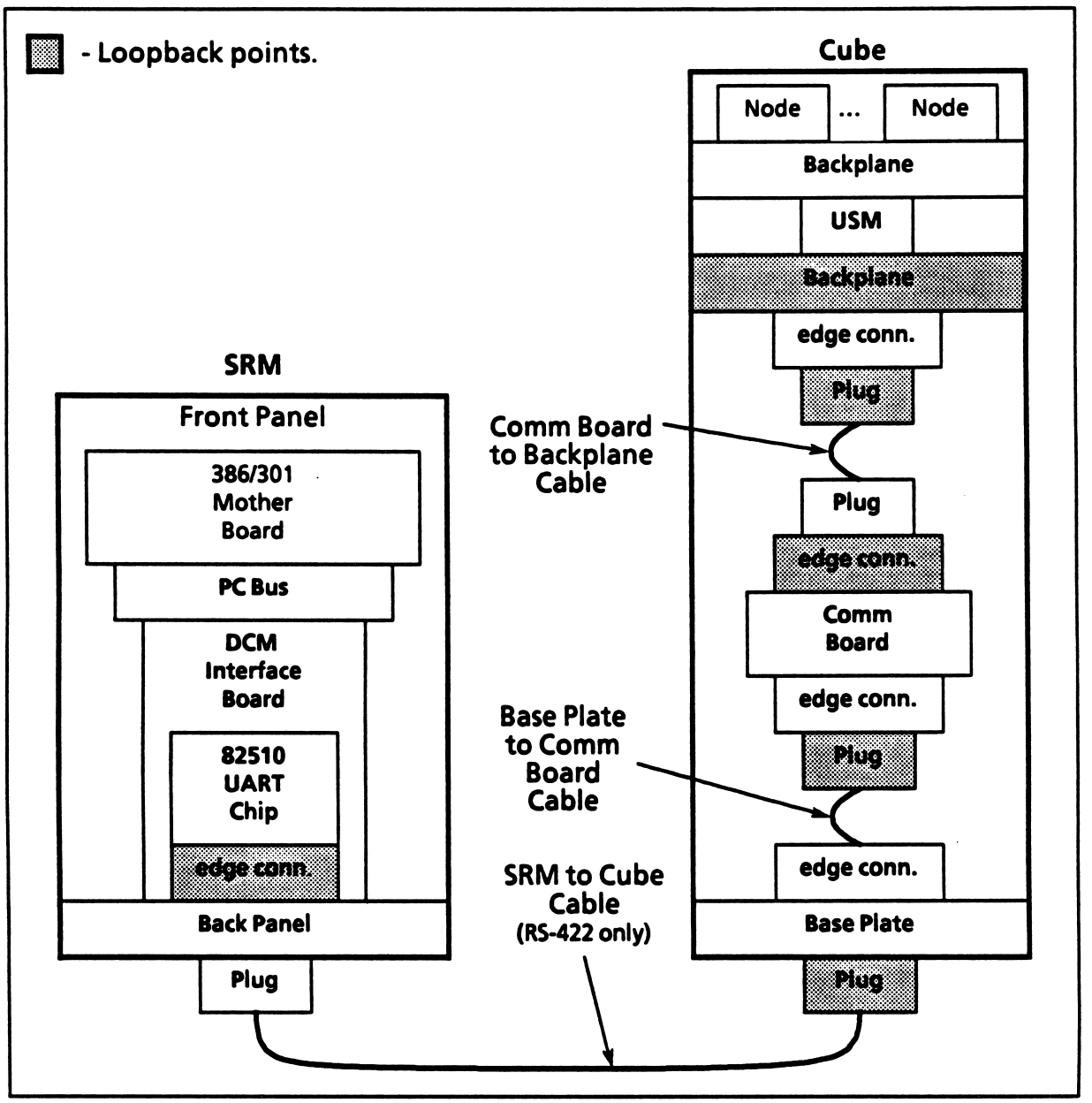


Figure 3-18. Loopback Points for Diag Link: Cable & USM Backplane Loopback Test [Extended]

Diag Link: USM & Node Backplane Loopback Test [Extended]

The USM & Node Backplane Loopback Test [Extended] assists in fault isolation of Diagnostic Link cube backplane problems between the SRM and USM and between the USM and each node. The test is designed to run with a loopback connector installed (see Appendix F). The loopback point is shown in Figure 3-19.

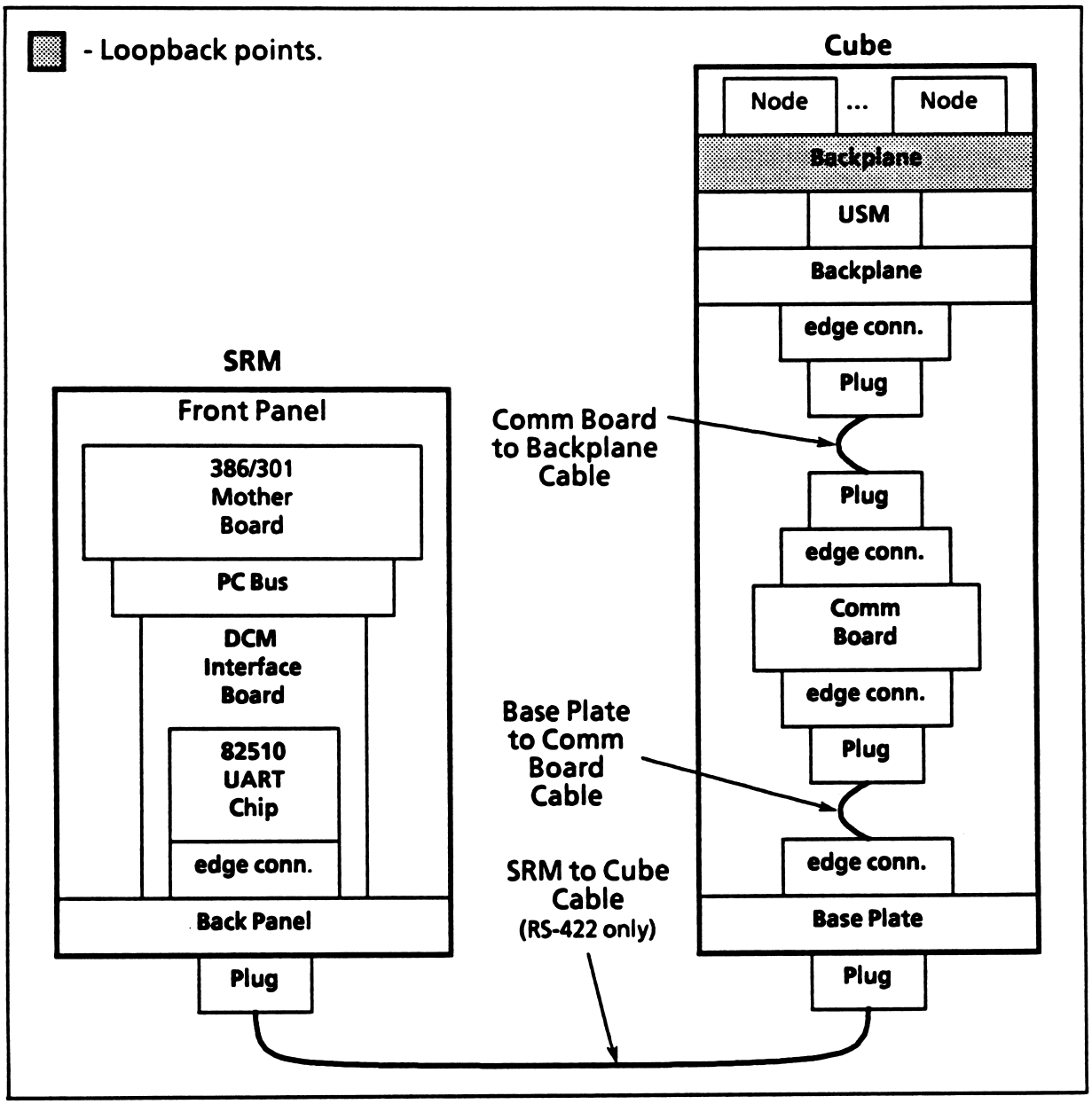


Figure 3-19. Loopback Point for Diag Link: USM & Node Backplane Loopback Test [Extended]

Host Link: Cable & Node Backplane Loopback Test [Extended]

The Host Link: Cable & Node Backplane Loopback Test [Extended] assists in fault isolation of Host Link cable and cube backplane problems between the SRM and cube backplane. The test is designed to run with a loopback connector installed (see Appendix F). The loopback points are shown in Figure 3-20.

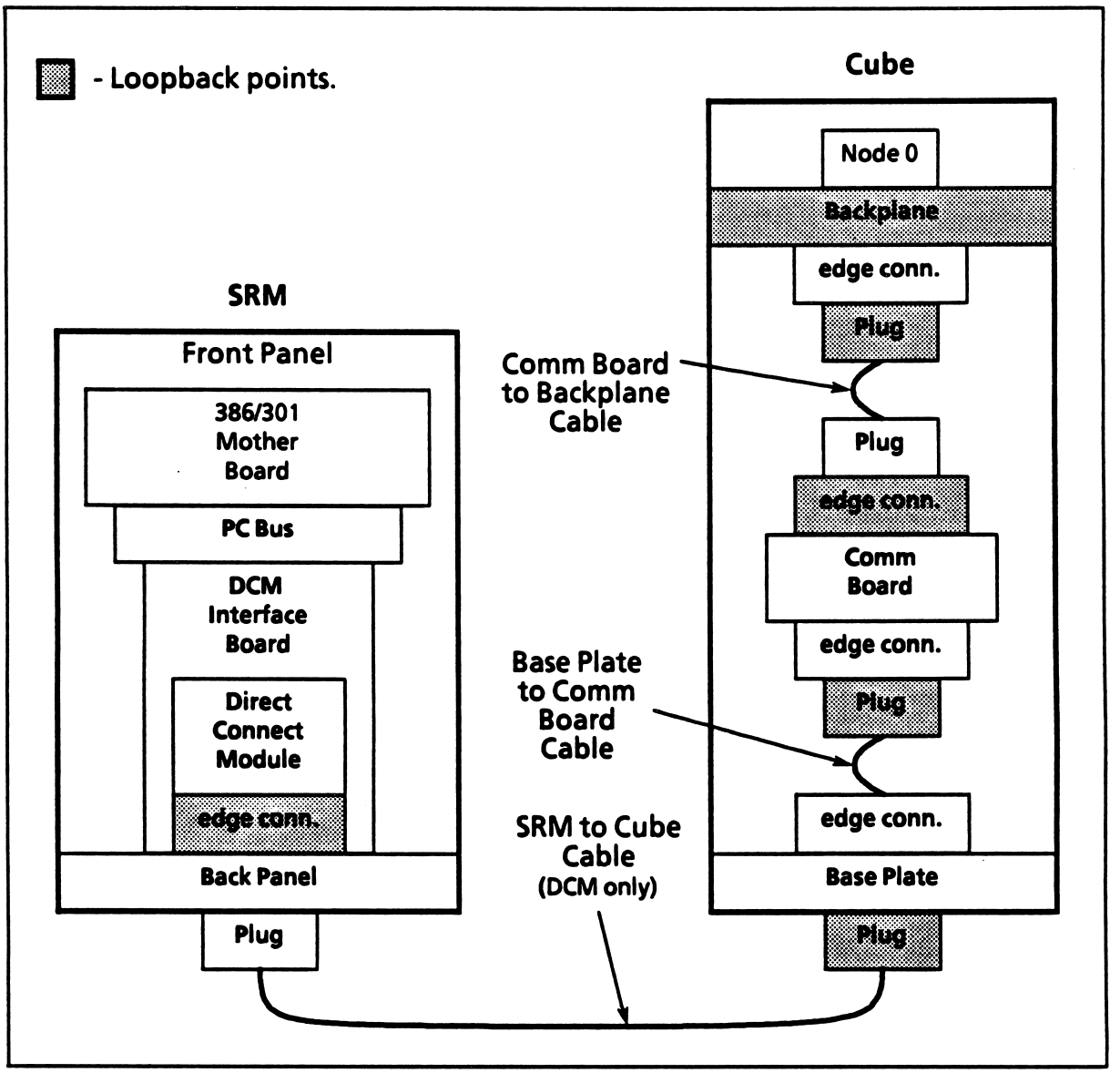
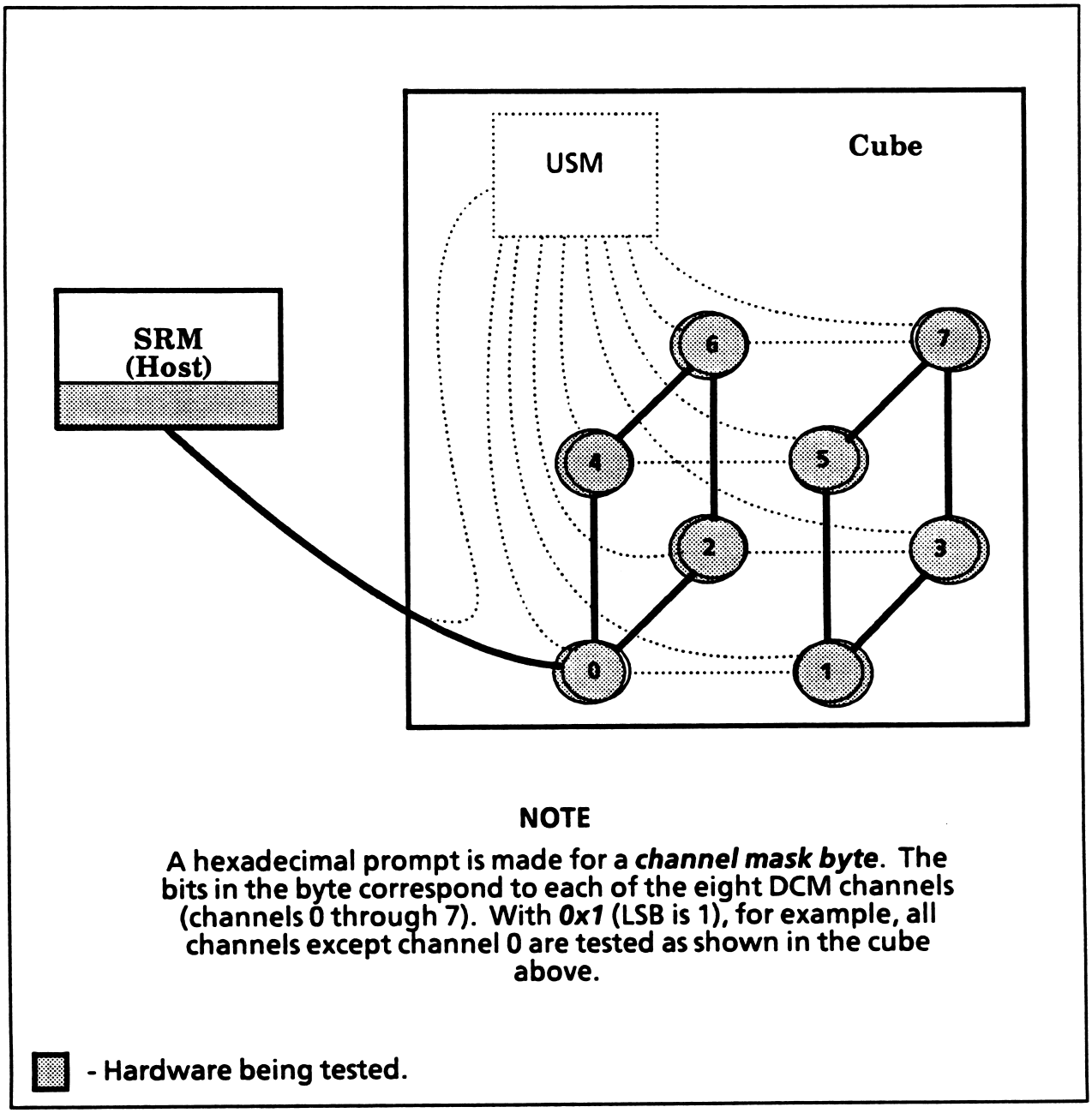


Figure 3-20. Loopback Points for The Host Link: Cable & Node Backplane Loopback Test

Cube Link: Host/Node Chan Mask Con Comm Test [Extended]

The Cube Link: Host/Node Chan Mask Con Comm Test [Extended] assists in engineering debug of the DCM. Figure 3-21 shows what the test does and how it is run.



NOTE

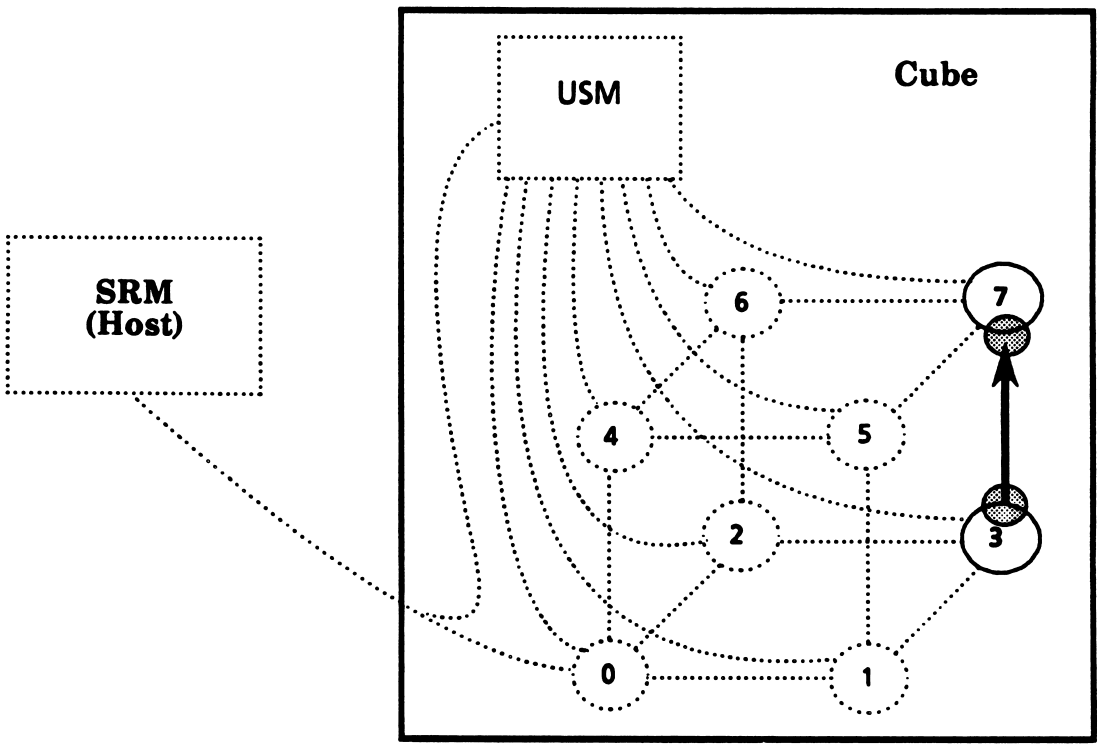
A hexadecimal prompt is made for a ***channel mask byte***. The bits in the byte correspond to each of the eight DCM channels (channels 0 through 7). With ***0x1*** (LSB is 1), for example, all channels except channel 0 are tested as shown in the cube above.

- Hardware being tested.

Figure 3-21. Features of The Cube Link: Host/Node Chan Mask Con Comm Test

Generic Link: Generic Test [Extended]

The Generic Link: Generic Test [Extended] assists in engineering debug of the DCM. Figure 3-22 shows what the test does and how it is run.



NOTE

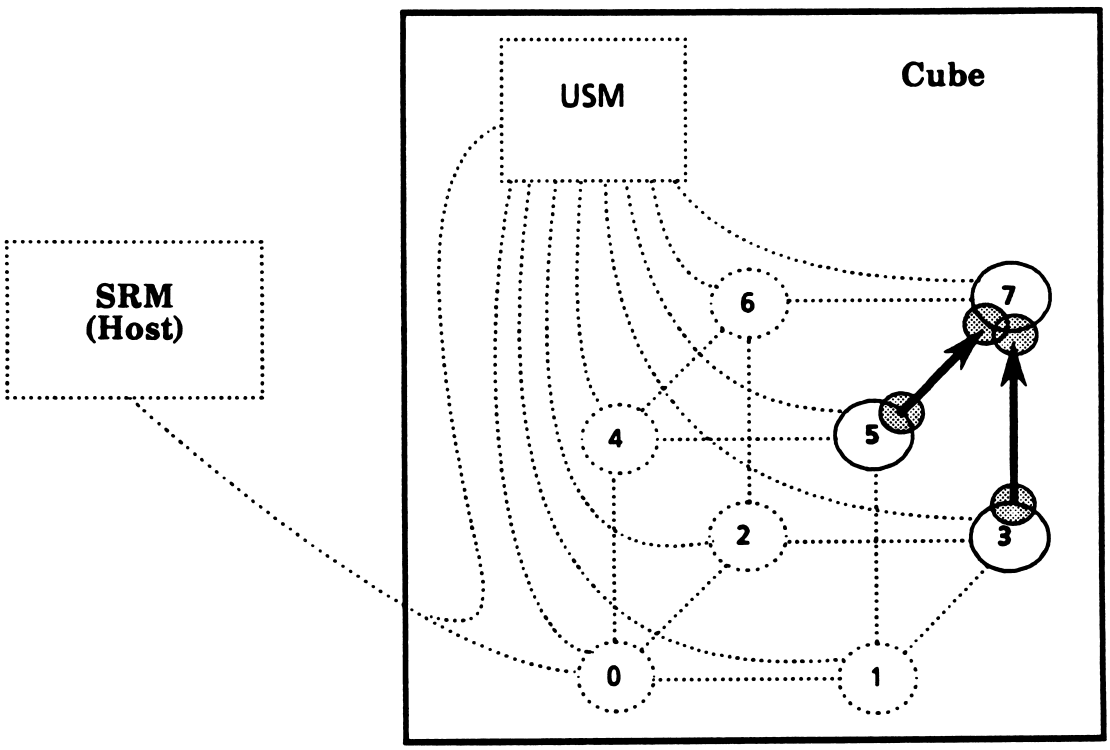
Two hexadecimal prompts are made -- one for a ***Source Node*** and one for a ***Destination Node***. The single DCM channel between the two nodes is then tested. For example, if 3 is entered for source and 7 for destination, this single link is tested as shown in the cube above.

▒ - Hardware being tested.

Figure 3-22. Features of The Generic Link: Generic Test

Generic Link: Multi-Generic Test [Extended]

The Generic Link: Multi-Generic Test [Extended] assists in engineering debug of the DCM. Figure 3-23 shows what the test does and how it is run.



NOTE

A prompt is made for the ***Number of Messages*** and for each ***Source*** and ***Destination Node***. Each DCM channel between the nodes is then tested. For example, if 2 is entered for Number of Messages, 3 for first source, 7 for first destination, 5 for second source, and 7 for second destination, the links are tested as shown above.

▒ - Hardware being tested.

Figure 3-23. Features of The Generic Link: Multi-Generic Test

OPTIONAL-HARDWARE TESTS

The following tests check the functionality of optional hardware. If an item of optional hardware is installed, the associated test or tests will run.

SX and 387 Processor Tests

The SX and 387 coprocessors both execute a set of tests called the paranoia tests. The paranoia tests consist of the basic mathematical operations necessary to demonstrate that the coprocessors conform to the standards set by IEEE. Any failure reported by these tests mean that the coprocessor must be replaced. It may also indicate that the node to which it is attached must be replaced.

VX Processor Tests

These tests check the fundamental capabilities of the Vector Processor Board. The tests are listed in Table 3-11. When a test fails, it probably indicates that the VX board must be replaced. However, in some instances, the node that communicates with the VX board that fails is at fault. The node board is in the slot numbered one less than the VX board. For example, the node in slot 16 communicates with the VX board in slot 17.

Table 3-11. VX Processor Tests

Test Name
Registers Tests
Memory Tests
Address and Data Tests
Arithmetic Tests

These VX Processor Tests are subdivided into five clusters of tests as described in the following paragraphs.

VX: REGISTER TESTS

These tests check the VX board registers, and include the following:

Base Address Register Test
Interconnect Register Test
Control & Status Register Test
Command Register Test

VX: MEMORY TESTS

These tests check the VX board program, static, and dynamic memory from the node board point of view across the LBX II interface, and include the following:

Program Memory Data Path Test
Program Memory Data Line Test
Program Memory Bit Test
Program Memory Address Test
Program Memory Uniqueness Test
Static Memory Data Path Test
Static Memory Data Line Test
Static Memory Bit Test
Static Memory Address Test
Static Memory Uniqueness Test
Dynamic Memory Test

VX: ADDRESS AND DATA TESTS

These tests check the onboard address and data buses on the VX board. Special microcode to execute these tests is downloaded to the VX board. Test results are then passed back to the node board via the LBX II interface. The tests include the following:

Sequencer Test
Feedback/RALU Test
Static Ram Mcode Test
Dynamic Ram Mcode Test

VX: ARITHMETIC TESTS

These tests check the VX board arithmetic and logic unit circuits. Special microcode to execute these tests is downloaded to the VX board. Test results are then passed back to the node board via the LBX II interface. The tests include the following:

ALU & MUL Register Test
32-Bit Data Path Test
Single Precision ALU Logical Test
Single Precision ALU Bi-Val Test
Single Precision Arithmetic Test
64-Bit Data Path Test
Double Precision ALU Bi-Val Test
Double Precision Arithmetic Test
FFT 4k SP Complex Test

I/O Tests and Utilities

The following paragraphs describe the various I/O tests and utilities. These include the SCSI Module Tests, Hard Disk Drive Tests, and Tape Drive Tests. Figure 3-24 shows the I/O diagnostic hardware model, which is used as an example for the hardware tested in the I/O tests.

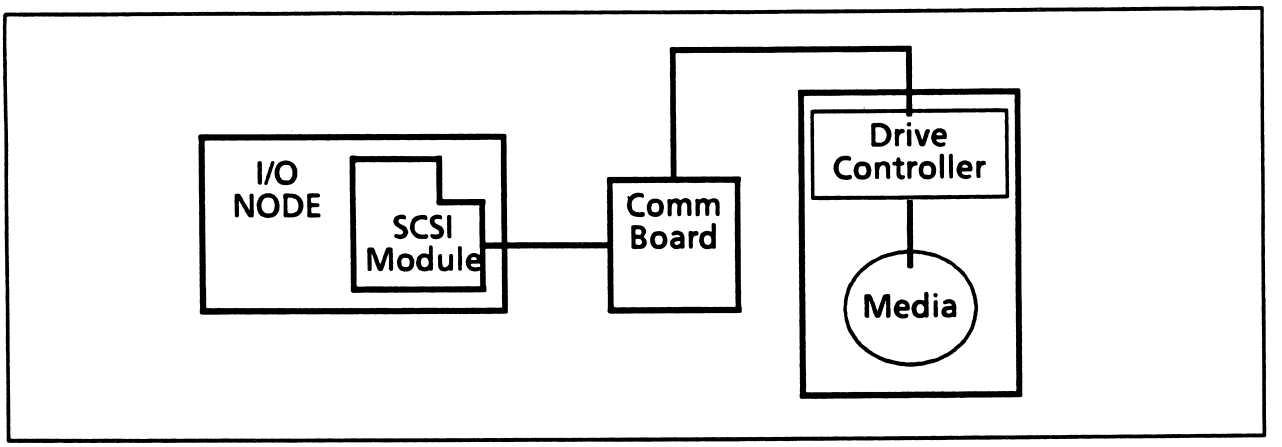


Figure 3-24. The I/O Diagnostic Hardware Model

SCSI MODULE TESTS

A SCSI module serves as the interface between an I/O node and the controller for the device. The SCSI module tests (as listed in Table 3-12) check as much of the SCSI Module hardware as possible without actually having a device attached to the SCSI bus (see Figure 3-25). The diagnostic hardware model for the SCSI module is shown in Figure 3-26.

Table 3-12. SCSI Module Tests

Test Name
Mode Select Test
FIFO Select Test
FIFO Flags Test
FIFO Address Lines Test
FIFO Pattern Sensitivity Test [Extended]
ESP Register Data Lines Test
ESP Register Uniqueness Data Test
ESP FIFO Uniqueness Data Test
ESP Interrupt Test

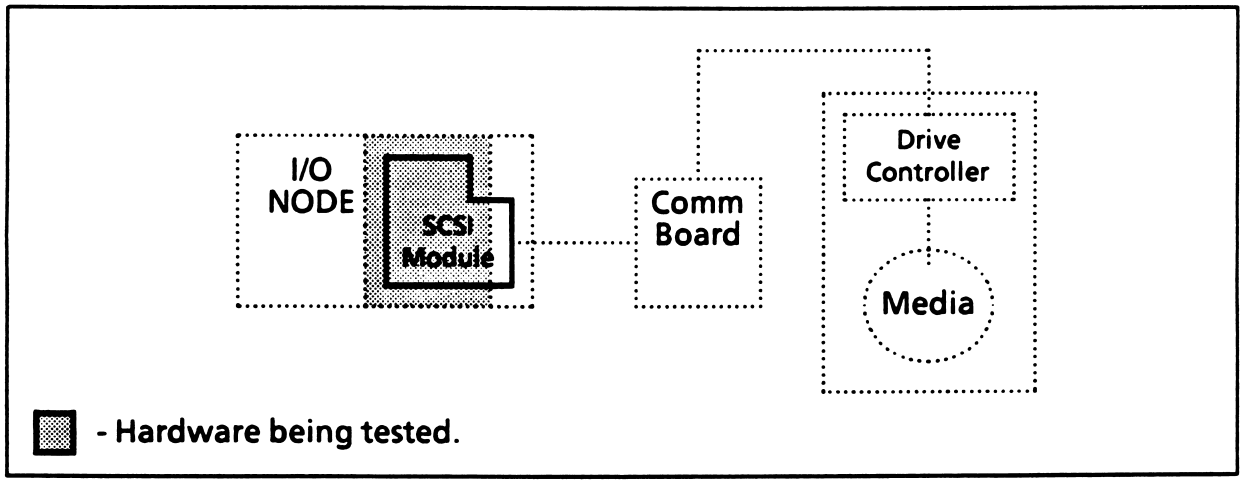


Figure 3-25. Hardware Tested by the SCSI Module Tests

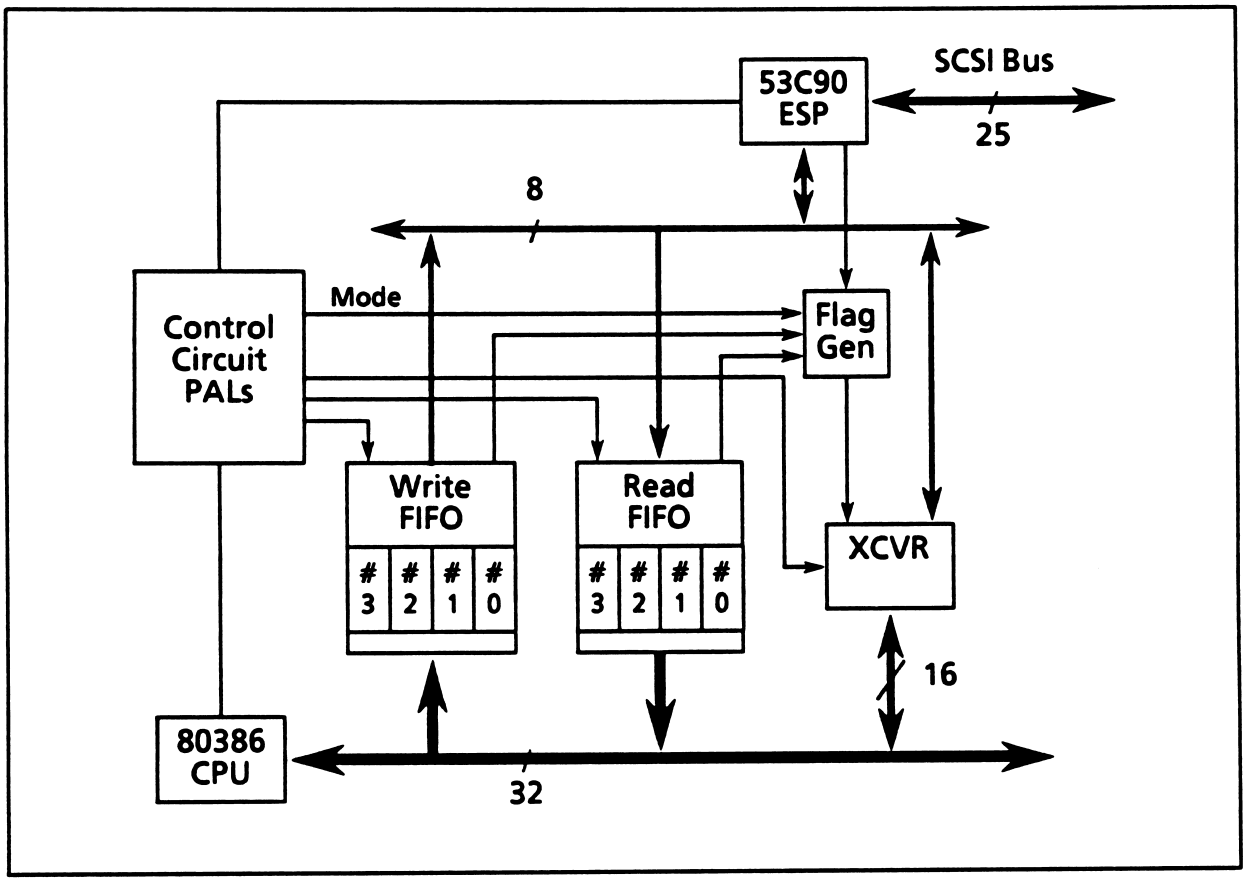


Figure 3-26. The SCSI Module Diagnostic Hardware Model

SCSI Module: Mode Select Test

This test checks the SCSI Module mode bit to make certain that it can be set and retained for both read and write mode (see Figure 3-27).

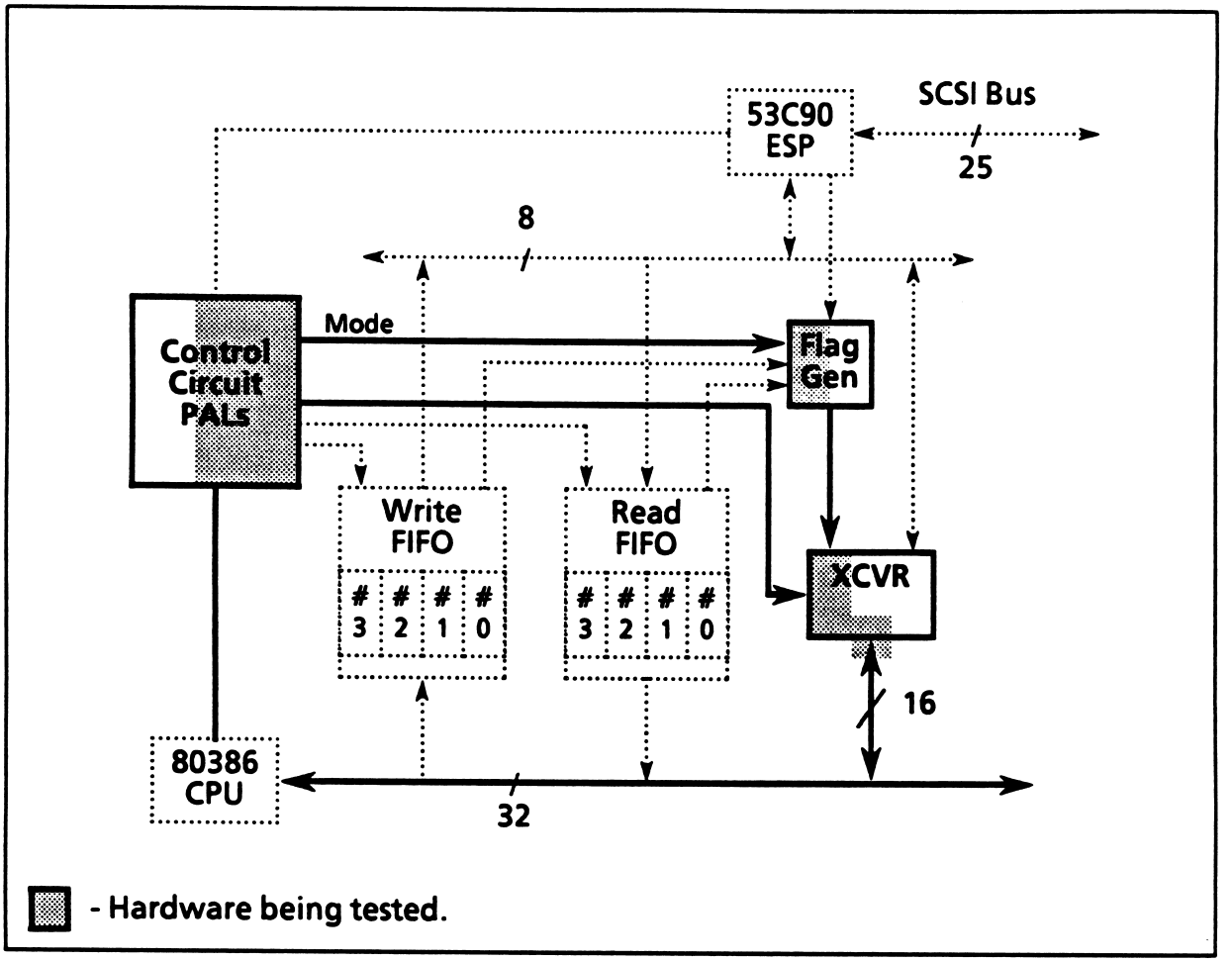


Figure 3-27. Hardware Tested by the SCSI Module: Mode Select Test

SCSI Module: FIFO Data Lines Test

This test walks a one and then a zero across the FIFO data bus. Both the read and write FIFOs are checked (see Figure 3-28).

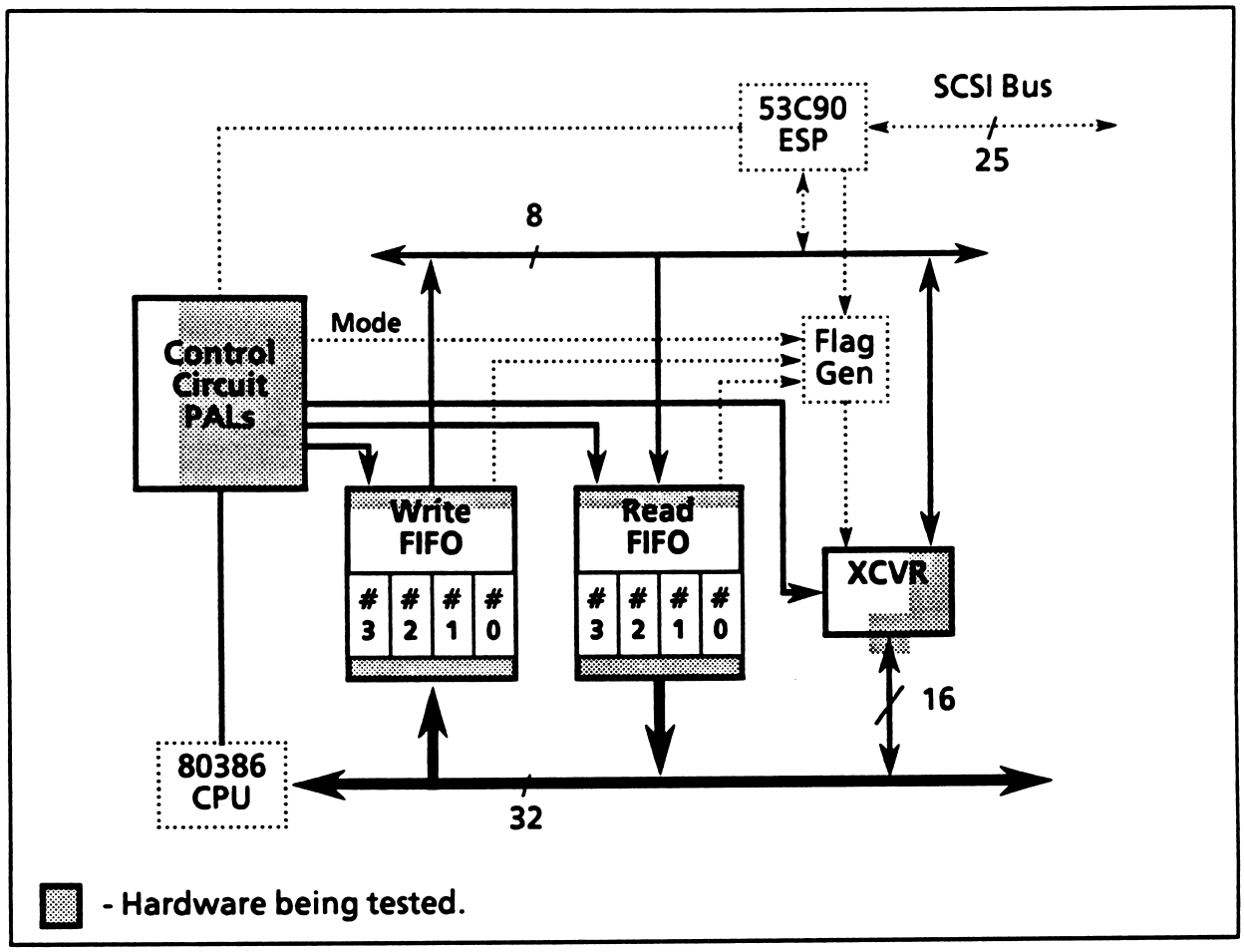


Figure 3-28. Hardware Tested by the SCSI Module: FIFO Data Lines Test

SCSI Module: FIFO Flags Test

This test checks the empty, half, and full flags of both the read and write FIFOs. It checks the flags at all thresholds of zero bytes to maximum bytes per FIFO (see figure 3-29).

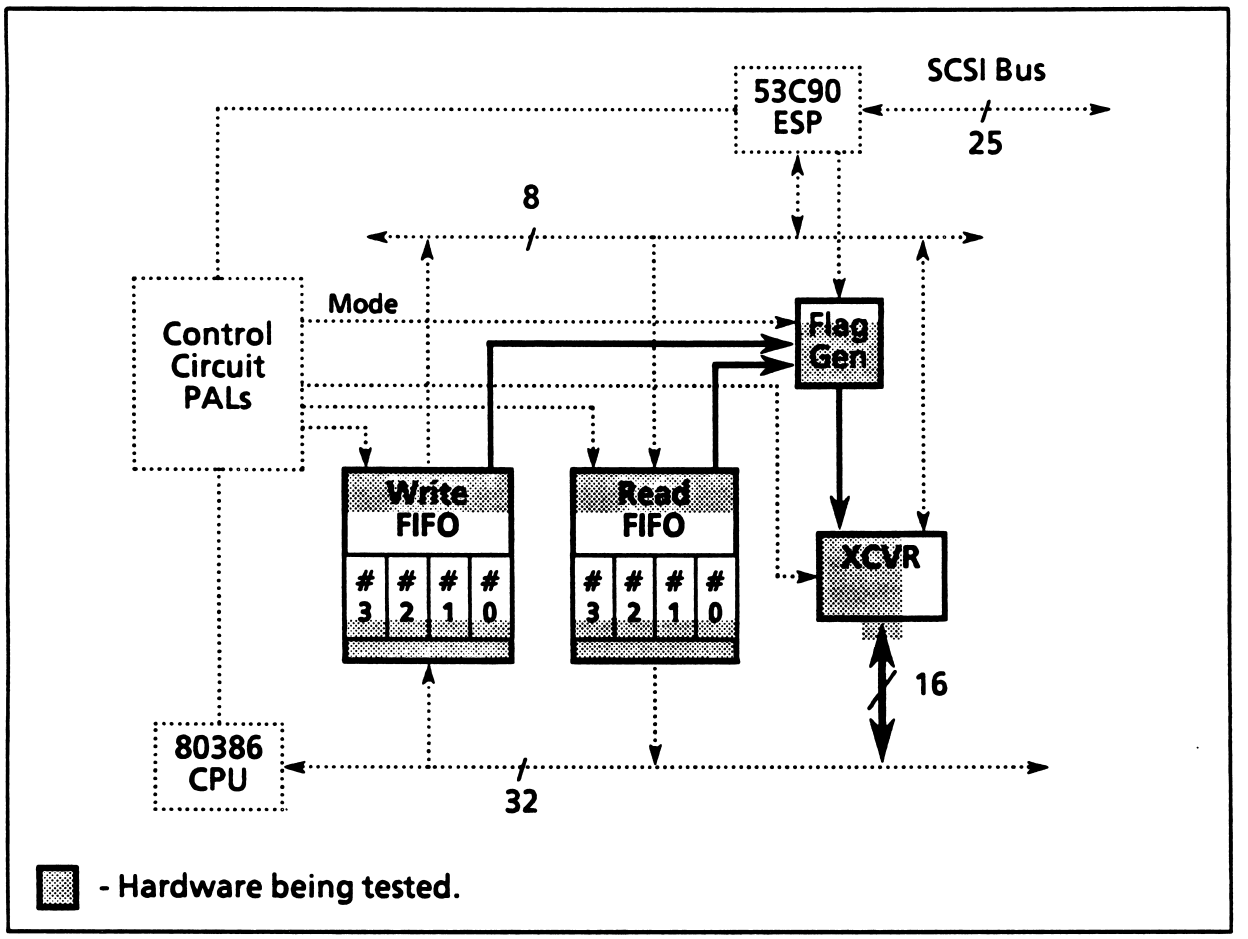


Figure 3-29. Hardware Tested by the SCSI Module: FIFO Flags Test

SCSI Module: FIFO Address Lines Test

This test checks the capability of the FIFOs to store unique data. The FIFOs are loaded with a unique counting pattern between 0 and 251 and the data are then read and verified. Both read and write FIFOs are checked (see Figure 3-30).

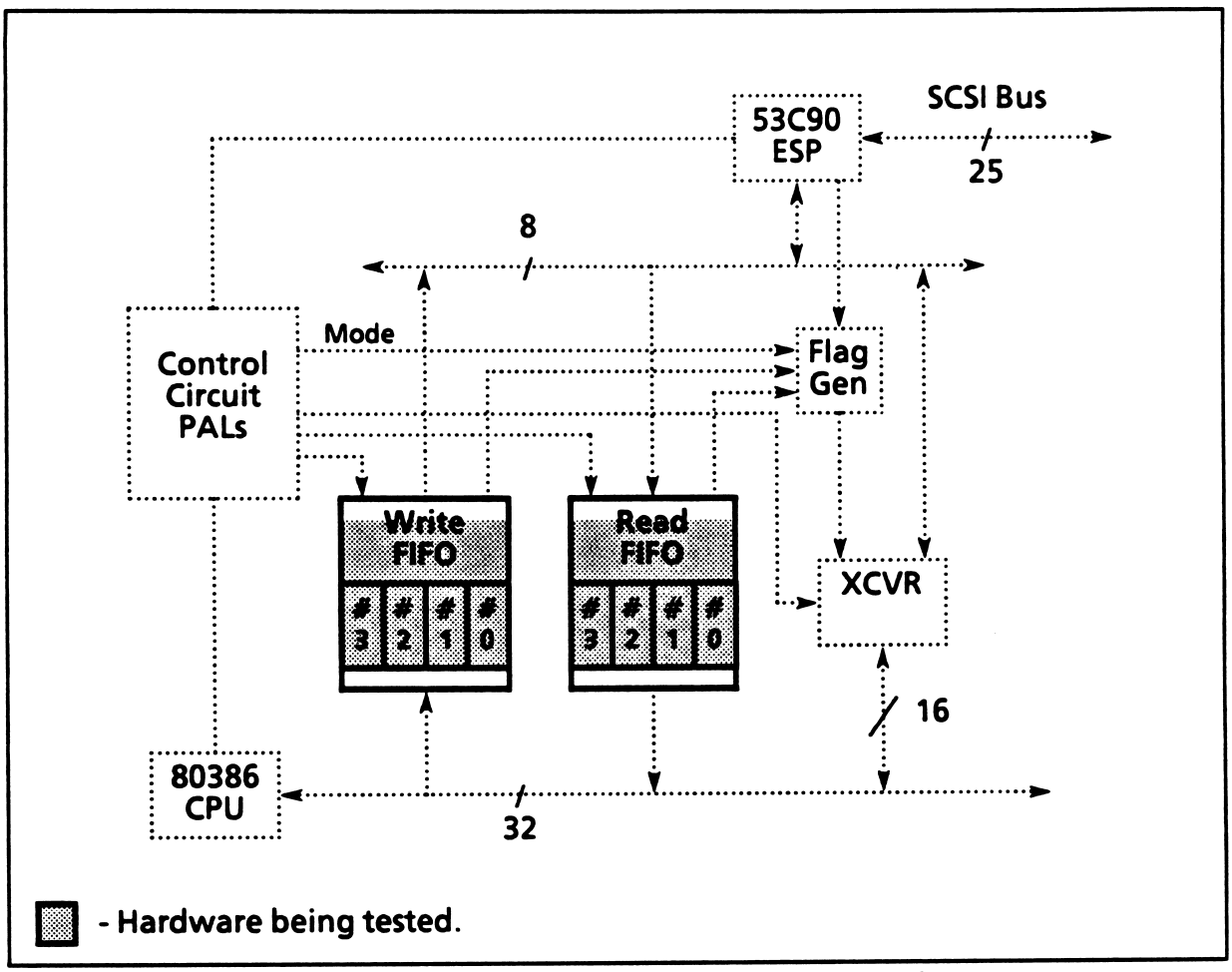


Figure 3-30. Hardware Tested by the SCSI Module: FIFO Address Lines & Pattern Sensitivity Tests

SCSI Module: FIFO Pattern Sensitivity (Extended) Test

This test is an extended type that writes, reads, and verifies all possible patterns of data with the FIFOs (see Figure 3-30). Note that this test requires approximately 8 hours for completion.

SCSI Module: ESP Register Data Lines Test

This test uses an internal SCSI processor register that is writable and readable. The test walks a one and then a zero across the register (see Figure 3-31).

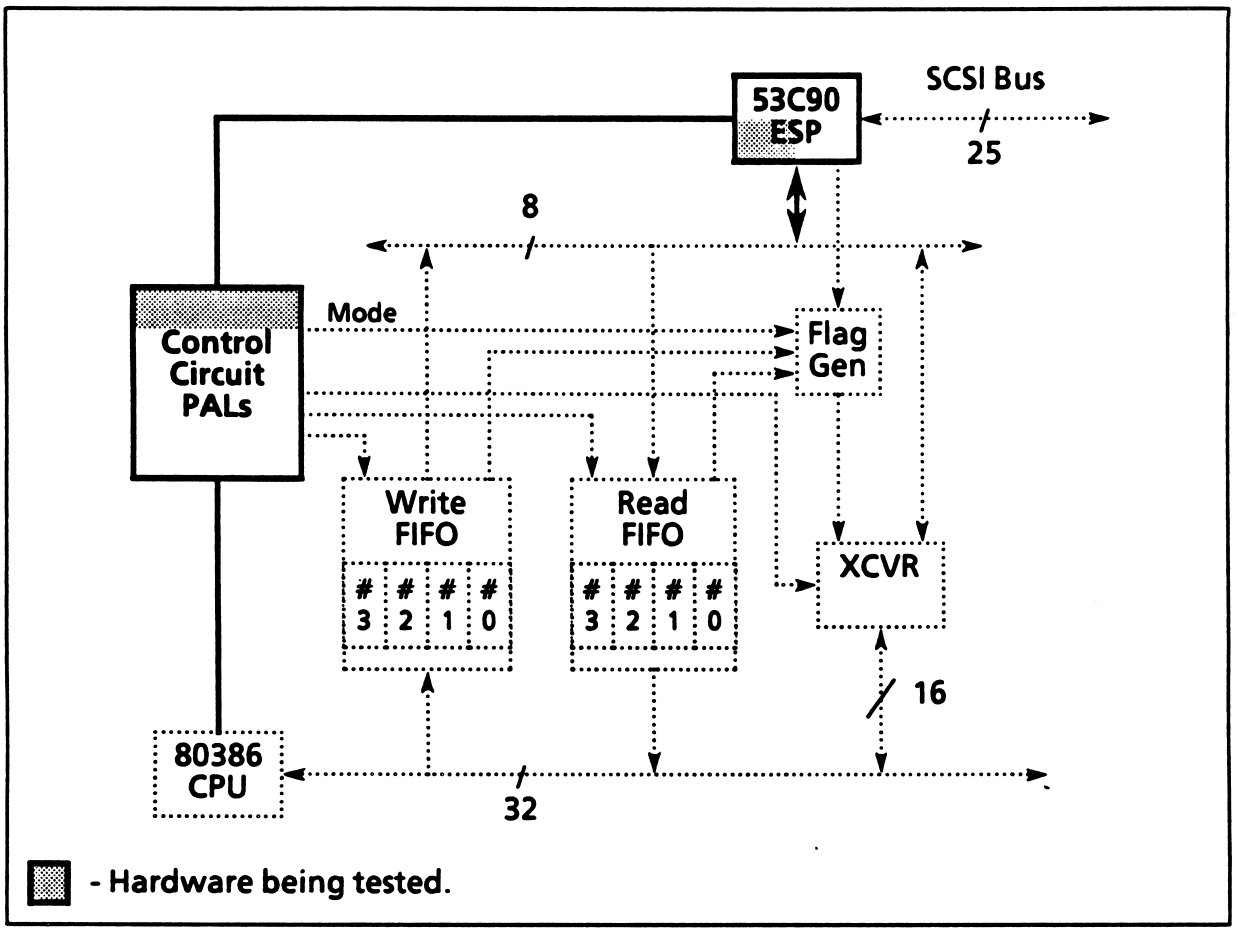


Figure 3-31. Hardware Tested by the SCSI Module: ESP Register Data Lines Test

SCSI Module: ESP Register Uniqueness Data Test

This test uses an internal SCSI processor register that is writable and readable. The test writes, reads, and verifies all possible data patterns with the register (see Figure 3-32).

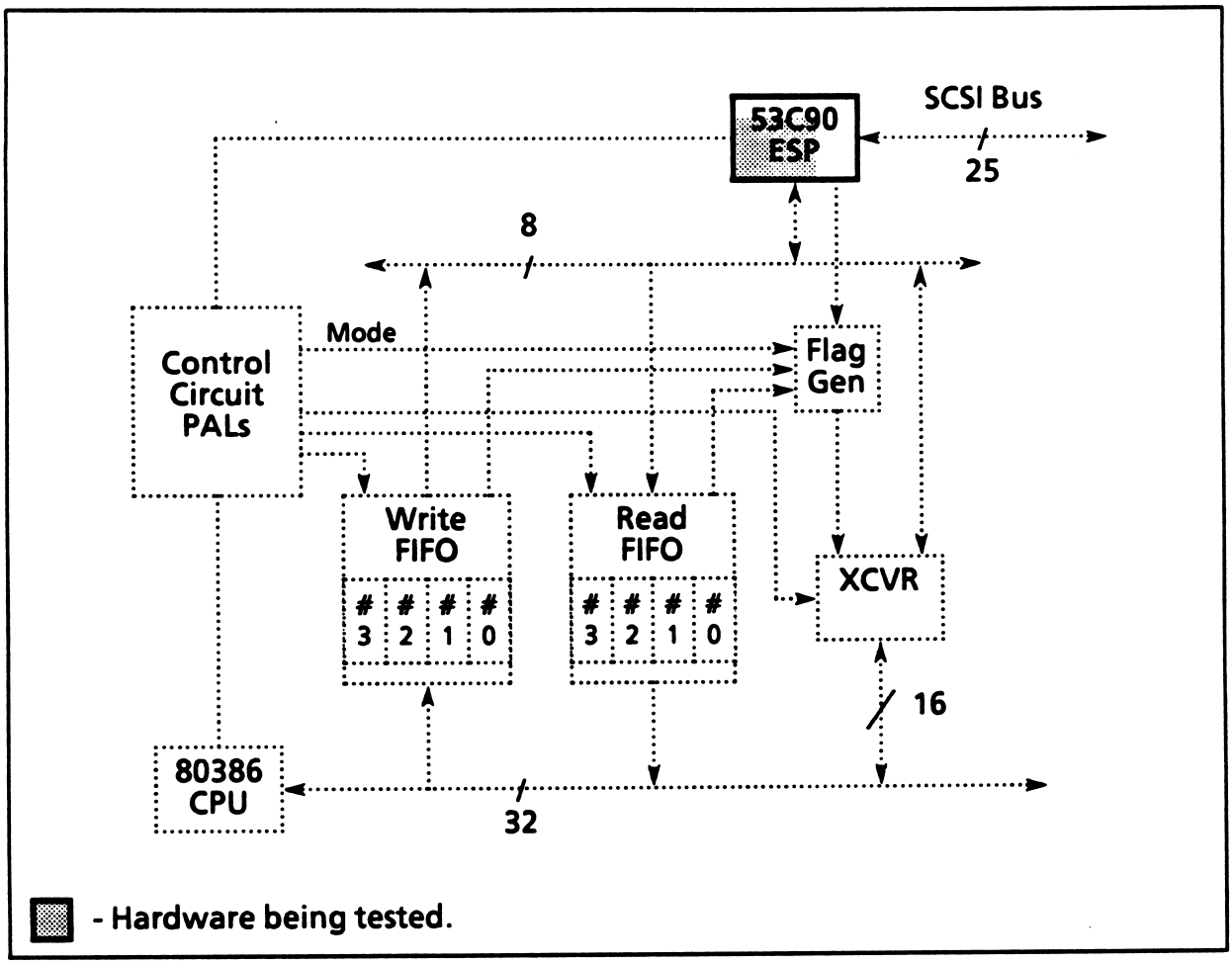


Figure 3-32. Hardware Tested by the SCSI Module: ESP Register & FIFO Uniqueness Tests

SCSI Module: ESP FIFO Uniqueness Data Test

This test loads the SCSI processor FIFO with a unique pattern, reads the FIFO, and verifies the data written (see Figure 3-32).

SCSI Module: ESP Interrupt Test

This test issues an illegal command to the SCSI processor and monitors that the processor asserts its interrupt. The test then issues a legal command to the processor and monitors that it deasserts its interrupt (see Figure 3-33).

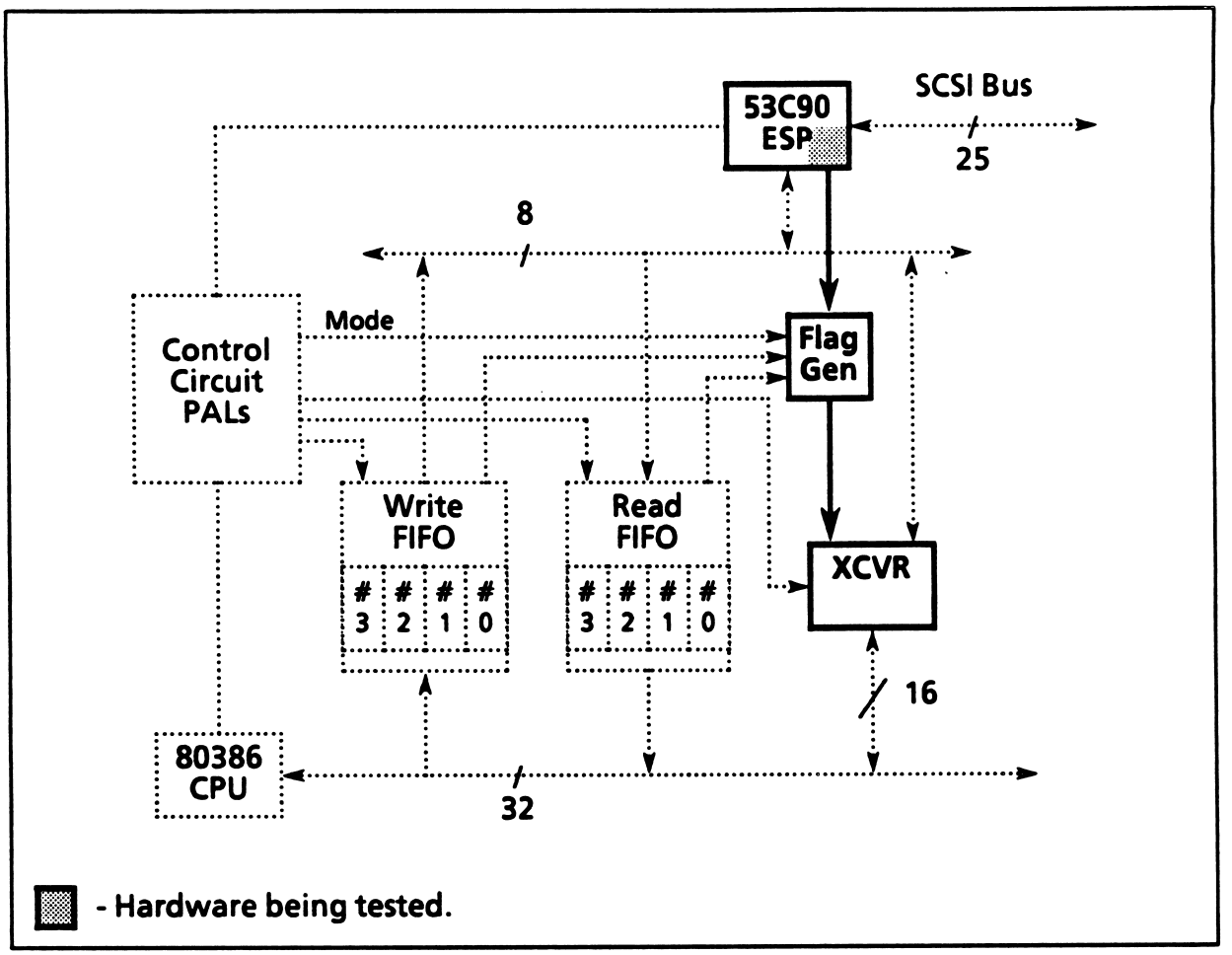


Figure 3-33. Hardware Tested by the SCSI Module: ESP Interrupt Test

HARD DISK TESTS

These tests check the capability of the hard disk to store and retrieve data. The Hard Disk Tests are listed in Table 3-13. Figure 3-34 shows the hard disk subsystem diagnostic hardware model. This model is used as an example for hardware tested by the Hard Disk Tests. The Hard Disk Tests also include options and utilities, which are described in later sections entitled "Hard Disk Test Options", "Basic Drive Utilities", and "Advanced Drive Utilities".

Table 3-13. Hard Disk Tests

Test Name
SCSI Bus Reset Test
Drive Ready Test
Drive ID Test
Buffer Memory Write/Read Test
Media Read Test
Media Write/Read Test
Worst Case Seek Test
Surface Analysis Test [Extended]

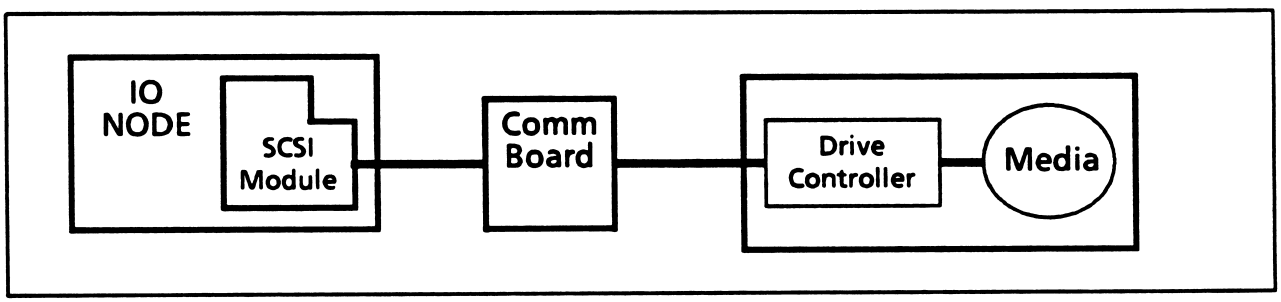


Figure 3-34. The Hard Disk Subsystem Diagnostic Hardware Model

Hard Disk: SCSI Bus Reset Test

This test issues a SCSI bus reset. Thus, all connected drives will reset and execute the associated startup sequence. The sequence will light the green LED located on the face plate of each drive for about two seconds. Therefore, by running this test and watching the green LED on each drive, you can determine if the drive is cabled properly to the SCSI module. Hardware tested is shown in Figure 3-35.

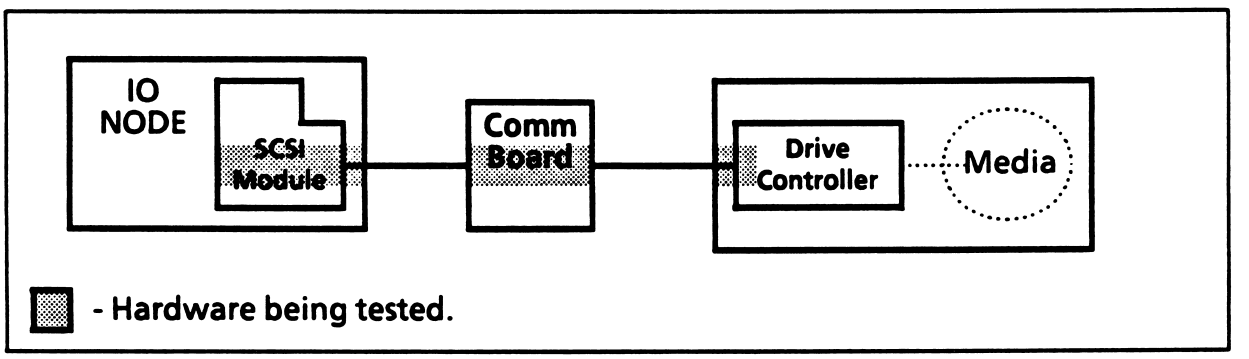


Figure 3-35. Hardware Tested by the Hard Disk: SCSI Bus Reset Test

Hard Disk: Drive Ready Test

This test requests that each drive (specified in the */usr/ipsc/conf/devconf* file) respond with a drive ready (see Figure 3-36).

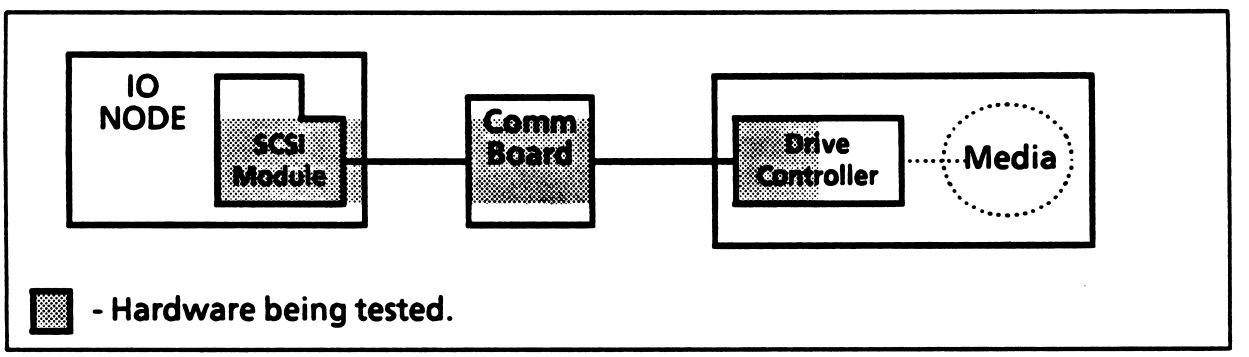


Figure 3-36. Hardware Tested by the Hard Disk: Drive Ready & Drive ID Test

Hard Disk: Drive ID Test

This test collects identification information from each of the drives and compares it with the information found in the */usr/ipsc/conf/devconf* file. A miscomparison of ID between the drive and */usr/ipsc/conf/devconf* file is reported as a failure and displays the expected and actual ID information. Hardware tested is shown in Figure 3-36.

Hard Disk: Buffer Memory Write/Read Test

This test fills the drive controllers buffer memory with a test pattern of 0xAA, reads the buffer memory, and compares data written with data read. Miscompares are displayed with expected and actual data. Hardware tested is shown in Figure 3-37.

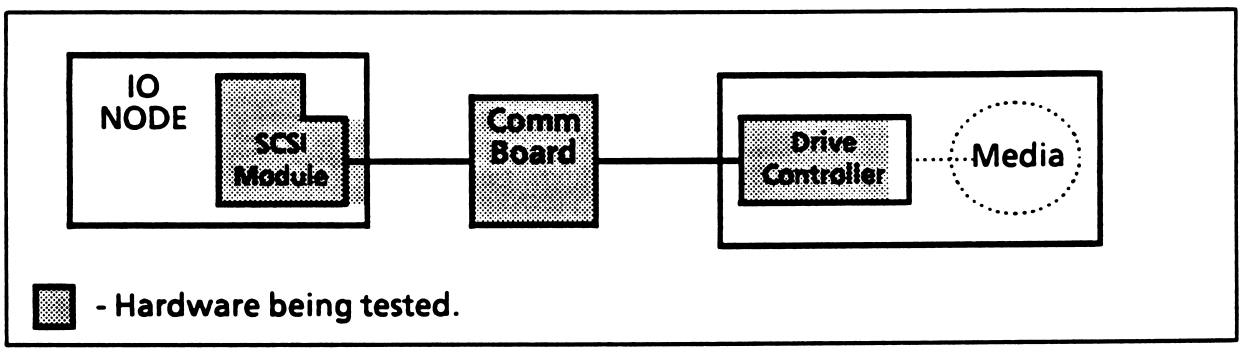


Figure 3-37. Hardware Tested by the Hard Disk: Buffer Memory Write/Read Test

Hard Disk: Media Read Test

This test reads all blocks of data (all sectors on all tracks) and verifies that all blocks have valid checksums. The data are not verified. Hardware tested is shown in Figure 3-38.

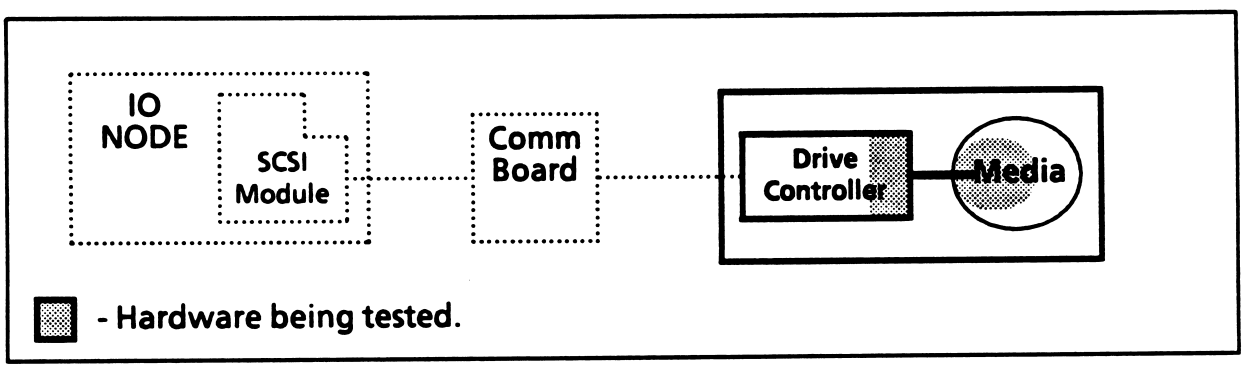


Figure 3-38. Hardware Tested by the Hard Disk: Media Read Test

Hard Disk: Media Write/Read Test

This test writes, reads, and verifies data on the diagnostic blocks. (Diagnostic blocks are blocks reserved from the node operating system (NX) for this purpose.) The blocks are physically located near the beginning and the end of the disk drive storage space. The data written to the blocks are made unique so that the drive is checked for addressing as well as data retention. Hardware tested is shown in Figure 3-39.

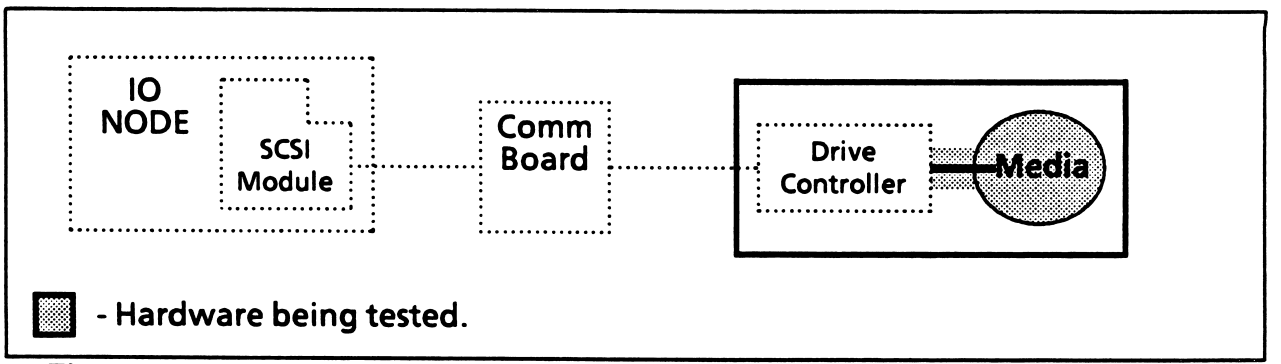


Figure 3-39. Hardware Tested by the Hard Disk: Media Write/Read, Worst Case Seek, & Surface Analysis Tests

Hard Disk: Worst Case Seek Test

This test runs the head assembly through three worst-case seek patterns. Pattern 1 proceeds from block 0 to block 1, 0 to 2, etc., to the last block. Pattern 2 proceeds from last to last -1, last to last -2, etc., to the beginning of the drive. Pattern 3 proceeds from middle -1, middle +1, middle -2, middle +2, etc., to the start and end of the drive. Hardware tested is shown in Figure 3-39.

Hard Disk: Surface Analysis Test (Extended)

CAUTION

This test will destroy all hard disk data.

You must select a specific node to be under test before this test will execute. Select a specific node to be under test from the options/configuration/modify menu by placing a valid node number into *test_node*. Normally, *test_node* is set to -1, which selects all nodes to be under test.

This test writes and then reads and verifies a unique data pattern for all blocks. Several options are available for this test (see the next section, entitled "Hard Disk Test Options"). Hardware tested is shown in Figure 3-39.

HARD DISK TEST OPTIONS

Options available for hard disk testing are described in the following paragraphs. Options are available while in the prompt mode. Prompt mode is selected when *test_node* equals a valid node ID.

SCSI Module: Pattern Sensitivity Test

This test allows the user to specify and test a particular FIFO. Any or all FIFOs may be tested. Each FIFO test requires approximately one hour.

Hard Disk: Buffer Memory Write/Read Test

This test requests the following:

a. buffer address offset
b. number of bytes
c. pattern

Hard Disk: Media Read Test

This test requests the following:

a. starting block
b. ending block
c. number of blocks per read

Hard Disk: Media Write/Read Test

This test requests the following:

a. starting block
b. ending block
c. pattern size (char, short, or integer)
d. pattern
e. verify only

Hard Disk: Worst Case Seek Test

This test requests the following:

a. starting block
b. ending block
c. blocks per pass

Hard Disk: Surface Analysis Test

This test requests the following:

a. starting block
b. ending block
c. pattern size (char, short, or integer)
d. pattern
e. time (in seconds) to pause before read
f. number of reads after write
g. number of blocks per operation
h. block size
i. verify only

BASIC DRIVE UTILITIES

Basic Drive Utilities are most useful for a technical user. These utilities are listed in Table 3-14. The menu contains the Controller Reset, SCSI Bus Reset, and Drive Ready Tests as well as utility functions. The utilities must be run in the prompt mode, which is selected when *test_node* equals a valid node ID. The utilities are described in the following paragraphs.

Basic Drive: Controller Reset Utility

This utility resets the SCSI Module and re-initializes the SCSI processor.

Basic Drive: SCSI Bus Reset Utility

This utility issues a SCSI bus reset. Thus, all connected drives will reset and execute the associated startup sequence. The sequence will light the green LED located on the face plate of each drive for about two seconds. Therefore, by running this utility and watching the green LED on each drive, you can determine if the drive is cabled properly to the SCSI module.

Table 3-14. Basic Drive Utilities [Extended]

Utility Name
Controller Reset
SCSI Bus Reset
Scan Device
Drive Ready
Request Sense
Drive Inquiry
Read Capacity
Seek
Read Buffer
Write Buffer
Read Block
WriteBlock
Mode Sense
Fill Data
Dump Data
Change Target
Add/Delete Target
Display Stats.
Change Op. Mode

Basic Drive: Scan Devices Utility (Extended)

This test scans the disk drive(s) to be tested and displays information about the subject drive(s). You must select a specific node to be under test before this test will execute. Select a specific node to be under test from the options/configuration/modify menu by placing a valid node number into the *test_node* parameter. Normally, the *test_node* parameter is set to -1, which selects all nodes to be under test.

The following information is displayed about the disk drive(s) attached to the selected node:

```
dev  type  qual  rmb  vers  fmt  reserved  Identification
```

These tests are useful in determining the drive ID numbers jumpered on the connected drives. Refer to the Maxtor Disk Manual for details.

Basic Drive: Drive Ready Utility

This utility requests that each drive (specified in the */usr/ipsc/conf/devconf* file) respond.

Basic Drive: Request Sense Utility

This utility issues a request sense command to the drive and displays the sense information returned.

```
sense info:
dev  class   seg  key info  code   FRU   bit   field
```

Basic Drive: Drive Inquiry Utility

This utility issues an inquire command to the drive and displays the results from the drive in the following format:

```
dev  type  qual  rmb  vers  fmt  reserved  Identification
```

The *dev* field indicates the drive ID and the rest of the fields are per the Maxtor Disk Manual.

Basic Drive: Read Capacity Utility

This utility issues a read capacity command to the drive and displays the results from the drive in the following format:

```
dev  highest  block number  block size  total bytes
```

Basic Drive: Seek Utility

This utility seeks the drive to the block entered.

Basic Drive: Read Buffer Utility

This utility is used to read the drive controller buffer RAM. A prompt is issued for starting address offset and the number of bytes to read. The data returned by the drive may be dumped with the Dump Data Utility.

Basic Drive: Write Buffer Utility

This utility is used to write the drive controller buffer RAM. A prompt is issued for starting address offset and number of bytes to write. The user should fill the write buffer with the Fill Data Utility first to preset the data.

Basic Drive: Read Block Utility

This utility is used to read one or more blocks of data from the drive. The user can specify the starting block number and the number of blocks. The data returned by the drive may be dumped with the Dump Data Utility.

Basic Drive: Write Block Utility

This utility is used to write one or more blocks of data to the drive. The user can specify the starting block number and the number of blocks. The user should fill the write buffer with the Fill Data Utility first to preset the data.

Basic Drive: Mode Sense Utility

This utility issues a mode sense command to the drive and displays the results in raw form with each page code data displayed on separate lines. Some of the more useful fields are labeled. The user is prompted for the desired page control, the page code, and whether the utility should get new sense data or display the currently held data. The first section of data merely summarizes the detailed information in the following format:

```
dev  cylinders  heads  bytes/block
```

This is followed by the complete sense information for each drive.

Basic Drive: Fill Data Utility

This utility is used to preset the data in the write data buffer. The user is prompted for the pattern size (units), the pattern, and the number of units to fill.

Basic Drive: Dump Data Utility

This utility is used to display the drive read data buffer. The user is prompted for a starting offset and for the number of bytes to display.

Basic Drive: Change Target Utility

This utility is used to change the drive that is used by the tests and utilities. The user can specify drives 0 thru 6, or all drives with 0xFF.

Basic Drive: Add / Delete Target Utility

This utility is used to add a drive to or delete a drive from the test suite. The addition or deletion is not permanent (*/usr/ipsc/conf/devconf* file is not changed). The user is prompted for the drive ID to add or delete the drive and, if an add operation is selected, the type of drive to add.

Basic Drive: Display Stats Utility

This utility is used to display detailed technical information about the drive states and a trace of the interrupt sequences. It is not very useful for the casual user, but is of great use for debugging the tests and utilities. The basic information shows data transferred and disconnect counts for each drive, along with the last drive status. The interrupt trace information shows the tag of the calling routine, the SCSI and interrupt status, the SCSI message, the drive ID, and the status of the driver (*s_status*) for each pass through the interrupt handler.

Basic Drive: Change Operations Mode Utility

This utility forces the SCSI driver to not let the drive disconnect during operations. The user is prompted for the drive ID and whether the driver should hold or allow disconnect.

ADVANCED DRIVE UTILITIES

Advanced Drive Utilities are most useful for a technical user. These utilities are listed in Table 3-15. The menu contains the Controller Reset, SCSI Bus Reset, and Drive Ready Tests, plus the following Basic Drive Utilities: Request Sense, Drive Inquiry, Mode Sense, and Change Target. The utilities must be run in the prompt mode. Prompt mode is selected when *test_node* equals a valid node ID.

Table 3-15. Advanced Drive Utilities [Extended]

Utility Name
Controller Reset
SCSI Bus Reset
Drive Ready
Request Sense
Drive Inquiry
Mode Sense
Mode Select
Format
Display Bad Block List
Reassign Bad Block
Change Target

The following paragraphs describe the Advanced Drive Utilities.

Advanced Drive: Controller Reset Utility

This utility is a Basic Drive Utility that has been included in the Advanced Drive Utilities Menu for user convenience. Refer to the Basic Drive Utilities section for more detailed information.

Advanced Drive: SCSI Bus Reset Utility

This utility is a Basic Drive Utility that has been included in the Advanced Drive Utilities Menu for user convenience. Refer to the Basic Drive Utilities section for more detailed information.

Advanced Drive: Drive Ready Utility

This utility is a Basic Drive Utility that has been included in the Advanced Drive Utilities Menu for user convenience. Refer to the Basic Drive Utilities section for more detailed information.

Advanced Drive: Request Sense Utility

This utility is a Basic Drive Utility that has been included in the Advanced Drive Utilities Menu for user convenience. Refer to the Basic Drive Utilities section for more detailed information.

Advanced Drive: Drive Inquiry Utility

This utility is a Basic Drive Utility that has been included in the Advanced Drive Utilities Menu for user convenience. Refer to the Basic Drive Utilities section for more detailed information.

Advanced Drive: Mode Sense Utility

This utility is a Basic Drive Utility that has been included in the Advanced Drive Utilities Menu for user convenience. Refer to the Basic Drive Utilities section for more detailed information.

Advanced Drive: Mode Select Utility

This utility is used to set the different pages of mode information within the drive. Each of the four pages is set to separate operations. For a complete description of each page contents, see the Maxtor Disk Manual.

The drive mode information must be set up before the drive is formatted. A typical application would be to use the Mode Select Utility to set up the number of alternate sectors per zone, the number of alternate tracks per drive, etc., and then use the Format Utility. A zone is a logical area on the drive, either one cylinder or number-of-heads cylinders. This utility uses number-of-heads. For currently available drives, there are 15 heads.

Advanced Drive: Format Utility

This utility is used to format an entire disk. It is assumed that the user has previously used the Mode Select Utility to set up the drive configurable parameters. The user is prompted for the interleave, the format data pattern, 'certify' on or off, 'stop format on error' yes or no, and whether the existing grown defect list of the drive should be left intact or purged. The defaults are interleave of 1, a data pattern of 0xE5, no certify, stop on error, and to leave the current grown defect list enabled.

Advanced Drive: Display Bad Block List Utility

This utility is used to display the bad block list(s) of the drive. The user is prompted for the list type and the format of the list. The default is the grown defect list, in bytes from index format.

Advanced Drive: Reassign Bad Block Utility

This utility is used to add a bad block to the grown defect list of the drive. The only user parameter is a block number that is double checked with the user before adding it to the list.

The only way to delete a bad block from the defect list is to reformat the drive, using the 'purge existing grown defect list' option.

Advanced Drive: Change Target Utility

This utility is used to change the drive that is used by the tests and utilities. The user can specify drives 0 through 6, or all drives using 0xFF

Bus Interface Adapter Tests

The Bus Interface Adapter (BIA) Tests check the node-to-VME Bus Interface Adapter module in a standalone environment. The BIA should be capable of passing these tests regardless of the presence or absense of a VME board. These tests offer approximately a 90-percent confidence level that the BIA is fully functional. The tests include the following:

Register Test
VME Bus Swap Test
Interrupt Test
LED Test

BIA: REGISTER TEST

This test uses a BIA memory mapped register that is writable and readable. The register is filled with a walking 1, then walking 0 data patterns, and then verified.

BIA: VME BUS SWAP TEST

This test uses a diagnostic loopback capability built into the BIA. The loopback captures data at the VME side of the BIA and allows the node to then read the data back. Thus, the loopback is used to stimulate the byte swap circuitry through all combinations of swapping.

BIA: INTERRUPT TEST

This test causes the BIA to generate an interrupt and then verifies that the interrupt is sensed by the node. The BIA is then forced to release the interrupt and the node verifies that the interrupt request disappears. The interrupt is tested via an exception handler and thus tests the physical interrupt line on the PBX interface between the BIA and node.

BIA: LED TEST

This test blinks each of the LEDs in a round-robin fashion. You must actually watch the LEDs to verify the operation.

HARDWARE TEST STATES

CDP records the state of the Cube hardware. If the hardware is in the required state at the time that the test is initiated, the test immediately executes. Thus, the key to getting the most performance from CDP is to minimize running tests sequences that change states. The states are shown in Figure 3-40 and Table 3-16 lists each of the tests and the hardware state required by each test.

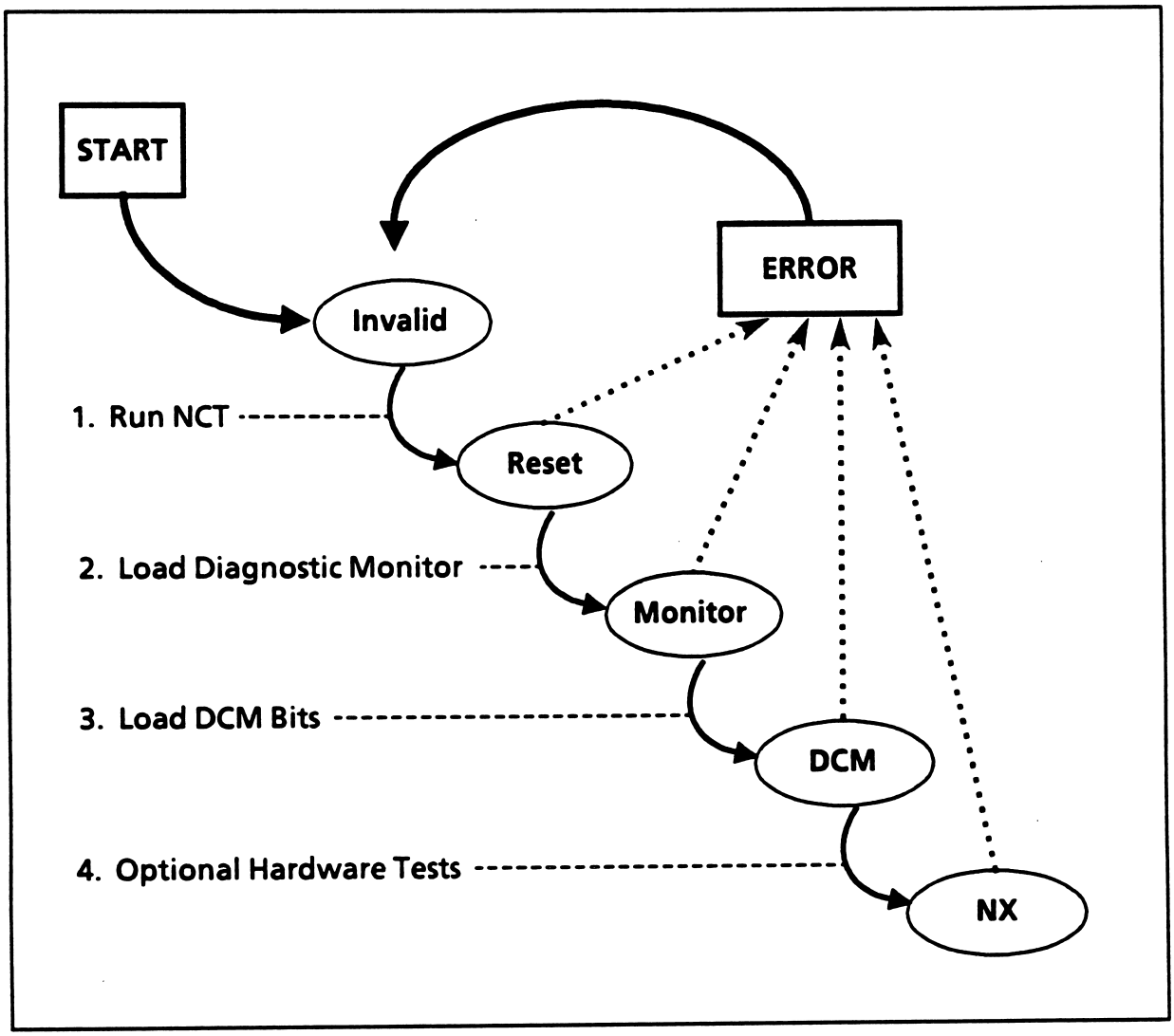


Figure 3-40. Cube States

Table 3-16. Tests Versus Hardware Test States

Test(s)	State
Diag Link: all tests except Node Echo Test	Any
Diag Link: Node Echo Test	Reset
Node Standalone: Node Confidence Test	Forces Reset
Node Standalone: RAM Size Test, Node Standalone: RAM Data Lines Test, and Node Standalone: RAM Address Lines Test	Monitor
Node Standalone: Compute Node DCM Loopback Test, Checksum Test	DCM
Host Link: SRM FIFO Loopback Test and Host Link Tests: Cable & Node Backplane Loopback Test [Extended]	Any
Host Link: Host/Node 0 Channel 7 Router Test, Host Link: Node 0 Receive Test, Host Link: Node 0 Transmit Test, and Host Link: DCM Checksum Test	DCM
Node Link: All Tests	DCM
Cube Link: All Tests	DCM
Generic Link: All Tests	DCM
Optional Hardware Tests	NX

POWER UP TESTS A

INTRODUCTION

This appendix describes the iPSC/2 Power Up Tests contained in the SRM and on the USM and node boards.

There are three separate power up tests in the iPSC/2 system. These are:

- Power On Self Test (POST) for the SRM.
- USM Confidence Test for the USM.
- Node Confidence Test (NCT) for each of the node boards.

POWER ON SELF TEST

The Power On Self Test (POST) checks the basic functions of the SRM . This test provides confidence that the SRM is capable of booting UNIX. Additional details about the POST are contained in the *iSBC® 386 AT User's Guide*.

USM CONFIDENCE TEST

The USM Confidence Test checks the internal operation of the USM. After the test passes, the reset line to the node boards is pulsed. The test checks the USM firmware checksum and timers 0 and 1. A failure maintains the red LED on and blinks the green LED six times to indicate a firmware checksum fault, seven times for a timer 0 fault, and eight times for a timer 1 fault.

The USM firmware also indicates Diagnostic Channel communication errors via its LEDs. See Appendix D for details.

NODE CONFIDENCE TEST

The following paragraphs provide a detailed technical description of the 386 Node Confidence Test (NCT). The paragraphs are divided as follows: First, the objectives of the NCT are described. This includes a brief overview. Second, the error indications are described. Third, the NCT Main Module is described along with its logical sequence of utilizing the NCT subtests. Fourth, each of the subtests is described in detail. Fifth, special features of the NCT that aid in component level debug are described. Last, the NCT memory map is provided.

NCT Objectives and Overview

The primary objective of the NCT is to verify the 386 node board hardware and provide a pass or fail indication to isolate a faulty node board. The second objective is to provide subsystem pass or fail status values that can be used to generate diagnostic information for use in manufacturing tests. The first objective is met by simply condensing the subsystem test status values into a single pass or fail result.

The NCT executes in real address mode and performs an ordered sequence of tests on power up or reset in less than seven seconds. The NCT exercises over 80 percent of the hardware on the 386 node board.

The NCT employs an inward/out testing methodology for fault isolation. Using such methods, subtests are executed in a fixed sequence so that the processor exercises devices from the inside out, testing the core hardware (hardcore) first, then the peripheral components as those are required. The 386™ node board hardcore consists of the microprocessor, the EPROM, and the first 16 kb of DRAM.

NCT Error Indications

The NCT indicates errors using four methods:

- LED Indications
- Diagnostic Light Pen
- Hardware Trace Register
- Status Table

Refer to the following paragraphs for details on these methods.

LED INDICATIONS

The NCT utilizes the node board red and green LEDs to indicate either error or pass as listed in Table A-1.

Table A-1. NCT LED Indications

Red LED	Green LED	Description
Off	Off	The 386 node board is not receiving power, a problem exists with the 80386 processor or its ability to correctly fetch and execute instructions from EPROM, or an interrupt was received before the interrupt vector table was initialized.
On	On	A problem exists in either the 80386, the EPROMs, or the first 16k bytes of DRAM. The contents of the Hardware Trace Register may be valid, containing the subtest number that failed. The red LED is flashing an error message at 1200 baud, readable via a diagnostic light pen and a terminal.
On	Off	A problem exists in one of the peripheral components or RAM. The contents of the Hardware Trace Register are valid, containing the subtest number that failed. The Node Boot Monitor is active if the failing test is not a hardcore failure. The red LED is flashing an error message at 1200 baud, readable via a diagnostic light pen and a terminal.
Off	On	The NCT passed and the Node Boot Monitor is active.
Blinking	Off	When the red LED flashes at approximately two hertz, this indicates that the 80386 processor received a spurious interrupt on its INTR or NMI inputs or trapped an internal exception.

DIAGNOSTIC LIGHT PEN

An error message is transmitted via the node board red LED. This is accomplished by turning the LED on and off at a 1200-baud data rate. To the naked eye, the red LED may appear to be on, but it is actually blinking rapidly. When the LED is on, it denotes a one bit; when off, it denotes a zero bit. Through the use of a Diagnostic Light Pen, these pulses of light are converted into an RS-232 data signal and displayed on a standard 1200-baud terminal. Simply hold the Diagnostic Light Pen near the red LED and read the error message on the terminal screen.

The Diagnostic Light Pen consists of a photo-transistor and a simple two-stage amplifier to boost the signal level enough to drive the terminal. Data are sent to the terminal Pin 3. Power for the circuit is drawn from the terminal data terminal ready (DTR) line on Pin 20, and ground on Pin 7, which eliminates the requirement for an external power supply. A schematic diagram of the Diagnostic Light Pen is shown in Figure A-1.

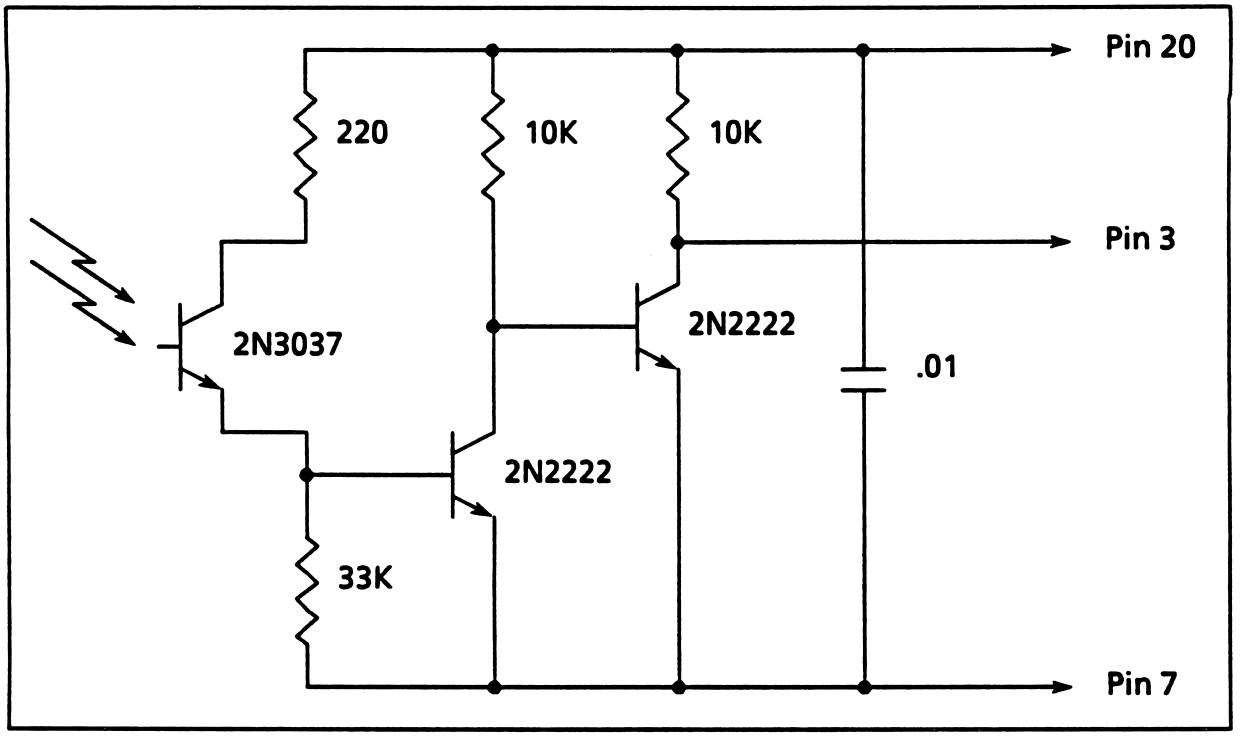


Figure A-1. Diagnostic Light Pen Schematic

HARDWARE TRACE REGISTER

A Hardware Trace Register is provided on the 386 node board for additional diagnostic information. The contents of this register are valid only if the red LED is on and the green LED is off. The trace register is five bits wide and is implemented in the least significant bits of On-Board Control Register #2. The pinout of the register is shown in Figure A-2.

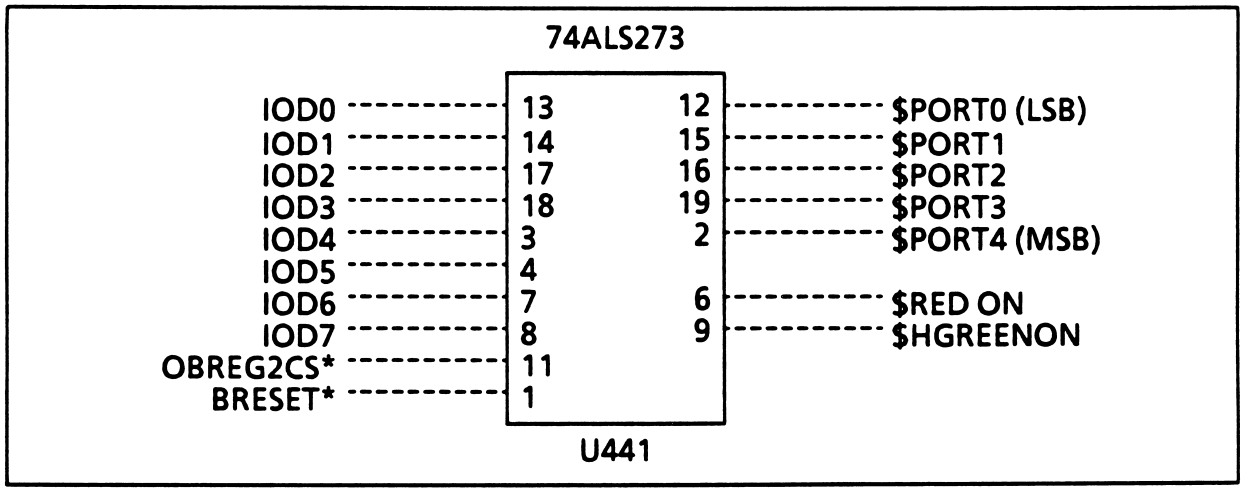


Figure A-2. Hardware Trace Register

The $PORTx lines from the Hardware Trace Register are made available on the backplane by routing those lines through the DCM to the P1 connector on the node board. The mapping of the $PORTx bits to the node board P1 connector and the backplanes P1 connector are listed in Table A-2.

Table A-2. Trace Register P1 Connector

Trace Register Bit	Node Board P1	Backplane P1
$PORT0	A14	14
$PORT1	B15	47
$PORT2	C4	68
$PORT3	C10	74
$PORT4	A18	18

A number associated with each subtest is written to the Hardware Trace Register as the subtest executes. Thus, if a subtest fails or the board hangs, the number of the subtest detecting the failure remains in the register as long as power is applied to the board. The numbers associated with the subtests and placed into the Hardware Trace Register are listed in Table A-3.

Table A-3. Subtests vs. Trace Register Contents

Subtest Name	Hex Value
EPROM Checksum Test	11
0-16K RAM Test	12
Test 0	0
80386 Processor Test	1
82510 UART Test	2
82258 ADMA Test	3
8259A Master PIC Test	4
8259A Slave PIC Test	5
80387 NPX Test	6
Parity Test	7
16K-1023K RAM Test	8
Cache Test	9

STATUS TABLE

The Status Table is a collection of RAM-resident data containing the NCT pass/fail result and control variables used during ICE-386 debug. The Status Table is listed in Table A-4. A more complete description of the Status Table is in the section of this chapter entitled "Special Features of the NCT".

Table A-4. Status Table Contents

#	Variable Name	Size
1	*node_id*	WORD
2	*cube_dim*	WORD
3	*mem_size*	WORD
4	*NCT_result*	WORD
5	*NCT_version*	WORD
6	*mode*	WORD
7	*ice_debug*	WORD
8	*test_index*	WORD
9	*halt_on_error*	WORD
10	*iteration_count*	DWORD
11	*failure_count*	DWORD
12	*execution_count*	DWORD
13	*cache_failure*	WORD
14	*mem_test_pattern*	DWORD
15	*start_offset*	WORD
16	*start_segment*	WORD
17	*end_offset*	WORD
18	*end_segment*	WORD

Immediately following the Status Table is a 512 byte Message Buffer. The contents of this buffer are valid for all tests except the EPROM Checksum and 0 - 16k RAM Tests. The buffer contains the null terminated ASCII string that is transmitted via the red LED for the Diagnostic Light Pen.

Immediately following the Message Buffer is a RAM Failure Table. The contents of this table are valid for the RAM Subtest, and are listed in Table A-5. A more complete description of the RAM Failure Table is in the section of this chapter entitled "Special Features of the NCT".

Table A-5. RAM Failure Table Contents

#	Variable Name	Size
1	*parity_error*	WORD
2	*failure_offset*	WORD
3	*failure_segment*	WORD
4	*expected_pattern*	DWORD
5	*actual_pattern*	DWORD
6	*XOR_pattern*	DWORD

NCT Main Module

Because it is the first to execute, the Main Module controls the NCT overall testing algorithm. The node board hardware, by default, starts after powerup or reset as follows:

- Red and Green LEDs are off.
- The USM Interrupt is asserted.
- The 386 microprocessor is initialized to real (8086) addressing mode.
- The 386 microprocessor fetches its first instruction from near the top of physical memory, at location F000:FFF0FFF0H.

NOTE

It is important to note that, for Code Segment (CS)-relative memory cycles, address lines A20-A31 are held high. When the first intersegment jump or call is executed, address lines A20-A3 drop low for CS- relative memory cycles, and the 386 will execute only instructions in the lower one megabyte of physical memory. A great deal of care has been taken to ensure that no far jumps or calls are executed by the NCT that would change the CS. If an intersegment jump or call is executed (perhaps by a spurious interrupt or invalid opcode fetch) the EPROMs are essentially lost.

The algorithm of the Main Module is shown in Figure A-3.

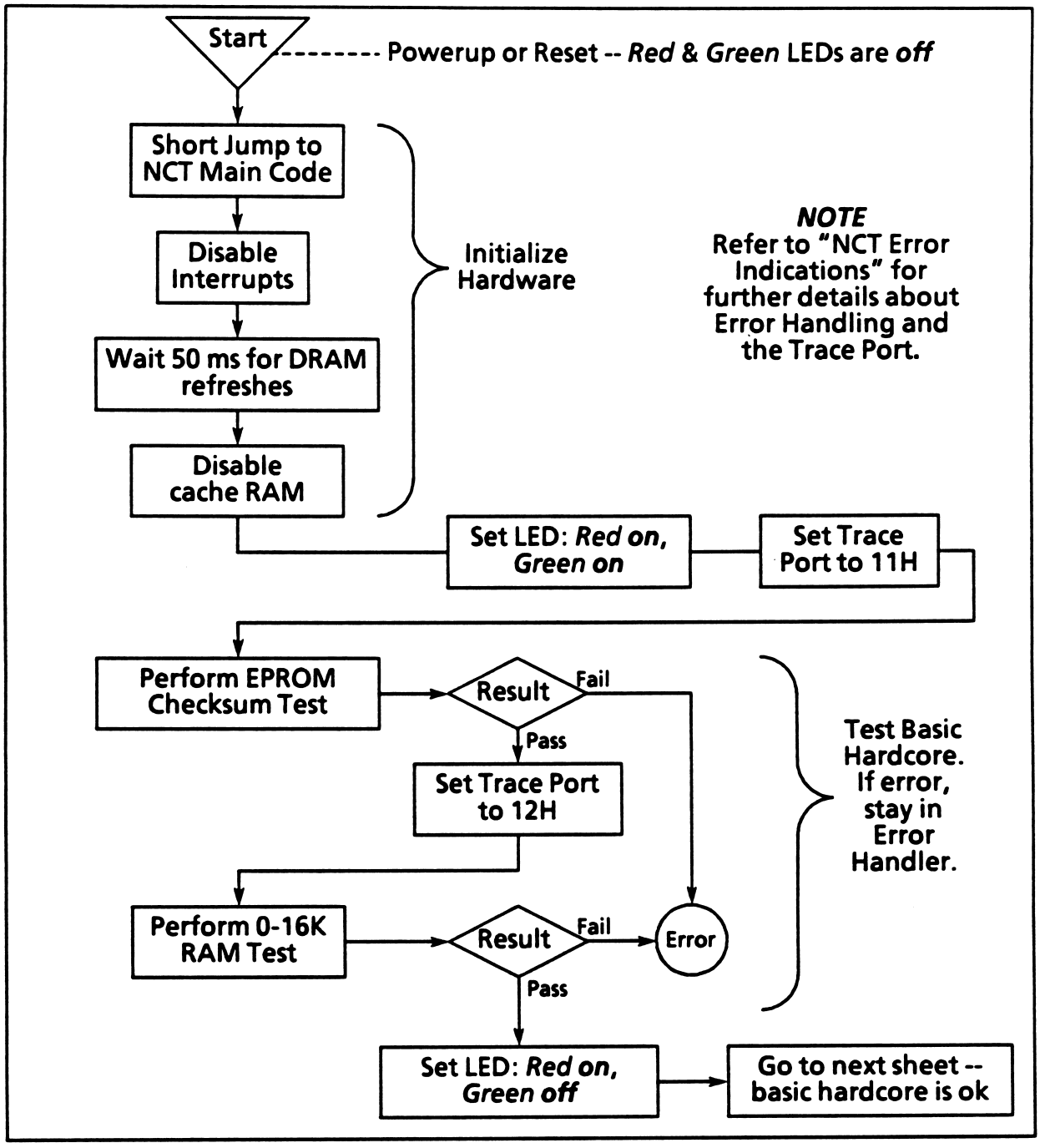


Figure A-3. NCT Main Module Flowchart (*Sheet 1 of 3*)

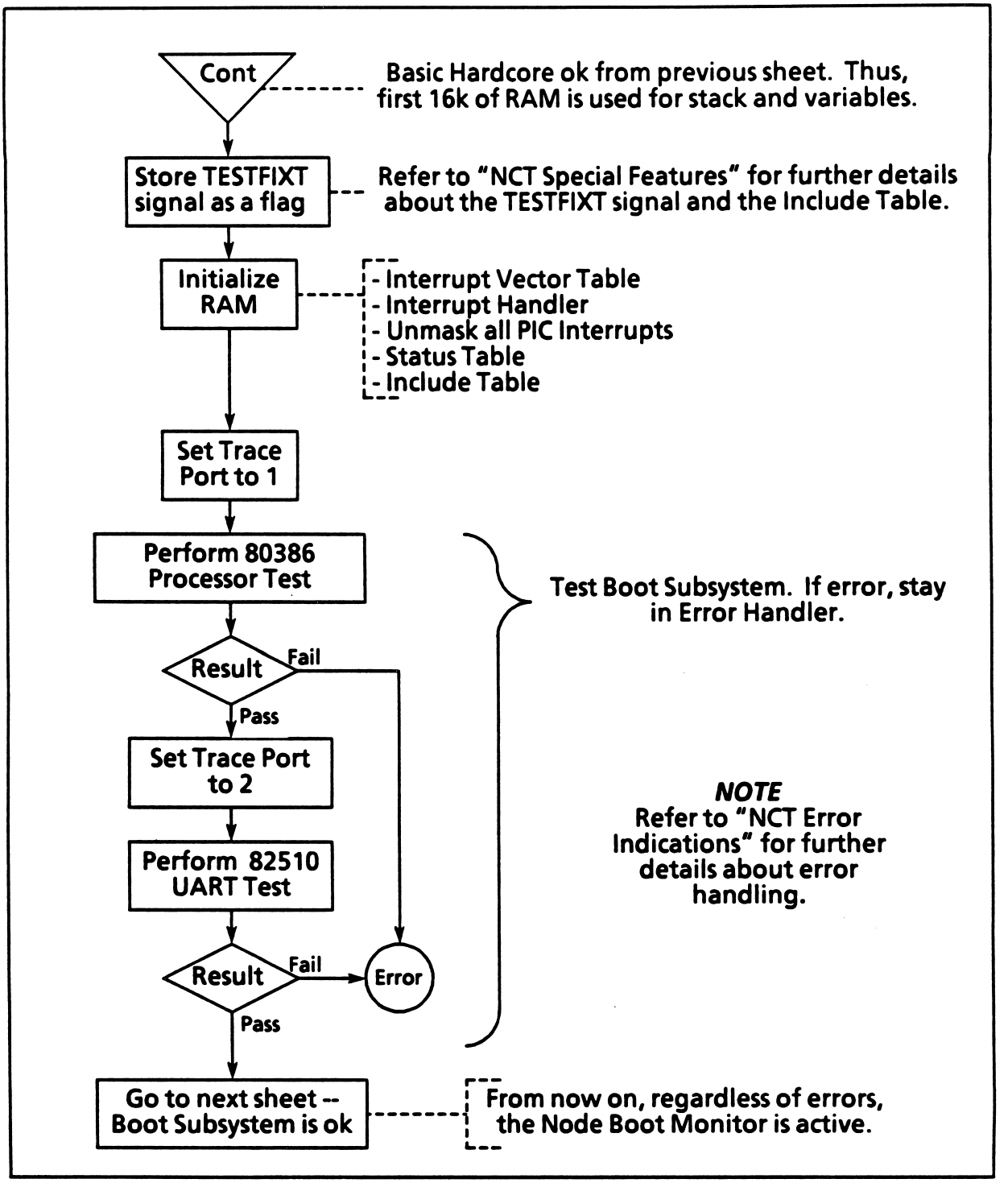


Figure A-3. NCT Main Module Flowchart (*Sheet 2 of 3*)

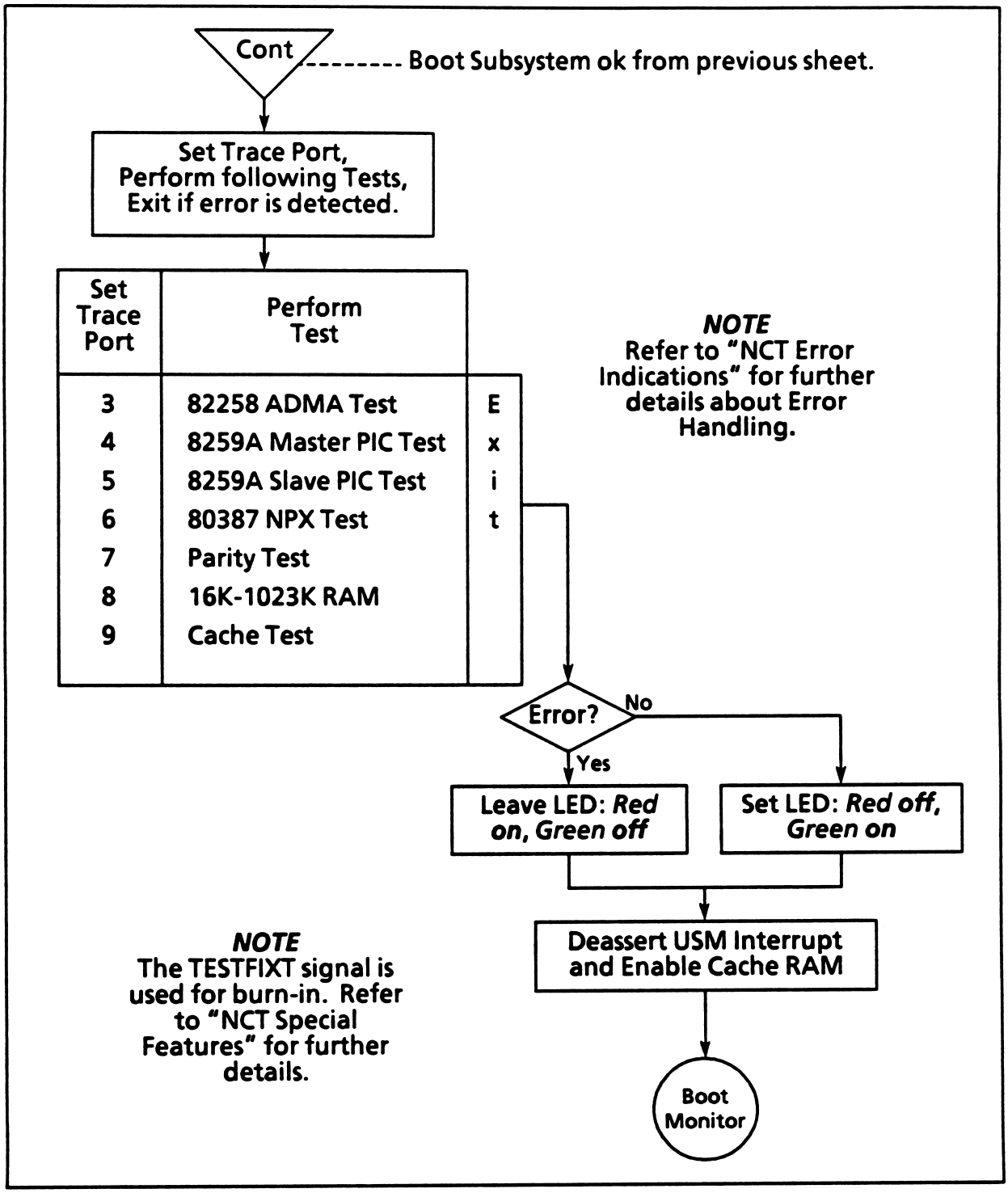


Figure A-3. NCT Main Module Flowchart (*Sheet 3 of 3*)

NCT Subtest Detailed Description

The NCT includes the following 11 subtests:

EPROM Checksum
0 - 16K RAM
386 Microprocessor
82510 UART
82258 ADMA
8259A Master PIC
8259A Slave PIC
80387 NPX
Parity
16 - 1023K RAM
Cache RAM

The following paragraphs provide an overview of the NCT and a detailed description of each of the subtests.

NCT OVERVIEW

The NCT is composed of several subtests. Each checks a specific portion of the 386 node board. Each subtest is written so that it has only one entry point and one exit point. That is, when the subtest is invoked from the mainline code, it begins executing at a particular point, and, when the subtest completes, it returns to the mainline code from a particular point. This technique is used for easy code development and debug.

With the exception of the EPROM Checksum Test and the 0 - 16K RAM Test, when each subtest completes its execution, it calls an NCT Utility (*set_ax*) that sets the AX register of the 386 microprocessor with the subtest pass/fail result. A pass is indicated by FFFFH and a fail is indicated by 0000H. This routine is convenient during code development as an ICE-386 breakpoint. When the ICE-386 performs the break, the AX register is simply examined to determine the pass/fail status.

EPROM CHECKSUM SUBTEST

This subtest computes the simple checksums of the two EPROMs containing the NCT and the Node Boot Monitor.

This subtest is invoked by a GOTO from the mainline; calls and variables cannot be used until the 0 - 16K RAM Subtest is run. The checksums of the two EPROMs are computed. The checksums are compared to the stored checksums in the last two words of the EPROMs to verify the EPROM contents.

A structure containing the actual and expected checksums is first initialized. Next, using 32-bit accesses, the even and odd bytes from each EPROM, respectively, are summed into a word variable forming the calculated checksums. Any overflow is discarded. The 32-bit accesses are accomplished in ASM 86 by using a define byte (DB) assembler directive and placing an operand size prefix in front of the instructions that require 32 bits. The two's complement of each calculated checksum is then determined to match the iUP201 PROM Programmer checksum algorithm. Finally, the calculated checksums are compared with the expected checksum values.

If the checksums match, a near jump is performed to an entry point in the mainline code. If the checksums do not match, the test fails and the 386 microprocessor enters a loop that continuously transmits an error message via the red LED. The processor enters this loop because no further testing can be done if the node board EPROMs cannot be accessed reliably.

0 - 16K RAM SUBTEST

This subtest verifies the first 16K bytes of DRAM, providing an environment for the NCT to run while testing the peripheral chips.

The algorithm used by the RAM subtest is referred to as a verify/complement/verify test. In this method, the RAM to be tested is filled with an initial background pattern. Then, for each location, the following loop is executed:

> Read the location and verify that the old pattern is still present. Complement the pattern in the location. Read the location and verify that the complement pattern is still present.

Using this method, both address and data lines are tested in addition to the DRAM devices.

This subtest is invoked by a GOTO from the mainline; calls and variables cannot be used until this subtest has passed. The memory pattern used during testing is A5A5H and its complement. The memory test pattern is placed in the AX register and its complement in BX. The ES:DI register combination points to the location under test. The count (2000H words) is placed in the CX register and the memory is filled using a REP STOSW instruction.

After the background pattern is written, the verify/complement/verify loop is entered. The DI register is cleared and the count is again placed in the CX register. Each location is then compared with the value in the AX register, complemented, and compared with the value in the BX register. Then the DI register is incremented to point to the next location as the loop continues.

If all locations pass, a near jump is performed to an entry point in the mainline code. If any location does not compare correctly with the contents of the AX or BX registers, the test fails and the 386 microprocessor enters a loop that continuously transmits an error message via the red LED. The processor enters this loop because no further testing can be done if the first 16K bytes of DRAM cannot be used reliably for data and stack.

386™ MICROPROCESSOR SUBTEST

This subtest verifies the 386 microprocessor internal register addressing, register data retention and stuck bits, logical and mathematical circuitry, conditional flags and jumps, and memory addressing methods.

This subtest is executed in five parts. The purpose for each part and the method used by each part are described in the following paragraphs:

Register Addressing Test -- Verifies the register addressing by writing a unique pattern to each register while verifing that none was overwritten by a pattern intended for another register.

Register Data Test -- Checks each register for stuck bits by writing and verifying complementary patterns (A5A5H and 5A5AH).

Register Logical And Mathematical Operators Test -- Tests the logic operators NOT, AND, OR, TEST, XOR, SHIFT, and ROTATE, and the mathematical operators ADD, ADC, INC, SUB, SBB, DEC, NEG, CMP, MUL, and DIV. This is accomplished by loading various registers, performing one of the above operations, and verifing the results. Verification is performed by comparing the registers and appropriate flags with expected values.

Registers Conditional Jumps Test -- Tests each of the conditional jumps. Tested combinations include ZERO FLAG, CARRY FLAG, OVERFLOW FLAG, SIGN FLAG, and PARITY FLAG. Flags are set up so that each conditional jump executes only once.

Addressing Modes Test -- Indirect jumps are performed based on various registers. Each block of code sets up a jump and increments the AX register. If the jumps operate correctly, AX contains the correct number at the end of the subtest. Otherwise this subtest fails by either:

1. Jumping to the wrong label, resulting in an invalid count, or
2. Detecting an illegal op-code fetch.

The 386 Addressing Modes portion of the 386 Microprocessor Subtest is performed in three parts. Each part tests a different level of indirection. Indirect jumps are accomplished by indexing into a jump table that contains CS- relative offsets to labels within the subtest. The parts are as follows:

Jumps Using Register Targets -- A register is loaded with the offset of the next target from the jump table. A jump is then performed with the register contents as the target. Registers tested are AX, BX, CX, DX, DI, SI, and BP.

Indirect Jumps With No Displacement -- A register points to the start of the jump table. A second register is used to form an offset within the table. A jump is then performed to the word to which these registers point. Registers tested are BX, SI, and DI.

Indirect Jumps With 16-Bit Displacement -- Two registers are used to form a 16-bit displacement from the start of the jump table. A jump is then performed to the word to which these registers point. Registers tested are BX, SI, DI, and BP.

82510 UART SUBTEST

This subtest verifies the function of the 82510 UART in internal loopback mode.

The 82510 UART provides a serial link between the 386 node board and the Unit Services Module (USM). This channel is used by the Node Boot Monitor to communicate with the Cube Manager. Two tests are performed on the 82510 UART. These tests are written in PL/M 86 and are described in the following paragraphs:

Power-Up Default Test -- A software reset is performed on the device. For each register bank, the power-up default register values (of the registers that have valid power up defaults) are verified. The registers that contain valid power-up defaults and the values of those defaults are shown in Table A-5.

Table A-6. Default UART Register Contents (in hexadecimal)

	Bank 0	Bank 1	Bank 2	Bank 3
Reg 0	XX	XX	XX	00
Reg 1	00	XX	00	04
Reg 2	01	21	41	61
Reg 3	00	30	00	84
Reg 4	00	00	0C	FC
Reg 5	60	00	00	0F
Reg 6	XX	XX	1E	00
Reg 7	00	XX	00	XX

Internal Loopback Test -- The UART is initialized to internal loopback mode and a transmit/receive test is performed. The initialized configuration used during testing is as follows:

1. 16X Receive, 16X Transmit
2. 16X divisor of 0001H
3. 8-bit character, 4 stop bits
4. Large Sampling Window
5. SCLK pin as baud-rate clocking source
6. Receive FIFO threshold of 1 byte
7. Receive FIFO interrupts enabled
8. Framing, Parity, and Overrun Error interrupts enabled
9. Auto interrupt acknowledge
10. Flush receive FIFO, reset receive machine, enable reception
11. Flush transmit FIFO, reset transmit machine, enable transmission

The internal loopback test uses a variety of test patterns to ensure that the transmit and receive registers and FIFOs are operating correctly. The test patterns are listed in Table A-6.

Table A-7. UART Internal Loopback Test Patterns (in hexadecimal)

00	01	02	04	08	10	20	40	80	A5	55	33	0F	E3

As each pattern is transmitted, it is internally looped back and received by the receive register. No data are actually transmitted on the TXD pin. After a character is transmitted, the UART Line Status Register (LSR) is polled for a receive FIFO interrupt. Framing, parity, or overrun errors are not allowed. If the interrupt occurs, the received data pattern is read from the FIFO and compared to the transmit data pattern. If the patterns match, the loop is repeated for the complement of the data pattern. This process continues until all patterns are tested.

82258 ADMA SUBTEST

This subtest verifies the channel registers and the function of each channel of the 82258 ADMA.

The 82258 ADMA Subtest is written in PL/M 86 and tests the 82258 ADMA in two parts as described in the following paragraphs:

Channel Register Data Patterns Test -- This test verifies the capability of the 82258 channel registers to latch and hold a variety of test patterns and the complements. No deviations are allowed. Each test pattern is written to the channel registers. The registers are then read and compared with the pattern written. If the patterns match, the test pattern is complemented and tested again. Test patterns used are 0000H and 5555H.

DMA Transfer Test -- This test verifies the DMA transfer capability by performing a transfer for each channel in the 82258. This transfer is accomplished by initializing the ADMA for a two-cycle DMA transfer. A source data block (16 bytes) is then filled with the Status Table memory test pattern. The destination block is cleared. The DMA is started by a write to the General Command Register (GCR). The DMA is allowed to occur and the status returned in the General Status Register is verified. The Channel Status Register and the command block status are also verified. Finally, the source and destination data blocks are compared. If any of the above checks does not yield the expected results, the subtest returns a fail. The entire process is then repeated with the complement of the memory test pattern.

8259A MASTER PIC SUBTEST

This subtest verifies the internal mask register and the capability of the Master 8259A PIC device to receive and recognize interrupts. It is important to note that, during this subtest, the 386 microprocessor does not actually process interrupts. If an interrupt did occur, the 386 microprocessor would execute a far (intersegment) call to the interrupt handler, causing the CS register to be updated. When this intersegment call is executed, address lines A20-A31 drop low for CS-relative memory cycles, and the 386 microprocessor executes instructions in the lower one megabyte of physical memory, causing the EPROMs to be essentially lost. To avoid this situation, the 386 microprocessor INTR input is disabled before the NCT is executed.

This subtest is written in PL/M 86 and is performed in two parts. Part One tests the mask register for its capability to handle a variety of patterns. Part Two tests the PIC for its capability to receive and recognize interrupts. The parts are described in the following paragraphs:

Mask Pattern Test -- This test verifies the capability of the PIC mask register to latch and hold a variety of test patterns and complements. No deviations are allowed. Each test pattern is written to the Interrupt Mask Register. The register is read and its contents compared with the pattern written. If the patterns match, the test pattern is complemented and tested again. Mask register test patterns are 00H and 55H.

Interrupt Test -- This test verifies the capability of the PIC to recognize a hardware interrupt request. Because of the hardware implementation of interrupts, this test is *hand-tailored* for each individual interrupt source. The types of interrupt tests are as follows:

Timer 0 Interrupt -- The 8254 PIT Timer 0 is first initialized to interrupt on terminal count (mode 0). This initialization causes the Timer 0 Interrupt line to drop low. The Master PIC Interrupt Request Register (IRR) is read to ensure that the interrupt is not present. Timer 0 is then issued a start count of 1000 that, with the 1-megahertz timer clock, should interrupt in approximately 1 millisecond. A timeout loop is then entered that reads the IRR and checks for the Timer 0 interrupt. If this loop times out, the test fails. If the interrupt is detected before the timeout loop expires, the test passes.

End Of Data 0 Interrupt -- The 82258 ADMA is first initialized. The Master PIC Interrupt Request Register (IRR) is read to ensure that the End Of Data 0 interrupt is not present. The ADMA channel command block is then initialized to perform a DMA transfer of one word. The ADMA is issued a command to start the DMA transfer on channel 0. A timeout loop is entered that reads the IRR and checks for the End Of Data 0 interrupt. If this loop times out, the test fails. If the interrupt is detected before the timeout loop expires, the test passes.

End Of Data 1 Interrupt -- The Master PIC Interrupt Request Register (IRR) is read to assure that the End Of Data 1 interrupt is not present. The ADMA channel command block is then initialized to perform a DMA transfer of one word. The ADMA is issued a command to start the DMA transfer on channel 1. A timeout loop is entered that reads the IRR and checks for the End Of Data 1 interrupt. If this loop times out, the test fails. If the interrupt is detected before the timeout loop expires, the test passes.

8259A SLAVE PIC SUBTEST

This subtest verifies the internal mask register and the capability of the Slave 8259A PIC device to receive interrupts. It is important to note that, during this subtest, the 386 microprocessor does not actually process interrupts. If an interrupt occurs, the 386 microprocessor executes a far (intersegment) call to the interrupt handler, causing the CS register to be updated. When this intersegment call is executed, address lines A20-A31 drop low for CS-relative memory cycles, and the 386 microprocessor executes instructions in the lower 1 megabyte of physical memory, causing the EPROMs to be essentially lost. To avoid this situation, the 386 microprocessor INTR input is disabled before the NCT is executed.

This subtest is written in PL/M 86 and is performed in two parts. Part One tests the mask register for its capability to handle a variety of patterns. Part Two tests the PIC for its capability to receive and recognize interrupts. The parts are described in the following paragraphs:

Mask Pattern Test -- This test verifies the capability of the PIC mask register to latch and hold a variety of test patterns and complements. No deviations are allowed. Each test pattern is written to the Interrupt Mask Register. The register is read and its contents compared with the pattern written. If the patterns match, the test pattern is complemented and tested again. Test patterns are 00H and 55H.

Interrupt Test -- This test verifies the capability of the PIC to recognize a hardware interrupt request. Because of the hardware implementation of interrupts, this test is *hand-tailored* for each individual interrupt source. The types of interrupt tests are as follows:

End Of Data 2 Interrupt -- The 82258 ADMA is first initialized, then both the Master and Slave PIC IRRs are read to make certain that the Slave PIC and End Of Data 2 Interrupts are not present. The ADMA channel command block is then initialized to perform a DMA transfer of one word, and the ADMA is issued a command to start the DMA transfer on channel 2. A timeout loop is entered that reads the Master PIC IRR and checks for the Slave PIC Interrupt. If this loop times out, the test fails. If the interrupt is detected before the timeout loop expires, the Slave PIC IRR is read and a check is made for the End Of Data 2 Interrupt. If this interrupt did indeed occur, the test passes; otherwise, the test fails.

End Of Data 3 Interrupt -- Both the Master and Slave PIC IRRs are read to make certain that the Slave PIC and End Of Data 3 Interrupts are not present. The ADMA channel command block is then initialized to perform a DMA transfer of one word, and the ADMA is issued a command to start the DMA transfer on channel 3. A timeout loop is entered that reads the Master PIC IRR and checks for the Slave PIC Interrupt. If this loop times out, the test fails. If the interrupt is detected before the timeout loop expires, the Slave PIC IRR is read and a check is made for the End Of Data 3 Interrupt. If this interrupt did indeed occur, the test passes; otherwise, the test fails.

Timer 1 Interrupt -- The 8254 PIT Timer 0 is first initialized as a rate generator (mode 2). A start count of 1 is issued to Timer 0 such that, with the 1- megahertz timer clock, the timer produces a Timer 0 output pulse of 500 kilohertz. The 8254 PIT Timer 1 is then initialized to interrupt on terminal count (mode 0). This initialization causes the Timer 1 Interrupt line to drop low. Both the Master and Slave PIC IRRs are then read to make certain that the Slave PIC and Timer 1 Interrupts are not present. Timer 1 is issued a start count of 1000 that, with the 500-kilohertz clock generated by Timer 0 applied to Timer 1, should interrupt in approximately 2 milliseconds. A timeout loop is then entered that reads the Master PIC IRR and checks for the Slave PIC Interrupt. If this loop times out, the test fails. If the interrupt is detected before the timeout loop expires, the Slave PIC IRR is read and a check is made for the Timer 1 Interrupt. If this interrupt did indeed occur, the test passes; otherwise, the test fails.

Timer 2 Interrupt -- The 8254 PIT Timer 2 is first initialized to interrupt on terminal count (mode 0). This initialization causes the Timer 2 Interrupt line to drop low. Both the Master and Slave PIC IRRs are read to make certain that the Slave PIC and Timer 2 Interrupts are not present. Timer 2 is then issued a start count of 1000 that, with the 1-megahertz timer clock, should interrupt in approximately 1 millisecond. A timeout loop is then entered that reads the Master PIC IRR and checks for the Slave PIC Interrupt. If this loop times out, the test fails. If the interrupt is detected before the timeout loop expires, the Slave PIC IRR is read and a check is made for the Timer 2 Interrupt. If this interrupt did indeed occur, the test passes; otherwise, the test fails.

UART Interrupt -- The UART is first initialized to internal loopback mode. A Serial Channel Interrupt is then generated by writing 5AH to the Transmit Data Register of the UART. A timeout loop is entered that reads the Master PIC IRR and checks for the Slave PIC Interrupt. If this loop times out, the test fails. If the interrupt is detected before the timeout loop expires, the Slave PIC IRR is read and a check is made for the Serial Channel Interrupt. If this interrupt did indeed occur, the test removes the interrupt by reading the Receive Data Register of the UART. A timeout loop is then entered that reads the Master PIC IRR and checks for the Slave PIC Interrupt to disappear. If this loop times out, the test fails. If the interrupt does disappear before the timeout loop expires, the Slave PIC IRR is read and a check is made to verify that the Serial Channel Interrupt does not exist. If this interrupt did indeed disappear, the test passes; otherwise, the test fails.

387 NPX SUBTEST

The 387 NPX Subtests check the capability of the 387 Numeric Processor Extension to perform various mathematical operations using long real variables.

The 387 NPX Subtest is not executed if a coprocessor other than the 387 is detected. The presence of the another coprocessor, such as the Weitek module, is determined by reading the Modem I/O Pins Register (MSR) in the 82510 UART. A bit in this register contains the status of the Data Carrier Detect (DCD) pin on the UART. If this bit is set, the Weitek module is installed and the 387 NPX Subtest is not executed.

This test is written in both PL/M 86 and ASM 86. If no other coprocessor is installed, the 387 is tested in the following sequence:

387 Initialization Check -- The 387 is initialized by executing an FNINIT (Initialize Processor) instruction. The 387 control word is then read and compared to the default value of 037FH. If the control word matches, this test passes; otherwise, it fails.

Summation Test -- The following loop is executed:

```
a = 0.0; b = 0.0;
DO i = 1 TO 5;
   b = b + 1000.0;
   a = a + b;
END;
IF a <> 15000.0 THEN result = FAIL;
```

Anti-Summation Test -- The following loop is executed:

```
b = 0.0;
DO i = 1 TO 5;
   b = b + 1000.0;
   a = a - b;
END;
IF a <> 0.0 THEN result = FAIL;
```

Negative Summation Test -- The following loop is executed:

```
a = 0.0; b = 0.0;
DO i = 1 TO 5;
   b = b - 1000.0;
   a = a + b;
END;
IF a <> -15000.0 THEN result = FAIL;
```

Anti-Negative Summation Test -- The following loop is executed:

```
b = 0.0;
DO i = 1 TO 5;
   b = b + 1000.0;
   a = a + b;
END;
IF a <> 0.0 THEN result = FAIL;
```

Double Test -- The following loop is executed:

```
a = 0.0; b = 100.0;
DO i = 1 TO 10;
   a = a + b; b = a;
END;
IF a <> 51200.0 THEN result = FAIL;
```

Divide And Subtract Test -- The following loop is executed:

```
DO i = 1 TO 10;
   a = a / 2.0;
   b = b - a;
END;
IF (b <> a) OR (a <> 50.0) THEN result = FAIL;
```

Factorial Test -- The following loop is executed:

```
DECLARE
   a_ptr POINTER,
   answer BASED a_ptr (1) WORD;
a_ptr = @a;
a = 1.0;
b = 2.0;
DO i = 1 to 9;
   a = a * b;
   b = b + 1.0;
END;
IF (answer(0) <> 7C00H) OR (answer(1) <> 4A5DH) THEN result
= FAIL;
```

Anti-Factorial Test -- The following loop is executed:

```
DECLARE
   a_ptr POINTER,
   answer BASED a_ptr (1) WORD;
DO i = 1 TO 10;
   b = b - 1.0;
   a = a / b;
END;
IF (answer(0) <> 0) OR (answer(1) <> 03F80H) THEN result =
FAIL;
```

PARITY SUBTEST

This subtest verifies the proper operation of the parity circuitry on the 386 node board. This subtest also ensures that data error interrupts are recognized by the Master 8259A PIC.

The Parity Subtest is executed in a manner similar to that of the RAM Subtest. The Parity Subtest uses the RAM Subtest start and end pointers from the Status Table to determine its testing limits. For more information on the Status Table, refer to the section in this chapter entitled "NCT Error Indications".

This subtest, unlike the RAM Subtest, tests only the first word on each 16K-byte boundary from the start pointer to the end pointer. This is sufficient to test the Parity Error Generation and Detection Logic. It does, however, leave much of the parity RAM untested. Testing each location would require too much time for the NCT to execute. Parity Testing is slow because of the number of port I/O accesses required to read the master PIC status and the number of wait states generated for each I/O bus cycle on the 386 microprocessor.

NOTE

The CDP Node Standalone RAM Tests check Parity RAM.

The Parity Subtest is written in PL/M 86 and ASM 86. The subtest begins by first clearing any existing parity errors and verifying that no errors are present. For each word at the beginning of every 16K-byte RAM boundary from the start pointer to the end pointer, the following loop is executed:

A parity error is generated on the even byte of the word. This is accomplished by writing to On-Board Control Register # 1 and activating the PFORCE signal to force bad parity. A test pattern of 55H is written to the test location. This pattern contains an even number of 1 bits. A write to On-Board Control Register # 1 disables the PFORCE signal. The test location is then read, which causes the Data Error Interrupt (DERRINT) signal to become active. The Master PIC Interrupt Request Register (IRR) is read and a check is made to determine if DERRINT is received. The parity error is then cleared from the Master PIC by writing to On-Board Control Register # 1 and asserting the parity clear (PCLR) signal and then de-asserting it with another write to the same register. The Master PIC IRR is read again to make certain that the DERRINT signal is not present. This test process (generate error, verify, clear, and verify) is then repeated again for the odd byte of the word.

16 - 1023K RAM SUBTEST

This subtest verifies DRAM locations from address 16K to 1023K inclusive. The algorithm used by the RAM Subtest is often referred to as a verify/complement/verify test. The RAM to be tested is filled with an initial background pattern, and the following loop is executed for each location:

Read the location and verify that the old pattern is still present. Complement the pattern in the location. Read the location again. Verify that the complement pattern is still present.

Using this method, both address and data lines are tested in addition to the DRAM devices.

The 16K - 1023K RAM Subtest is written in PL/M 86 and ASM 86. The memory pattern used during testing is the Status Table memory test pattern variable. The default pattern is A5A5A5A5H and its complement. The start and end test pointers are also taken from the Status Table. For more information on the Status Table, refer to the section in this chapter entitled "NCT Error Indications".

The memory is tested in 16K-byte blocks using 32-bit data accesses. This is accomplished in ASM 86 by using a define byte (DB) assembler directive and placing an operand size prefix in front of the instructions that require 32 bits.

For each 16K-byte block from start pointer to end pointer, a call is made to fill the RAM with the background pattern. The memory test pattern is placed in the EAX register. The ES:DI register combination points to the location under test. The count (1000H dwords) is placed in the CX register and the memory is filled using the REP STOSW instruction.

After the background pattern is written, the verify/complement/verify loop is entered. For each 16K-byte block from the start pointer to the end pointer, a call is made to test the RAM. The memory test pattern is placed in the EAX register and its complement in EBX. The ES:DI register combination points to the location under test. The count (1000H dwords) is placed in the CX register. Each location is then compared with the value in the EAX register, complemented, and compared with the value in the EBX register. Then the DI register is incremented to point to the next location as the loop continues.

If all locations pass, the next 16K-byte block of RAM is tested in the same manner. This continues until all the RAM from the start pointer to the end pointer is tested. If the contents of any location do not match the contents of the EAX or EBX register, the test fails. After each 16K-byte block of RAM is tested, a check is made to see if a parity error is received by the Master PIC. If a parity error occurs, the parity error flag in the RAM Failure Table is set to FFFFH, the offset and segment of the 16K-byte block that causes the error are saved, and the test fails. For more information on the RAM Failure Table, refer to the section in this chapter entitled "NCT Error Indications".

CACHE RAM SUBTEST

The cache RAM Subtest verifies the functionality of the on-board cache circuitry. This subtest is written in PL/M 86 and ASM 86.

The cache RAM is tested in 16K-byte blocks using 32-bit data accesses. This is accomplished in ASM 86 by using a define byte (DB) assembler directive and placing an operand size prefix in front of the instructions that require 32 bits. The Cache RAM Subtest operation is as follows:

First, the cache is enabled and a 64K-byte block of RAM (the size of the cache RAM) starting from 1000:0000 is read using a REP LODSW instruction. This read operation caches these 64K RAM locations so that the Cache RAM Subtest can be run.

Next, a data patterns test is performed on the cache RAM in the same manner as the 16K - 1023K DRAM Subtest. A call is made to fill the cache RAM with a background pattern.

A verify/complement/verify loop is then entered. For each 16K-byte block, a call is made to test the RAM. The memory test pattern is placed in the EAX register and its complement in EBX. The ES:DI register combination points to the location under test. The count (1000H dwords) is placed in the CX register. Each location is then compared with the value in the EAX register, complemented, and compared with the value in the EBX register. Then the DI register is incremented to point to the next location as the loop continues.

If all locations pass, the next 16K byte block of RAM is tested in the same manner. If the contents of any location do not match the contents of the EAX or EBX register, the test fails. This continues until the entire 64K-byte cache RAM is tested.

NCT Special Features

The NCT includes two Special Features that aid in component level debug of the node board: Text Fixture and ICE-386. These features are described in Appendix B.

Memory Map

Table A-6 supplies useful information about segment placement and address assignments in Version 1.2 of the NCT.

Table A-8. NCT V1.2 Segment Map

Start	Stop	Size	Align	Name	Class
00000H	00000H	0000H	G	*MEMORY*	*MEMORY*
00500H	0072BH	022CH	G	*STATUS*	*STATUS_TABLE*
0072CH	0073FH	0014H	W	*RAM*	*DATA*
00740H	00755H	0016H	W	*IC*	*INCLUDE_TABLE*
00760H	008BFH	0160H	G	*DATA*	*DATA*
008C0H	0092FH	0070H	G	*BOOTLOADER_DATA*	*DATA*
00F00H	00FFFH	0100H	W	*STACK*	*STACK*
F0000H	F5F1BH	5F1CH	G	*CODE*	*CODE*
F5F20H	F5F6CH	004DH	G	*INTERRUPT_CODE*	*CODE*
F5F70H	F5F70H	0001H	G	*GET_RESULT_CODE*	*CODE*
F5F80H	F62E8H	0369H	G	*BOOTLOADER_CODE*	*CODE*
F62F0H	F6306H	0017H	G	*K_NPX_CODE*	*CODE*
F6310H	F66BFH	03B0H	G	*RAM_KER_CODE*	*CODE*
F66C0H	F6851H	0192H	G	*RAM016K_CODE*	*CODE*
F6860H	F6869H	000AH	G	*SET_AX_CODE*	*CODE*
F6870H	F68D5H	0066H	G	*CHIP_CODE*	*CODE*
F68E0H	F69C5H	00E6H	G	*PLMLED_CODE*	*CODE*
F69D0H	F69D3H	0004H	G	*LIB_87_INIT*	*CODE*
F69E0H	F6A17H	0038H	W	*TESTS*	*TEST_TABLE*
FFF00H	FFF07H	0008H	W	*ICE*	*ICE_TABLE*
FFFF0H	FFFFFH	0010H	G	*PUT*	*CODE*

NCT SPECIAL FEATURES B

INTRODUCTION

This appendix describes two Special Features of the NCT that aid in component level debugging of the node board. The first feature uses a test fixture and the second feature uses the ICE-386.

USING THE NCT WITH A TEST FIXTURE

The node board provides a special hardware signal line that is checked by the NCT before testing is initiated. This line, TESTFIXT, is connected to pin 83 on the node board P1 (communications) connector and to the Data Set Ready (DSR) input of the 82510 UART. The TESTFIXT line is ordinarily pulled high by a pull-up resistor on the 386 node board. When the NCT begins execution, the UART reads the status of the DSR line.

In a normal iPSC/2 Cube Unit, the TESTFIXT line is not connected on the backplane. Thus, the pull-up resistor on the node board forces this signal high and the NCT executes normally. If the node board is installed in a special test fixture that shorts the TESTFIXT signal on the backplane to ground, the DSR input on the UART is pulled low. When the DSR input is low, the NCT turns on both the red and green LEDs and enters a loop that tests the board forever. In this mode, the green LED indicates that the NCT is passing and the red LED indicates the NCT is executing. When a failure is encountered, the green LED is turned OFF and the red LED transmits a flashing error message that is readable with a Diagnostic Light Pen. The Node Boot Monitor is active only if the failing test is not in the hardcore.

USING THE NCT WITH ICE™-386

The NCT is designed with ICE-386 debugging in mind. A set of hooks is included in the firmware to interface with an ICE-386 tool and allow easy control of NCT test execution. This interface is implemented through a set of tables in EPROM and RAM. Using an ICE-386 tool, the contents of these tables can be accessed and controlled, changing the test flow of the NCT. The following paragraphs describe these tables. Later paragraphs describe the hooks.

ICE™ Table

This table is so named because it is used as a guide for the ICE-386 interface. The ICE Table is the root of the table structure and contains the addresses of all of the other tables. It also includes a breakpoint used for ICE-386 debugging. The ICE Table is an EPROM-resident table located at address F000:FFF0FF00 and contains the four variables listed and briefly described in Table B-1.

Table B-1. ICE™ Table

Variable	Description
ice_break	This variable is the CS-relative offset of a breakpoint in the NCT mainline code. This breakpoint is used extensively during ICE 386 board debugging and is the only mainline breakpoint provided. An example of using the *ice_break* point is shown later in this chapter.
test_table	This variable is the CS-relative offset of the Test Table.
status_table	This variable is the DS-relative offset of the Status Table.
include_table	This variable is the DS-relative offset of the Include Table.

The first two variables in the table (*ice_break* and *test_table*) contain addresses relative to the CS register (F000H). The next two variables (*status_table* and *include_table*) contain addresses relative to the DS register (0000H). Each variable in the ICE Table is described in detail later.

Test Table

The Test Table is an EPROM-resident table that contains the start address of each subtest and the address of a null terminated ASCII string containing the name of each subtest. The word contents of the Test Table are CS-relative offsets. The contents of the Test Table are described in Table B-2.

Table B-2. Test Table

Variable	Description
test_0_start	This variable is the CS-relative offset to Test 0 code. Test 0 runs all included tests. Refer to the section entitled "Include Table" for details.
test_0_name	This variable is the CS-relative offset to Test 0 name. The name consists of ASCII bytes terminated with a Null.
test_1_start	This variable is the CS-relative offset to Test 1 code.
test_1_name	This variable is the CS-relative offset to Test 1 name. The name consists of ASCII bytes terminated with a Null.
test_n_start	This variable is 0, indicating the last test.There are more than two NCT subtests.
test_n_name	This variable is 0, indicating the last test. There are more than two NCT subtests.
boot_start	This variable is the CS-relative offset to the Node Boot Monitor. It follows the two Nulls that follow the last test.

The contents of the Test Table repeat for each subtest until a null entry is encountered. Following the null entry, there is one additional word that contains the start address of the Node Boot Monitor. The number of subtests in the NCT is the number of words in the Test Table before the null entry, divided by two.

Status Table

The Status Table is a RAM-resident table that contains information about the node board, NCT pass/fail result, and control variables used during ICE-386 debugging. This table provides the interface for ICE-386 board debugging, allowing advanced control over the NCT test flow. The Status Table is further described in Table B-3.

Table B-3. Status Table (*sheet 1 of 2*)

Variable	Description
node_id	This variable contains the backplane slot ID.
cube_dim	This variable contains the cube dimension (set to 0).
mem_size	This variable contains the memory size read from the Memory Size Status Register of the MMOX Module.
NCT_result	This variable contains the NCT pass/fail result (pass = FFFFH, fail = 0). This word also contains the individual subtest pass/fail result during ICE 386 debugging.
NCT_version	This variable contains the NCT Version number.
mode	This variable contains the Error Message Level (FFFFH = verbose, 0 = silent). Silent produces tight loops for ICE-386 debug.
ice_debug	This variable is used during ICE 386 board debugging. It is initialized to false (0) by the mainline before the NCT executes. During ICE 386 debugging, it is set to true (FFFFH) as part of the ICE initialization to inform the NCT mainline code that an ICE is being used for debugging. Refer to the section entitled "ICE Initialization" for further details.
test_index	This variable contains the number associated with the current subtest to be executed. It is initialized to 0000H to indicate that all included subtests are to be run. Refer to the section entitled "Include Table" for further details.
halt_on_error	This variable determines if the NCT should halt (true = FFFFH, default) or continue (false = 0) when an error is detected.
iteration_count	This variable indicates the number of NCT iterations (1 is the default). A value of 0 executes forever.
failure_count	This variable (initially 0) increments each time a subtest fails.
execution_count	This variable (initially 0) increments each time a subtest runs.
cache_failure	This variable indicates whether the Cache RAM Test failed (FFFFH) or passed (0). The Node Boot Monitor checks this variable and enables the cache RAM only if the test passes.
mem_test_pat	This variable contains the pattern used during the ADMA, RAM, and Cache Subtests. The default pattern is A5A5A5A5H.

Table B-3. Status Table (*Sheet 2 of 2*)

Variable	Description
start_offset	This variable, along with *start_segment*, points to the first location tested by the RAM and Parity Subtests. This start pointer must be a location on a 16K-byte boundary. The default for *start_offset* is 4000H because the first 16K bytes are tested by the 0 - 16K RAM Subtest and are reserved for data and stack.
start_segment	This variable is the Starting Segment for *start_offset* (above).
end_offset	This, variable along with the *end_segment*, points to the last location tested by the RAM Subtest and Parity Subtests. This end pointer must be a location on a 16K-byte boundary. The default for *end_offset* is FFFFH.
end_segment	This variable is the Ending Segment for *end_offset* (above).

Error Message Buffer

The NCT places all of its error messages into the Error Message Buffer, which occupies 512 bytes immediately following the Status Table. Except for the EPROM Checksum and 0 - 16k RAM Subtests, when a failure is detected and the mode flag in the Status Table is set to verbose (FFFFH), the contents of this buffer are valid. This buffer contains a null terminated ASCII string describing the NCT failure.

RAM Failure Table

The NCT RAM Subtest places information into the RAM Failure Table when an error is detected. The RAM Failure Table is mapped into memory immediately after the Message Buffer. The RAM Failure Table is further described in Table B-4.

Table B-4. RAM Failure Table

Variable	Description
parity_error	This variable indicates if the RAM Subtest detected a parity error. If 0, the error is not a parity error and the remainder of the RAM Failure Table is valid. If FFFFH, the RAM Subtest detected a parity error and only the pointer formed by the *failure_offset* and *failure_segment* words are valid. Note that the pointer will be rounded to the nearest 16k-byte boundary and is not necessarily the exact location that caused the parity error. Testing each location for parity errors requires too much time for the NCT.
failure_offset	This variable, along with *failure_seg*, points to the failing location detected by the RAM Subtest.
failure_seg	This variable is the Failure Segment for *failure_offset* (above).
expected_pat	This variable contains the expected pattern of the failing RAM location (valid only if *parity_error* [above] contains 0).
actual_pat	This variable contains the actual pattern of the failing RAM location (valid only if *parity_error* [above] contains 0).
XOR_pat	This variable contains the exclusive OR (XOR) of the actual and expected patterns (valid only if *parity_error* [above] contains 0). This indicates the failing bits in the RAM location.

Include Table

The Include Table is a RAM-resident table used during ICE-386 board debugging to ignore or recognize tests from the Subtest 0 (execute all included subtests) sequence. Before the NCT runs, each entry in the Include Table is initialized to FFFFH, indicating that all subtests are recognized and should be executed. The table repeats with one entry for each test. The number of tests is indicated by the number of entries in the Test Table. The Include Table is further described in Table B-5.

Table B-5. Include Table

Variable	Description
include_test_0	This variable must be left set to FFFFH.
include_test_1	When this variable is set to FFFFH, Test 1 will execute as part of the Test 0 sequence. If it is set to 0, Test 1 will not execute as part of the Test 0 sequence.
include_test_n	Same as Test 1 except for Test n. There is a separate entry for each Test defined in the Test Table.

Using The ICE™-386 Hooks

The NCT ICE-386 hooks are operated through the table structures defined above. To access these table structures, the node board 386 microprocessor must be replaced by an ICE-386 emulator. Because the node board being debugged is possibly defective and therefore in an unknown state, it is recommended that the user install the Optional Isolation Board (OIB) between the Processor Module (PM) and the node board. This will provide a level of isolation between the 386 microprocessor and the node board. Once the user is confident that the node board is correctly wired and cannot damage the 386 microprocessor, the OIB may be removed. Note that, when the OIB is used, the CLK2 of the 386 microprocessor is limited to a maximum of 16 MHz. Refer to the *ICE™-386 In-Circuit Emulation User's Guide* for more information. Once the ICE-386 is properly installed, power up the system, invoke the ICE-386 software, and activate the ICE tool.

The following paragraphs provide further details on using the ICE-386 hooks.

INITIALIZATION

The node board must be allowed to initialize before the ICE-386 hooks are used. This is accomplished by the NCT firmware on power up or reset. Thus, with ICE, allow the processor to run from its reset vector until it reaches a breakpoint. The address of this breakpoint is defined in the ICE Table. When the NCT initializes, the following operations are performed:

1. Disable Interrupts.
2. Disable Caching.
3. Execute the EPROM Checksum Subtest (for code).
4. Execute the 0 - 16K RAM Subtest (for data and stack).
5. Initialize the interrupt vector table.
6. Download an interrupt handler to RAM.
7. Initialize Master and Slave PICs.
8. Initialize the Status Table.
9. Initialize the Include Table.

If a problem occurs in any of the above operations before the breakpoint is reached, the ICE-386 hooks cannot be used reliably.

The final step in initializing the 386 node board is to notify the NCT that it is being debugged by an ICE-386. This is accomplished by setting the *ice_debug* flag in the Status Table to FFFFH. Again, the start of the Status Table is defined in the ICE Table.

SUBTEST CONTROL/STATUS VARIABLES

The NCT ICE-386 hooks are operated through the table structures previously defined. Below is a list of the control/status variables. These variables, when used properly, provide a simple but effective interface to the NCT. As each variable is described, consider not only its individual use, but how it may be used in conjunction with the other variables. This will allow you a more thorough understanding of the interface mechanism provided by the NCT ICE-386 hooks.

NCT Result

This variable contains the 386 NCT pass/fail result. A pass is indicated by an FFFFH value. A fail is indicated by a 0000H value. The *NCT_result* is initialized to *FAIL* by the NCT. During ICE-386 debugging, this variable is updated with the individual subtest pass/fail result.

Mode

This flag variable controls the error message level, which is is initialized to verbose (FFFFH). If the error message level is verbose (FFFFH) when a failure is detected by the NCT, a detailed error message describing the failure is written to the Error Message Buffer. If the error message level is silent (0000H), no error messages are written to the Error Message Buffer. Setting this flag to silent (0000H) provides faster loop execution when a failure is detected, which allows for tighter scope loops; but it provides less debugging information.

Test Index

This variable contains the number associated with the current subtest to be executed, and is initialized to 0000H, which indicates that Subtest 0 (all included subtests) should be run. During Subtest 0 execution, the value is incremented as each subtest is run. If the NCT fails (as indicated by *NCT_result*), *test_index* contains the number of the subtest that detected the failure. Note that for Subtest 0 execution, *test_index* contains the number of the subtest that detects the failure only if halt on error is enabled (FFFFH).

Halt On Error Flag

When initialized to true (FFFFH), this variable determines the error action taken by the NCT when a failure is encountered. True (FFFFH) indicates that the NCT should halt on error; false (0) indicates the NCT should continue on error.

Iteration Count

This variable contains the requested iteration count of the subtest indicated by *test_index*; that is, the number of times the *test_index* subtest is to be executed. The default value for the *iteration_count* is 1. An *iteration_count* of 0 executes the subtest forever. During ICE-386 debugging, when the ICE reaches the breakpoint, the *iteration_count* is automatically set back to 1.

Failure Count

This variable contains the *failure_count*. Initially set to 0, this variable is incremented each time a subtest returns a fail result. It is the user's responsibility to clear this counter between subtest executions, if so desired.

Execution Count

This variable contains the *execution_count*. Initially set to 0, this variable is incremented each time a subtest is executed regardless of the subtest pass/fail result. It is the user's responsibility to clear this counter between subtest executions, if so desired.

RAM SUBTEST CONTROL/STATUS VARIABLES

The variables described in the following paragraphs are used primarily during RAM Subtest execution.

Memory Test Pattern

This variable contains the pattern used during the 82258 ADMA, RAM, and Cache Subtests. The default *memory_test_pattern* is A5A5A5A5H.

Start Pointer

Two variables (*start_offset* and *start_segment*) form the Start Pointer, which contains the starting address of the first location tested by the RAM and Parity Subtests. This start pointer must point to a location on a 16K-byte boundary. The default for Start Pointer is 0000:4000H. Because the first 16K bytes have already been tested by the 0 - 16k RAM subtest and are reserved for stack and data, the Start Pointer must point to 0000:4000H or above. Also, the Start Pointer must point to an address that is less than the End Pointer.

End Pointer

Two variables (*end_offset* and *end_segment*) form the End Pointer, which contains the ending address of the last location tested by the RAM Subtest and is used by the Parity Subtest. This End Pointer must point to a location that is a multiple of 16k bytes from the Start Pointer. The default for End Pointer is F000:FFFFH. Also, the End Pointer must point to an address that is greater than the Start Pointer.

Parity Error

This variable indicates whether the RAM Subtest detected a parity error. If it is false (0), the error is not a parity error and the remainder of the RAM Failure Table is valid. If it is true (FFFFH), the RAM Subtest detected a parity error and only the Failure Pointer is valid. Note that, because parity errors are checked only after each 16k-byte block of RAM is tested, the *failure_segment* and *failure_offset* words form a pointer to the first location in the 16K-byte block of RAM that caused the parity error, NOT the exact location that caused the parity error.

Failure Pointer

Two variables (*failure_offset* and *failure_segment*) form the Failure Pointer. This pointer contains the address of the failing location detected by the RAM Subtest.

Expected Pattern

This variable holds the pattern that the failing RAM location is supposed to contain. This pattern is the same as the Memory Test Pattern or its complement. The variable is vaild only if *parity_error* is false (0).

Actual Pattern

This variable holds the pattern that the failing RAM location contained as read during the RAM Subtest. This variable is valid only if *parity_error* is false (0).

XOR Pattern

This variable contains the exclusive-OR (XOR) of the actual and expected patterns; that is, it indicates the failing bits in the RAM location as detected by the RAM Subtest. This variable is valid only if *parity_error* is false (0).

ICE™-386 EXAMPLES

The following paragraphs contain examples illustrating the features of the ICE-386 debugging hooks. The <cr> represents the the return key on the keyboard. The <delete> represents the delete key on the keyboard. The -> represents the ICE-386 prompt. Brackets, [ ], indicate that the offset of the variable enclosed within the brackets should be substituted.

For example, assume that the *ice_break* breakpoint in the Ice Table is found to be 0183H. In the command:

```
-> go til cs:0fff0[ice_table.ice_break]h<cr>
```

the user would actually type:

```
-> go til cs:0fff00183h<cr>
```

Initialization Example

To initialize the 386 node board and NCT for ICE-386 debugging, execute the following commands at the ICE-386 prompt:

```
-> reset unit<cr>
-> go til cs:0fff0[ice_table.ice_break]h<cr>
-> ord2 [status_table.ice_debug]p = 0ffffh<cr>
```

Subtest Execution Examples

The following paragraphs describe examples of running a subtest.

Run Subtest 0 Once -- This example illustrates one execution of Subtest 0 (run all included subtests). This is the same execution scheme used by the 386 NCT standalone (that is, without an ICE-386).

```
-> ord2 [status_table.test_index]p = 0<cr>
-> go<cr>
```

Note that the *status_table. iteration_count* is not initialized to 1 because this is done by the NCT before the breakpoint is reached. The ICE-386 maintains the previous breakpoint setting in an internal buffer. Because the breakpoint has previously been set by the initialization shown above, a simple **go** command without parameters is all that is required by the ICE-386.

Subtest 0 is executed and, when the ICE-386 reaches the breakpoint, execution halts. The *status_table. NCT_result* contains the Subtest 0 pass/fail result. If Subtest 0 fails, *status_table. test_index* contains the number of the subtest that detected the failure and *status_table. failure_count* is incremented. If the RAM Subtest is the one that failed, the RAM Failure Table contains valid information about the failure. If Subtest 0 passes, *status_table. test_index* contains a value that is one greater that the number of subtests in the NCT (not counting the 0 - 16k RAM Subtest). This occurs because, as each subtest is executed, the *test_index* variable is incremented until a null entry is found in the Test Table. In either case, pass or fail, *status_table. execution_count* is incremented.

Run Subtest 0 More Than Once -- Running Subtest 0 more than once is similar to the previous example. Simply execute a command from the ICE-386 to set the *status_table. iteration_count* to the desired loop count. For example,

```
-> ord4 [status_table.iteration_count]p = 2<cr>
-> ord2 [status_table.test_index]p = 0<cr>
-> go<cr>
```

executes Subtest 0 twice.

In another example,

```
-> ord4 [status_table.iteration_count]p = 0<cr>
-> ord2 [status_table.test_index]p = 0<cr>
-> go<cr>
```

executes Subtest 0 forever. An infinite execution loop is not recommended unless the *status_table. halt_on_error* flag is set. It is important to note, however, that even with halt-on-error enabled, if the test never fails, the ICE-386 breakpoint will never be encountered. This situation may be desired if an unknown 386 Node Board is being burned-in for several hours or if intermittent failures are suspect.

An infinite execution loop may still be broken with the ICE-386, however, by issuing the following command.

```
-> <delete>
-> halt<cr>
```

This forces the ICE-386 to stop executing NCT code, even though it has not yet reached the breakpoint. It is then possible to simply examine the execution and failure counters. For instance,

```
-> ord4 [status_table.failure_count]p<cr>
```

displays the current failure count. To restart the ICE-386 from where it halted, use the following command:

```
-> go<cr>
```

Run an Individual Subtest -- An individual subtest is executed in exactly the same manner as Subtest 0. Subtest 0 is, after all, simply an individual subtest that executes in a special way. The same commands and rules apply to any individual subtest. The following example shows four executions of subtest number seven.

```
-> ord4 [status_table.iteration_count]p = 4h<cr>
-> ord2 [status_table.test_index]p = 7h<cr>
-> go<cr>
```

Using The Halt-On-Error Flag

The halt-on-error flag controls the action taken by the NCT when a failure is detected. Setting the halt-on-error flag causes the NCT to abort the current test sequence when a failure is detected. That, in turn, results in the ICE-386 breakpoint being encountered. Clearing the halt-on-error flag causes the NCT to continue executing the current test sequence when a failure is detected.

For example, if a 386 Node Board is suspected of failing the RAM Subtest intermittently, the halt-on-error flag could be set and the iteration count could be cleared. With this setting, the RAM Subtest executes until the first failure is detected. This is shown in the following example:

```
-> ord2 [status_table.halt_on_error]p = 0ffffh<cr>
```

Keep in mind that, with halt-on-error disabled, the subtest continues executing until the iteration count is expired. With halt-on-error enabled, the subtest continues executing until the iteration count expires or a failure is detected. Therefore, it is possible for the NCT to execute a subtest indefinitely if the iteration count is 0 and the subtest never fails.

An infinite execution loop can always be broken with the ICE-386 by issuing the following command:

```
-> <delete>
-> halt<cr>
```

This causes the ICE-386 to stop executing NCT code, even though it has not yet reached the breakpoint. It is then possible simply to examine the execution and failure counters. For instance,

```
-> ord4 [status_table.failure_count]p<cr>
```

displays the current failure count. To restart the ICE-386 from where it halts, use the following command:

```
-> go<cr>
```

Ignoring/Recognizing A Subtest

Subtests may be ignored or recognized from the Subtest 0 test flow through the Include Table, which notifies the NCT whether a subtest is *included* in the Subtest 0 test flow. The Include Table has no effect if an individual subtest (that is, a subtest other than Subtest 0) is executed. To use the Include Table effectively, it is necessary to understand its effect on the counters in the main test loop within the NCT.
In Subtest 0, before an individual subtest is executed, a check is made to determine if the subtest is recognized. If the subtest is recognized (indicated by a FFFFH in the Include Table for the subtest), it is executed normally and the counters are adjusted accordingly. If it is ignored (indicated by a 0000H in the Include Table), it is not executed and the counters are not adjusted. It is important to realize that the counters are not adjusted for ignored tests. The Include Table contents are not checked if an individual subtest is executed.

It is therefore possible to hang the NCT in an infinite loop by attempting to execute Subtest 0 if all of the other subtests are ignored. An infinite loop may be broken, however, and the ICE-386 returned to its breakpoint by recognizing the subtest in the Include Table as shown below.

```
-> <delete>
-> halt<cr>
-> ord2 [include_table.subtest]p = 0ffffh<cr>
-> go<cr>
```

Ignoring a subtest is simply a matter of clearing the Include Table entry for that subtest, as in the following:

```
-> ord2 [include_table.subtest]p = 0<cr>
```

NODE BOOT MONITOR C

INTRODUCTION

This appendix describes the Node Boot Monitor. The monitor is located in the node board EPROM along with the Node Confidence Tests (NCTs). The monitor is invoked after the NCTs execute.

PURPOSE OF NODE BOOT MONITOR

The primary purpose of the Node Boot Monitor is to provide an interface through which a program may be downloaded into the node board memory and begin execution. This downloaded program is typically an operating system or another, more powerful, second-stage boot loader. The monitor may also be used to perform low level functions such as dumping the node board memory. These functions are described only to the detail required for use by the Cube Diagnostic Program (CDP).

BOOT MONITOR AND CDP

The CDP (Cube Diagnostic Program) communicates with the Boot Monitor using the USM *PIPE_MODE* (see the Diagnostic Channel Library in Appendix D). The CDP depends upon the following Boot Monitor features:

Echo Mode
Global Select
Dump Memory
Load Memory
Go

COMMUNICATION CONSTANTS

The CDP uses the constants listed in table C-1 while communicating with the Boot Monitor.

Table C-1. Boot Monitor Constants

Name	Value (Hex)
ECHO_PROMPT	8A
GSELECT_PROMPT	C0
DUMP_PROMPT	84
LOAD_PROMPT	81
GO_PROMPT	82
SYNC	F0
UFILL	FF
ACK	86

ECHO MODE

The CDP uses the Boot Monitor Echo Mode to execute the Diag. Link: Node Echo Test. The Echo Mode is used as follows:

1. The Node under test is selected for Global communications (see GLOBAL SELECT below).

2. The Boot Monitor for the selected node is set into Echo Mode by sending it an *ECHO_PROMPT* followed by a zero. After the Boot Monitor Echo Mode is selected the node sends each byte of data it receives back to the host.

3. The Node Boot Monitor is removed from the Echo Mode by sending it four bytes: *SYNC*, *UFILL*, *SYNC*, and *UFILL*. The *UFILL* bytes are required so that the USM will pass the SYNC bytes on to the nodes. Thus, the nodes receive two *SYNC*s that remove the nodes from the Echo Mode. If you want the node to echo a *SYNC*, send out *SYNC* and two *UFILL*s.

GLOBAL SELECT

The CDP uses the Boot Monitor Global Select feature for the Node Echo Mode as described earlier and for dumping node board memory (see the later paragraph entitled DUMP MEMORY). The Global Select feature is enabled by sending the Boot Monitor a *GSELECT_PROMPT* followed by a node ID. The Node ID sent denotes the Node that is placed into the Global Select Mode. Nodes in the Global Select Mode are allowed to talk as well as listen. Nodes that are not in the Global Select Mode are allowed to listen only.

DUMP MEMORY

The CDP uses the Boot Monitor Dump Memory feature to obtain data from the node board RAM. For example, Dump Memory is used to collect the Node Confidence Test status that indicates whether the node passed or failed. The Dump Memory feature is used as follows:

1. The node that is to perform the dump is selected for Global communications (see the later paragraph entitled GLOBAL SELECT). Only one node can dump memory at a time because only one node at a time is allowed to talk.

2. The node is sent a *DUMP_PROMPT*.

3. The node is sent four bytes indicating the size of the dump. The size of the dump is the number of bytes to be sent back to the host as soon as the dump process begins.

4. The node is sent four bytes indicating the starting address of the memory dump.

5. The node is sent an *ACK*. This starts the dump process and the Node Boot Monitor sends the host the first byte from the starting address. The address is then bumped to the next memory location in preparation for the next *ACK* received. As soon as the host sends the next *ACK*, the next byte is sent back to the host. The *ACK*, therefore, is used as a throttling mechanism between the host and node. The *ACK* / send byte sequence continues until number of bytes indicated by size (step 3) have completed.

LOAD MEMORY

The CDP uses the Boot Monitor Load Memory feature to load a diagnostic monitor program. After the diagnostic monitor is loaded, the Boot Monitor is commanded to execute with the Go feature (see the later paragraph entitled GO). The Load Memory feature is used as follows:

1. The node is sent a *LOAD_PROMPT*.
2. The node is sent four bytes indicating the size of the load. The size of the load is the number of bytes the host will send that are to be loaded into RAM.
3. The node is sent four bytes indicating the starting address of the memory load.
4. The bytes are then sent from the host to the node as two characters. As each byte is received, the Boot Monitor loads the bytes into RAM at contiguous memory locations beginning at the start address (step 4). The node saves the starting address for the Go prompt. The load is terminated by sending SYNC, UFILL, SYNC, UFILL to the nodes.

GO

The CDP uses the Boot Monitor Go feature to begin execution of the Diagnostic Monitor program. The program is first loaded by using the Load Memory feature described above. The Go feature is used by sending the Node Boot Monitor a *GO_PROMPT*.

DIAGNOSTIC CHANNEL LIBRARY D

INTRODUCTION

This appendix describes the Diagnostic Channel Library. The purpose of the library is first stated, then each library routine is described.

PURPOSE OF LIBRARY

The Diagnostic Channel Library contains software routines that allow a programmer easy access to low-level hardware features provided by the Unit Services Module (USM). These features permit node board resets, interrupts, and RS-422 communications without the requirement to understand the low-level details of the hardware.

PURPOSE OF USM

The USMs allow the SRM to perform the following via the Diagnostic Link:

1. Enable, disable, or reset individual nodes at the hardware level
2. Send a Nonmaskable Interrupt (NMI) to all nodes
3. Communicate with the nodes
4. Control the USM front panel LEDs

LIBRARY ROUTINES

Tables D-1 and D-2 summarize the calling sequence for the library routines and the constants used. Following the tables is a description of the routines, including the requirements for the use of each.

Table D-1. Calling Summary for Diagnostic Channel C Routines

Routine	Calling Sequence
usm_open	*status = usm_open (dimension)*
usm_close	*status = usm_close()*
usm_disable	*status = usm_disable (node)*
usm_enable	*status = usm_enable (node)*
usm_reset	*status = usm_reset (node)*
usm_nmi	*status = usm_nmi (usm)*
usm_getstat	*status = usm_getstat (usm, usm_status)*
usm_leds	*status = usm_leds (led_value, usm)*
wait_usm_intr	*status = wait_usm_intr (max_time)*
wait_no_usm_intr	*status = wait_no_usm_intr (max_time)*
usm_put	*status = usm_put (mode, buf, size)*
usm_get	*status = usm_get (buf, size)*

Table D-2. Library Constants

Name	Value
PASS	*0*
FAIL	*-1*
EQ_BAD_NODE	*100*
EQ_BAD_USM	*101*
EQ_BAD_DIM	*102*
EQ_BAD_MODE	*103*
EQ_BAD_STATUS	*104*
GLOBAL_SELECT	*-1*
ECHO_MODE	*1*
PIPE_MODE	*2*

LIBRARY INTRODUCTION

The library routines return a PASS or FAIL status value. In addition, the UNIX errno variable is set and an error message is loaded into the CDP Test Manager. The Test Manager is described in Appendix E. The combination of a pass/fail status indication and automatic interface with the CDP Test Manager message buffers makes coding simple. For example,

```
if (usm_put(PIPE_MODE, buf, sizeof(buf)) != PASS) ERROR;
```

The simple line of code above transmits the message in *buf* from the Diagnostic Channel. If a failure occurs, an error message is automatically generated by the CDP Test Manager. The Test Manager is described in Appendix E.

NOTE

The *usm_open* routine must be used first.

usm_open

usm_open

The usm_open routine establishes a link to the Diagnostic Channel.

Calling Sequence

```
int status;

  .
  .
  .
status = usm_open (dimension);
if (status != PASS) END_TEST;
```

Input Parameters

dimension The cube dimension; defined as a configurable option inside CDP.

Return Values

status *Indicates a PASS or FAIL.*

Description

The *usm_open* routine opens the Diagnostic Channel and sets its baud rate to the proper setting per configuration (either 19.2 or 38.4K baud). In addition, the asynchronous driver used inside UNIX is set into all of the proper modes (for example, NOFLSH). This routine must be used before any of the other library routines is used.

Errors

EQ_BAD_DIM Error if *dimension* is greater than 7.

Standard errors from UNIX open apply.

usm_close

usm_close

The *usm_close* routine removes the Diagnostic Channel link.

Calling Sequence

```
int status;

  .
  .
  .
status = usm_close ();
if (status != PASS) END_TEST;
```

Input Parameters

None.

Return Values

status	Indicates a *PASS* or *FAIL*.

Description

The *usm_close* routine closes the UNIX asynchronous driver to the Diagnostic Channel.

Errors

Standard errors from UNIX close apply.

usm_disable

usm_disable

The *usm_disable* routine disables nodes.

Calling Sequence

```
int status, node;

  .
  .
  .
status = usm_disable (node);
if (status != PASS) END_TEST;
```

Input Parameters

node	The node board that is disabled. If *node* is *GLOBAL_SELECT*, all node boards are disabled.

Return Values

status	Indicates a *PASS* or *FAIL*.

Description

The routine forces the USM to hold the indicated nodes board(s) reset line(s) low (disabled).

Errors

EQ_BAD_NODE	The *node* input parameter is invalid.

Standard errors from UNIX write apply.

usm_enable

usm_enable

The *usm_enable* routine enables nodes.

Calling Sequence

```
int status, node;

  .
  .
  .
status = usm_enable (node);
if (status != PASS) END_TEST;
```

Input Parameters

node	The node board that is enabled. If *node* is *GLOBAL_SELECT*, all node boards are enabled.

Return Values

status	Indicates a *PASS* or *FAIL*.

Description

The routine forces the USM to hold the indicated nodes board(s) reset line(s) high (enabled).

Errors

EQ_BAD_NODE	The *node* input parameter is invalid.

Standard errors from UNIX write apply.

usm_reset

usm_reset

The *usm_reset* routine resets nodes.

Calling Sequence

```
int status, node;

  .
  .
  .
status = usm_reset (node);
if (status != PASS) END_TEST;
```

Input Parameters

node	The node board that is reset. If *node* is *GLOBAL_SELECT*, all node boards are reset.

Return Values

status	Indicates a *PASS* or *FAIL*.

Description

The routine pulses the node(s) reset line(s). The node(s) are left enabled after the reset is pulsed.

Errors

EQ_BAD_NODE	The *node* input parameter is invalid.

Standard errors from UNIX write apply.

usm_nmi

usm_nmi

The *usm_nmi* routine sends a Non-Maskable Interrupt to the nodes.

Calling Sequence

```
int status, usm;

  .
  .
  .
status = usm_nmi (usm);
if (status != PASS) END_TEST;
```

Input Parameters

usm	The USM board that generates the NMI. If *usm* is *GLOBAL_SELECT*, all USMs will generate a NMI.

Return Values

status	Indicates a *PASS* or *FAIL*.

Description

The routine pulses the Non-Maskable Interrupt line (NMI) to each node board connected to the specified USM.

Errors

EQ_BAD_USM The *usm* input parameter is invalid.

Standard errors from UNIX write apply.

usm_getstat

usm_getstat

The *usm_getstat* routine returns the USM status value.

Calling Sequence

```
int status, usm, usm_stat;

  .
  .
  .
status = usm_getstat (usm, &usm_stat);
if (status != PASS) END_TEST;

/* valid status is now located in usm_stat */
```

Input Parameters

usm	The USM board that returns its status. If *usm* is *GLOBAL_SELECT*, the *usm_status* returned is the ORed result of the status all USMs.

Return Values

status	Indicates the *PASS* or *FAIL* status of the *usm_getstat* function. Unless the value of *status* is *PASS*, the *usm_stat* is undefined.
usm_stat	The actual status returned from the USM(s). There are 16 bits of *usm_stat* as shown in Figure D-1. The Timer 0 Failure, Timer 1 Failure, and Invalid PROM Checksum errors are detected only when the USM is powered-up. The Invalid command number and global return status errors should never occur when the Diagnostic Library code is used exclusively to communicate with the USM(s). The other errors are related to communications failures on the Diagnostic Channel.

usm_getstat *(cont.)*

usm_getstat *(cont.)*

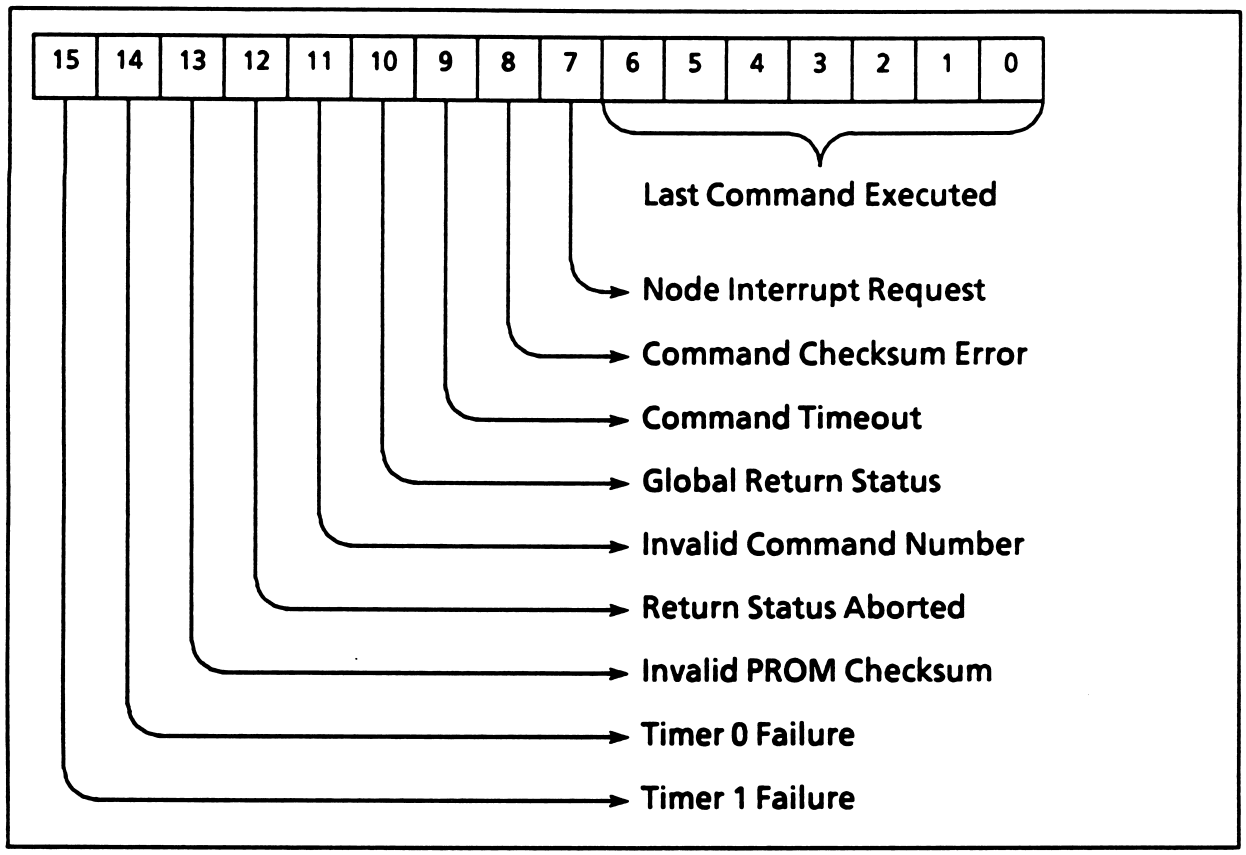


Figure D-1. USM Status

Description

The routine returns the USM status value. The status value includes errors, the state of the Node Interrupt Request line, and the last command executed.

Errors

EQ_BAD_USM	The *usm* input parameter is invalid.
EQ_BAD_STATUS	A communication failure occured while obtaining USM status.

Standard errors from UNIX write apply.

usm_leds

usm_leds

The *usm_leds* routine sets the USM LEDs state.

Calling Sequence

```
int status, led_value, usm;

  .
  .
  .
status = usm_leds (led_value, usm);
if (status != PASS) END_TEST;
```

Input Parameters

led_value	The state to which the USM LEDs are to be set. A value of 0 turns both LEDs off, 1 turns red on and green off, 2 turns red off and green on, and 3 turns both LEDs on. A value of 4 or greater sets the LEDs into *local control* of the USM.
usm	The USM board that sets its LEDs. If *usm* is *GLOBAL_SELECT*, all USMs set the LEDs.

Return Values

status	Indicates a *PASS* or *FAIL*.

Description

The routine sets the USM LEDs on the indicated USM to the specified state.

Errors

EQ_BAD_USM The *usm* input parameter is invalid.

Standard errors from UNIX write apply.

wait_usm_intr wait_usm_intr

The *wait_usm_intr* routine returns when a node requests an Interrupt.

Calling Sequence

```
int status, max_time;

  .
  .
  .
status = wait_usm_intr (max_time);
if (status != PASS) END_TEST;
```

Input Parameters

max_time The maximum time (seconds) that any node will be allowed to assert its Interrupt Request line.

Return Values

status Indicates a *PASS* or *FAIL*. A *PASS* indicates a node asserted Interrupt Request before *max_time* (seconds). A *FAIL* indicates that none of the nodes asserted in time.

Description

The routine waits as long as *max_time* (seconds) for any node board to assert an Interrupt Request.

Errors

timeout Indicates that all nodes failed to assert an Interrupt Request within the designated time.

Standard errors from UNIX write apply.

wait_no_usm_intr wait_no_usm_intr

The *wait_no_usm_intr* routine returns after all nodes stop interrupt requests.

Calling Sequence

```
int status, max_time;

  .
  .
  .
status = wait_no_usm_intr (max_time);
if (status != PASS) END_TEST;
```

Input Parameters

max_time The maximum time (seconds) that all nodes will be allowed to deassert the Interrupt Request lines.

Return Values

status Indicates a *PASS* or *FAIL*. A *PASS* indicates all nodes deasserted the Interrupt Request before *max_time* (seconds). A *FAIL* indicates at least one node did not deassert in time.

Description

The routine waits as long as *max_time* (seconds) for all node boards to deassert the Interrupt Request lines.

Errors

timeout Indicates that at least one node failed to deassert an Interrupt Request within the designated time.

Standard errors from UNIX write apply.

usm_put

usm_put

The *usm_put* routine sends a message out the diagnostic link.

Calling Sequence

```
int status, mode, size;  char buf[10];
  .
  .
mode = PIPE_MODE;
size = 10;
status = usm_put (mode, buf, size);
if (status != PASS) END_TEST;
```

Input Parameters

mode	Value is either *ECHO_MODE* or *PIPE_MODE*.
buf	A buffer of characters (bytes) containing the message.
size	The size of the message.

Return Values

status	Indicates a *PASS* or *FAIL*.

Description

The routine sends a message from the diagnostic link. If *ECHO_MODE* is specified, the message is echoed by the USM back to the SRM. If *PIPE_MODE* is specified, the message is piped through the USM directly to the nodes. Message sizes greater than 1 should not be used in *ECHO_MODE* because characters may be overwritten.

Errors

EQ_BAD_MODE	Must use *ECHO_MODE* or *PIPE_MODE* only.

Standard errors from UNIX write apply.

usm_get usm_get

The *usm_get* routine receives a message from the diagnostic link.

Calling Sequence

```
int status, size;  char buf[10];
  .
  .
size = 10;
status = usm_get (buf, size);
if (status != PASS) END_TEST;
```

Input Parameters

buf A buffer in which the received message is placed.

size The size of the received message.

Return Values

status Indicates a *PASS* or *FAIL*.

Description

The routine receives a message from the diagnostic link. The message may originate from either the USM or a node board.

Errors

timeout Indicates *serial_timeout* (seconds) elapsed before message reception.

Standard errors from UNIX write apply.

CDP TEST MANAGER LIBRARY E

INTRODUCTION

This appendix describes the Cube Diagnostic Program (CDP) Test Manager Library. The library is described in sufficient detail to allow a Diagnostic Engineer to write tests.

PURPOSE OF LIBRARY

The CDP Test Manager Library is the interface between CDP and its tests that execute on the nodes intended for testing Optional Hardware. It contains software structures, variables, and routines for interfacing tests with CDP. The Test Manager handles the Test Menu human interface and also handles accounting for failures, errors, and iteration counts. The tests that are written under the test manager are treated as Optional-Hardware Tests from the CDP main menu. Thus, hardware can be added to the system and tests may be written to exercise it by a test writer who knows nothing about the internal details of CDP itself. For example, an OEM customer may wish to write tests for hardware that will be added into the system.

LIBRARY THEORY OF OPERATION

There are two parts in the test manager design. One is the Cube Test Manager (CTM) and the other is the Node Test Manager (NTM). The CTM part resides on the SRM (Host) and the NTM part resides on each of the nodes. The CTM and NTM communicate across the host channel.

External Design Considerations

The external design of the test manager is viewed by the diagnostic user and the test writer. The user's view is the human interface and is consistent with CDP tests that run resident on the SRM. In other words, the CTM portion of CDP makes the human interface transparent to the test writer, thus relieving the test writer of this burden. The test writer's view of the design is more concerned with the NTM portion of the test manager. The test writer is not required to modify any portion of CDP or the test manager when writing a test or suite of tests.

The tests that are written under the test manager are treated as Optional-Hardware Tests from the CDP main menu. Thus, hardware can be added to the system and tests may be written to exercise it by a test writer who knows nothing about the details of CDP.

Internal Design Considerations

The focal point of the internal design of the test manager is with the CDP and CTM code. The CDP code handles the main menu function and passes program control to CTM when the Optional-Hardware Tests run. The test writer is not required to modify any portion of CDP or CTM when writing a test or suite of tests for new hardware. The test writer is required, however, to register the tests with the iSC Diagnostic Group so that tables internal to CDP can be enhanced to allow the new hardware keywords to exist in the */usr/ipsc/conf/cubeconf* file. These keywords are required so that CDP can load the appropriate executable file when required according to the users needs.

NOTE

The following parts of the system are used by NTM and CTM:

SRM Hardware
UNIX
CDP
Host Channel
Node Operating System
Standard parts of the node hardware

CTM Portion of Test Manager

The CTM portion of the test manager coordinates loading the Node Operating System, the test execution module, and handles all of the human interface. The interface consists of the following format while tests are running:

```
Node: <nnn> <Test Name> <Test Status>
```

The <nnn> field identifies the node number. The <Test Name> field is the name of the test as specified by the test writer. The <Test Status> field consists of one of the following words or phrases:

Running OK

Running with error

Passed

Failed

Timed out

In addition, the test writer may include as many as two error messages for the user when the test detects an error. One error message is displayed when the user selects the normal error message level and both are displayed when the user selects the verbose error message level. Standard messages must fit into a 256-byte buffer and verbose error messages must fit into a 512-byte buffer.

NTM Portion of Test Manager

The NTM portion of the test manager is used by a test writer for quickly and easily developing tests. It provides a means for identfiying the test, indicating an error and indicating a pass or fail condition. In this context, error and failure are different because a test may detect several errors before it eventually fails, depending upon the user's selection of the stop-on-error option.

The rest of this Appendix describes the NTM portion of the test manager in sufficent detail for a test writer. The NTM is located in a library named *ntm.a*. Table E-1 lists the tests that are currently available for optional hardware that are written under the Test Manager along with the load module names.

Table E-1. Tests Available for Optional Hardware Under the Test Manager

Keyword in config file	Load Module	Description
VX2000 VX3000 VX3100 VX4100	*VX.exe*	Vector Processor Tests.
EV	*EV.exe*	Evaluation Tests for testing the Test Manager.
387	*387.exe*	80387 Tests.
SX SXA	*SX.exe*	Weitek 1167 and 1167A Tests.
SCSI	*SCSI.exe*	SCSI module Tests.
DISK	*SCSI.exe*	Hard Disk Tests.
BIA	*BIA.exe*	Bus Interface Adapter Tests.

TEST IDENTIFICATION

The Test Manager must resolve a Test Descriptor Table (TDT) during the linkage step of compilation because the TDT identifies each test for the Test Manager. Figure E-1 shows the structure definition of a TDT along with descriptors for some of the SCSI Module Tests. In summary, each time a test is added to the system, an entry is made in the TDT to describe the test. The fields in the TDT are described in Table E-2.

```
struct
      test_descriptor_table {
         int   testlevel;
         char  testname[50]; /* use only 49 (leave one for a null) */
         int   (*test_ptr)(); /* pointer to function or "dummy" for menu */
         int   ext_test;   /* 0 = standard test, 1 = extended test */
         int   reserved;   /* used by NTM (set to 0) */
         int   trial_ctr;  /* used by NTM (set to 0) */
         int   error_ctr;  /* used by NTM (set to 0) */
         int   fail_ctr; } tdt[] = {
/*
lvl  testname                          testp     ext tbd tri err fail */
  1, "IO",                             (init),   101,  0,  0,  0,   0,
  2, "SCSI Module Tests",              (dummy),    0,  0,  0,  0,   0,
  0, "Controller Reset",    (reset_controller),    0,  0,  0,  0,   0,
  0, "Mode Select Teset,               (modet),    0,  0,  0,  0,   0,
  0, "FIFO Data Lines Test",         (ffdatat),    0,  0,  0,  0,   0,
            .
            .
            .
  -1 }; /* lastlevel terminates the TDT */
```

Figure E-1. Test Descriptor Table (TDT) with Diagnostic Link Tests Identified

Table E-2. Test Descriptor Table (TDT) Fields

TDT Field	Description
testlevel	One of four values: 1 is for first entry, 2 is for a menu entry, 0 is for a test entry, and -1 is for the last entry.
testname	The name of the test or menu as it will appear on the human interface. A test or menu name must be 49 characters or less in length so that there is at least one character left for a null.
test_ptr	A pointer to the test routine name. For example, if the test routine name is xyz(), then test_ptr is (xyz). All test routines (functions) that are not in the same module as the TDT structure definition must be defined as external before the TDT module will compile.
ext_test	One of two values: 0 indicates a standard test ; anything other than 0 indicates an extended test.
reserved	Initialize to zero. Reserved for possible future enhancement.
trail_ctr	Initialize to zero. Counts the number of trials a test has run.
error_ctr	Initialize to zero. Counts the number of errors detected by a test.
fail_ctr	Initialize to zero. Counts the number of test failures detected by a test. This field is maintained seperately because one test failure may encouter several errors, for example, if continue on error is set, before the test will fail.

LIBRARY ROUTINES

Tables E3 through E5 summarize the calling sequence for each library routine , the public variables used, and the constants used. Following the tables is a description of the routines, including the requirements for the use of each.

Table E-3. Calling Summary for Test Manager C Routines

Routine	Calling Sequence
ok	*abort_test = ok()*
err	*stop_test = err()*
silent	*flag = silent()*
terse	*flag = terse()*
normal	*flag = normal()*
verbose	*flag = verbose()*
first_run	*flag = first_run()*
prompt_mode	*flag = prompt_mode()*
prompt	*choice = prompt (min, max, default, message)*

Table E-4. Public Variables

Variable	Description
n_err_msg	A message buffer for normal level error messages. The message buffer is automatically emptied by the Test Manager when the err() routine is called.
v_err_msg	Same as *n_err_msg* except for verbose messages.
test_result	The PASS or FAIL result of a test. The *test_results* variable is automatically set to PASS before the test executes and is automatically set to FAIL when the err() routine is called.

Table E-5. Library Constants and Macros

Name	Value and Purpose
PASS	0: indicates no errors detected.
FAIL	-1: indicates at least one error detected.
MAX_N_ERR_MSG	256: size of the *n_err_msg* buffer.
MAX_V_ERR_MSG	32768: size of the *v_err_msg* buffer.
CHECK_IN	*if(ok()) return (test_result):* indicates test alive status.
ERROR	*if(err()) return (test_result):* indicates an error.
END_TEST	*return (test_result):* returns pass/fail status.

NOTES

The first step in writing a test for CDP is to define the test in the Test Manager TDT.

A test routine must return a PASS or FAIL status. It is recommended that you use the *test_result* public variable for this purpose. For example:

```
return (test_result);
```

Test routines should use the last three macros above as a standard. By using the *CHECK_IN*, *ERROR*, and *END_TEST* macros, you obtain consistent, easy to read test code, as in the following example:

```
example_test() /* demos usage of code macros */
{
   int pattern = 0xA5A5A5A5, xxx;

   CHECK_IN;

   xxx = pattern;

   if (xxx != pattern) ERROR;

   END_TEST;
}
```

ok

ok

The *ok* routine indicates that the test is still alive and whether to continue running.

Calling Sequence

```
int abort;
  .
  .
abort = ok();
if (abort) END_TEST; /* user wants to stop testing */
```

Input Parameters

None.

Return Values

abort	Value is set to *TRUE* if the user has pressed the test abort key; otherwise, it is set to *FALSE*.

Description

The routine indicates to the user that the test is still alive. It also provides a way for a test to prematurely terminate itself. The *ok()* routine should be called approximately every five seconds to assure the user that the test has not hung.

Errors

None.

err

The *err* routine handles test errors and indicates whether to continue running.

Calling Sequence

```
int stop;
  .
  .
sprintf(n_err_msg, "Normal Error Message");
sprintf(v_err_msg, "Verbose Error Message");
stop = err();
if (stop) END_TEST; /* user wants to stop after first error */
```

Input Parameters

n_err_msg	The Normal Error Message buffer is displayed for the user. This buffer is predefined as public by the Test Manager.
v_err_msg	The Verbose Error Message buffer is displayed for the user. This buffer is predefined as public by the Test Manager.

Return Values

stop	Value is set to *TRUE* if the user has pressed the test abort key or if the stop after first error option is set; otherwise, set to *FALSE*.
test_result	The *test_result* public variable provided by the Test manager is set to *FAIL*. The Test Manager sets *test_result* to *PASS* before a test executes.

Description

The routine indicates to the user that the test detected an error. The Test Manager automatically sets the public *test_result* variable to *FAIL* and unloads (which, in this context means that the data are removed from the buffers, sent to the host, displayed, and logged) the Normal and Verbose error message buffers (*n_err_msg* and *v_err_msg*). The buffers are then left empty (*NULL*). The error message buffers eventually reach the user and, therefore, must be in ASCII / English format.

Errors

None.

silent, terse, normal, and verbose

These routines indicate the Error Message Level setting.

Calling Sequence

```
.
.
/* an error detected */
if (silent() || terse()) { ERROR; END_TEST; }
if (normal()) sprintf(n_err_msg, "Normal Error Message");
if (verbose()) sprintf(v_err_msg, "Verbose Error Message");
ERROR;
```

Input Parameters

None.

Return Values

cdp_emsg The routines return *TRUE* if the indicated Error Message Level or higher is set. For example, *normal()* returns *TRUE* if Normal level is set or Verbose level is set; otherwise, it returns *FALSE*.

Description

These routines provide an alternative to always filling message buffers. For example, it may be desirable to obtain maximium performance from a test by having it avoid filling the error message buffers. This practice is not necessary, however, unless performance is an issue, because the Test Manager will display only the appropriate buffers, whether those are filling or not.

Errors

None.

first_run first_run

The *first_run* routine indicates whether the current iteration is the first time a test has run.

Calling Sequence

```
.
.
if (first_run()) initialize_hardware();
```

Input Parameters

None.

Return Values

first A *TRUE* return indicates that the current iteration is the first time the test has run. For example, if the user requested a test to perform 10 iterations, the first iteration would return *TRUE* and the other 9 iterations would return *FALSE*.

Description

The routine enhances test performance when more than one iteration of a test is performed because assumptions can be made about the hardware state. Perhaps, for example, the test could avoid writing a test pattern during all iterations except the first because the pattern would be written by the first iteration. and need not be written again It also makes most sense to prompt for test information only during the first run.

Errors

None.

prompt__mode prompt__mode

The *prompt__mode* routine indicates if the user has set *test__node* to a specific node number, thus, enabling the Prompt Mode.

Calling Sequence

```
.
.
if (prompt_mode())
  answer = prompt(0, 1, 0 , "Enter a 0 or 1");
else
  answer = 0;
```

Input Parameters

None.

Return Values

enb A *TRUE* returns if Prompt Mode is enabled; otherwise, *FALSE* returns.

Description

The routine indicates when Prompt Mode is enabled. The Prompt Mode is enabled when *test__node* is set to a specific node number in the range of 0 through 127.

Errors

None.

prompt prompt

The *prompt* routine prompts the user for test data.

Calling Sequence

```
int min = 0, max = 1, default = 0, answer;
char buf[100];
  .
  .
sprintf(buf, "Enter a 0 or 1");
if (prompt_mode())
  answer = prompt(min, max, default, buf);
else
  answer = default;
```

Input Parameters

min	The minimum value acceptable.
max	The maximum value acceptable.
default	The value acceptable by default if nothing is entered.
buf	A prompt message.

Return Values

value	The value entered by the user.

Description

The routine prompts the user with the message provided in *buf* and shows the minimum, maximum, and default values acceptable. Typically, an *if first_run* is placed in front of prompt calls so the user is prompted only once during multiple test trials. The prompt also provides a *Press <Return> to continue* message by setting *min* and *max* to the same value.

Errors

None.

LOOPBACK CONNNECTORS F

INTRODUCTION

This appendix describes loopback connectors. These connectors are used while running some of the CDP extended loopback tests. The connectors loop key signals from one connection in the system to another. Thus, CDP can stimulate an output and then verify an input for the expected result. This is useful in isolation of faults in cables through elimination of other hardware in the system. Typically, the use of loopback connectors and extended CDP tests is restricted to factory trained personnel.

STANDARD CABINET SRM CABLE LOOPBACK CONNECTOR

15 Pin Female DSUB Connector

pin	to	pin
1	----------	3
2	----------	4
5	----------	6
9	----------	11
10	----------	12
13	----------	14

NOTE

Serial Channel loopback pins are 5 to 6 and 13 to 14 only. All other connections are DCM Channel loopback pins.

COMPACT CABINET SRM INTERFACE BOARD LOOPBACK CONNECTOR

25 Pin Male DSUB Connector

pin	to	pin
1	----------	3
2	----------	4
5	----------	6
8	----------	9
14	----------	16
15	----------	17
18	----------	19
21	----------	22

NOTE

Serial Channel loopback pins are 8 to 9 and 21 to 22 only. All other connections are DCM Channel loopback pins.

This loopback connector may be used to test the SRM to cube cable by placing a female/female gender changer on the end of the cable.

BASE PLATE CABLE LOOPBACK CONNECTOR

20 Pin Male Cable Connector

pin	to	pin
1	----------	5
2	----------	6
3	----------	7
4	----------	8
9	----------	11
10	----------	12
15	----------	17
16	----------	18

NOTE

Serial Channel loopback pins are 15 to 17 and 16 to 18 only. All other connections are DCM Channel loopback pins.

COMM BOARD J15 (SERIAL CHANNEL) LOOPBACK CONNECTOR

10 Pin Female Cable Connector

pin	to	pin
5	----------	7
6	----------	8

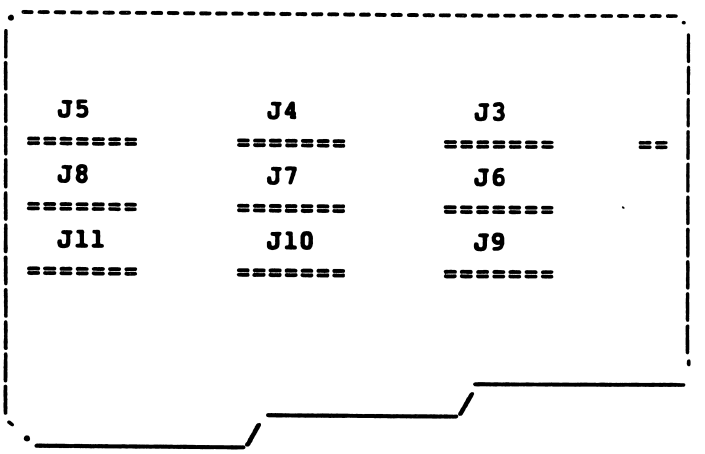
NOTE

This loopback connector may be used to test the Comm Board Serial Channel to the backplane by placing a male/male gender changer on the end of the cable.

STANDARD CABINET I/O BACKPLANE LOOPBACK TEST CONFIGURATION

Attach standard interconnect cables as follows:

```
CHANNEL 4 connections - J3-J5
CHANNEL 5 connections - J6-J8
CHANNEL 6 connections - J9-J11
```



19" BACKPLANE (back side)

TEST NOTES

There is a prerequisite that the cube under test be a D5, D4VX, or D3VX configuration (Vector boards not necessary). This is because the mapping of node pairs via the cables. A quick glance at the backplane schematic will illustrate this.

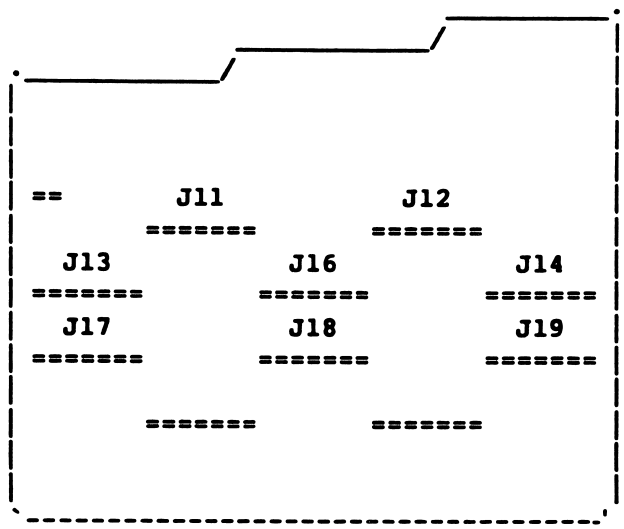
Verify that the backplane revision is specified correctly in the */usr/ipsc/conf/cubeconf* file. The revision shown in this file determines the pairs between which CDP will attempt to send messages.

COMPACT CABINET NEW BACKPLANE LOOPBACK TEST CONFIGURATION

Attach standard interconnect cables as follows:

```
CHANNEL 5 connections - J11-J12
                        J13-J14

CHANNEL 6 connections - J16-J18
                        J17-J19
```



NEW BACKPLANE (back side)

TEST NOTES

There is a prerequisite that the cube under test be a D5, D4VX, or D3VX configuration (Vector boards not necessary). This is because the mapping of node pairs via the cables. A quick glance at the backplane schematic will illustrate this.

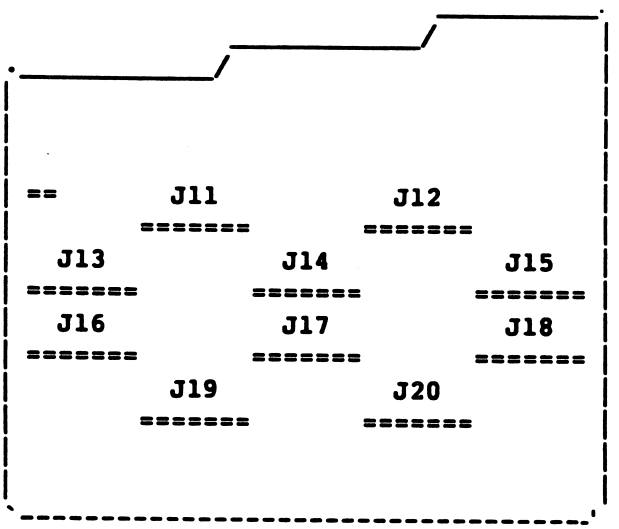
Verify that the backplane revision is specified correctly in the *usr/ipsc/conf/cubeconf* file. The revision shown in this file determines the pairs between which CDP will attempt to send messages.

COMPACT CABINET OLD BACKPLANE LOOPBACK TEST CONFIGURATION

Attach standard interconnect cables and loopback connectors as follows:

CHANNEL 5 connections - J11-J12
J13-J14
J15-Loopback Connector (shown on next page)

CHANNEL 6 connections - J16-J17
J19-J20
J18-Loopback Connector (shown on next page)



OLD BACKPLANE (back side)

TEST NOTES

There is a prerequisite that the cube under test be a D5, D4VX, or D3VX configuration (Vector boards not necessary). This is because the mapping of node pairs via the cables. A quick glance at the backplane schematic will illustrate this.

Verify that the backplane revision is specified correctly in the */usr/ipsc/conf/cubeconf* file. The revision shown in this file determines the pairs between which CDP will attempt to send messages.

OLD BACKPLANE DCM LOOPBACK CONNECTOR

60 Pin Female Cable Connector

pin	to	pin
1	----------	40
2	----------	39
3	----------	38
4	----------	37
5	----------	44
6	----------	43
7	----------	42
8	----------	41
9	----------	48
10	----------	47
11	----------	46
12	----------	45
13	----------	52
14	----------	51
15	----------	50
16	----------	49
17	----------	56
18	----------	55
19	----------	54
20	----------	53
21	----------	60
22	----------	59
23	----------	58
24	----------	57

This loopback connector may be used to test the Inter-Cube cable by placing a male/male gender changer on the end of the cable.

I/O COMM BOARD DCM LOOPBACK CONNECTOR

10 Pin Female Cable Connector

pin		pin
1	----------	10
2	----------	9
3	----------	8
4	----------	7
5	----------	6

NOTE

This loopback connector may be used to test the I/O Comm Board DCM Cable by placing a gender changer or converter on the end of the cable.

NODE P1 LOOPBACK CONNECTOR

96 Pin Male DIN Connector

pin		pin
B30	----------	C30

USM P2 LOOPBACK CONNECTOR

96 Pin Male DIN Connector

pin		pin
A28	----------	A30
A29	----------	A31